Analgesia Database:
C++ Program for PalmOS PDA

Version 0.95

J.M. van Schalkwyk

February 27, 2009

Contents

1 Introduction **9**
1.1 Structure of this document 9
1.2 Obtaining code — DogWagger 10

2 Error-handling routines **11**
2.1 Error includes and constants 11
2.2 Error function headers 13
2.3 Error Initialisation and Exit 14
2.3.1 Initialisation 14
2.3.2 Clean-up and exit 15
2.4 Error messaging — the meat! 15
2.4.1 ERRmsg: display coded error 16
2.4.2 ErrDisplay and ERRSTRING: avoid us! 17
2.4.3 ERRADD: Append error string 18
2.4.4 ERRINT: Append integer to error string 18
2.4.5 ERRSHOW: Display error string 18
2.4.6 ErrSetConsole 19
2.5 PalmOS error wrappers 20
2.5.1 Delete: free memory on exit 20
2.5.2 Close Database 20
2.5.3 Unlock memory 20
2.5.4 Release record 21
2.5.5 Reserve memory 21

2.5.6 Locate database . . . 21
2.5.7 Create Database . . . 21
2.5.8 Open Database . . . 22
2.5.9 Create new record . . . 22
2.5.10 Get record . . . 22
2.5.11 Write to a database record . . . 22
2.5.12 Unload a library . . . 23

3 Wrappers for PalmOS functions 24
3.1 Included files and class definition . . . 24
3.2 Memory wrappers . . . 27
3.2.1 Requesting memory . . . 27
3.2.2 Freeing memory . . . 27
3.2.3 Locking memory . . . 27
3.2.4 Unlocking memory . . . 28
3.2.5 Freeing a pointer . . . 28
3.2.6 Unlocking a pointer . . . 28
3.3 Text wrappers . . . 29
3.3.1 Getting a text character . . . 29
3.3.2 Writing a text character . . . 29
3.3.3 Finding the size of a character . . . 29
3.4 Database wrappers . . . 29
3.4.1 Finding a database record . . . 30
3.4.2 Releasing a database record . . . 30
3.4.3 Finding a named database . . . 30
3.4.4 Opening a database . . . 31
3.4.5 Closing a database . . . 31
3.4.6 Create a new record . . . 31
3.4.7 Create a new database . . . 32
3.4.8 Delete a database . . . 32
3.4.9 Write to a record . . . 33
3.4.10 Search for a sorted record . . . 33
3.4.11 Open a record for reading only . . . 34
3.4.12 Remove a record . . . 34
3.4.13 Resize a record . . . 34
3.4.14 Determine the size of a database . . . 35
3.5 String wrappers . . . 35
3.5.1 Find the length of a string . . . 35
3.5.2 Copy a string . . . 35
3.5.3 Concatenate two ASCIIZ strings . . . 36

3.5.4 Convert ASCII to an Integer . . . 36
3.5.5 Convert integer to ASCII . . . 36
3.6 Control wrappers . . . 36
3.6.1 Create a new control . . . 36
3.6.2 Set the label of a control . . . 37
3.6.3 Set the value of a control . . . 37
3.7 Form wrappers . . . 37
3.7.1 Kill current form, reactivate another . . . 37
3.7.2 Create a new form . . . 38
3.7.3 Delete a form . . . 38
3.7.4 Erase a form . . . 38
3.7.5 Display a modal dialogue . . . 38
3.7.6 Get the current, active form . . . 39
3.7.7 Get the ID of the active form . . . 39
3.7.8 Draw the form . . . 39
3.7.9 Go to a form . . . 39
3.7.10 Create a form label . . . 39
3.7.11 Get the index of an object in a form . . . 40
3.7.12 Get a pointer to a form object . . . 40
3.7.13 Count objects within a form . . . 40
3.7.14 Get the ID of a form . . . 41
3.7.15 Load and open form (without closing current) . . . 41
3.7.16 Remove object from form . . . 41
3.7.17 Show form object . . . 41
3.7.18 Set form title . . . 42
3.8 Field wrappers . . . 42
3.8.1 Create a new field . . . 42
3.8.2 Get a handle on the text in a field . . . 43
3.9 List wrappers . . . 43
3.9.1 Create a new list . . . 43
3.9.2 Set the choices for a list . . . 43
3.9.3 Set the drawing function for a list . . . 44
3.10 System and other wrappers . . . 44
3.10.1 Create an array of pointers to some ASCIIZ strings . . . 44
3.10.2 Clumsily turn ASCIIZ into floating point double . . . 45
3.11 xCopy — a useful function . . . 45

4 Fundamental routines **46**
4.1 Utility includes . . . 46
4.2 Utility class definition . . . 46

4.3 Utility: private functions and data 49
4.4 Utility ‘constructor’ and destructor 50
4.4.1 constructor: SetupUtility 50
4.4.2 Destructor: KillUtility 52
4.5 Console writing — a frill 54
4.6 General-purpose utility functions 58
4.6.1 xNew: request memory 58
4.6.2 Delete: free memory 59
4.6.3 Same: compare strings 60
4.6.4 xCompare: odd man out 60
4.6.5 Fill string with a byte value 61
4.6.6 Advance: find position of character 61
4.6.7 ReadInt32: endian read integer from pointer 61
4.6.8 WriteInt32: endian write 32 bit integer from string 62
4.6.9 Clumsy short integer read from text 62
4.6.10 Generate ‘unique’ ID 63
4.6.11 Crutch functions 63
4.7 Utility form functions 64
4.7.1 Dynamically create form 64
4.7.2 Activate dynamic form 65
4.7.3 Insert a new widget 65
4.7.4 Create a new form ‘control’ 66
4.7.5 List creation 71
4.7.6 Finding a list item by index 74
4.8 Utility field functions 75
4.8.1 Did a field change? 75
4.8.2 Get text from field 76
4.8.3 Set currently active field 77
4.8.4 Get ID of a field 77
4.9 Utility ‘database’ functions 77
4.9.1 Open PalmOS ‘file’ 78
4.9.2 Close PalmOS ‘file’ 78
4.10 Utility: the SELECT statement 79
4.11 Utility: the UPDATE statement 80
4.12 The INSERT statement 81
4.13 Utility: miscellaneous buffer functions 81
4.13.1 Create empty record 82
4.13.2 Create buffer file 82
4.13.3 Find PalmOS ‘file’ 83
4.13.4 Create stack 83

4.13.5 Close stack 84
4.14 Utility list/node functions 85
4.14.1 Create list node 85
4.14.2 Find list node 86
4.14.3 Insert list node 87
4.14.4 Delete all list nodes 87
4.14.5 Text nodes 88
4.15 Utility: console functions 89

5 Higher level routines **91**
5.1 Includes 91
5.2 Class definition 92
5.2.1 A note on reloading a sequence of menus 95
5.2.2 function list 95
5.3 Initialisation and clean-up 97
5.4 Menu handling 99
5.4.1 EnterMenu 100
5.4.2 FetchMenuDepth 100
5.4.3 WriteRestorationData 100
5.4.4 RestorePriorState 102
5.4.5 PostMenuEvent 105
5.4.6 NewMenu 105
5.4.7 SleepMenu 110
5.4.8 Rolling menus 111
5.5 Widget creation 112
5.5.1 CreateManyWidgets 112
5.5.2 CreateOneWidget 114
5.5.3 Populating a poplist 118
5.6 Monomorphic tables 119
5.6.1 CreateOneTable 119
5.7 Polymorphic tables 122
5.7.1 MakeWholeColumn 125
5.7.2 GetRowHeight & GetNumLines 128
5.8 Response handling 128
5.9 Some utility routines 131
5.9.1 PushString 131
5.9.2 PushInteger 131
5.9.3 PopFloat 132
5.9.4 PopInteger 132
5.9.5 PopString 133

5.9.6 PeekAndPopStringZ . . . 133
5.9.7 ResolveX . . . 134
5.9.8 FindDelimiter . . . 134
5.9.9 FindDelimiter2 . . . 135
5.10 Script execution — long! . . . 135
5.10.1 A test routine . . . 146
5.10.2 Caching commands . . . 146
5.10.3 MENU command . . . 147
5.11 Other execution . . . 152
5.11.1 ExeQuery . . . 153
5.11.2 ExeSQL . . . 154
5.12 Debugging-style routines . . . 155
5.12.1 InsertAndGo . . . 155
5.12.2 JustGo . . . 156
5.13 Script function handling . . . 156
5.13.1 IniFxBuffer . . . 157
5.13.2 CloseFxBuffer . . . 159
5.13.3 PushCurrentScript . . . 160
5.14 Alerts and interaction . . . 160
5.14.1 MakeTextForm . . . 160
5.14.2 KillTextForm . . . 161
5.14.3 RespondToAsk . . . 161
5.14.4 SayAlert . . . 162
5.14.5 Confirm . . . 163
5.15 Configuration and colour handling — rudimentary . . . 163
5.15.1 HexColour . . . 163
5.16 SetPaperOrInk . . . 164
5.16.1 HackWidget . . . 164
5.16.2 Enabled . . . 165
5.16.3 SetLabel . . . 166
5.17 Misc. Widget utilities . . . 168
5.17.1 MakeWidget . . . 168
5.17.2 PrependWidget . . . 168
5.17.3 KillOnId . . . 169
5.17.4 KillAllWidgets . . . 169
5.17.5 FindWidgetDatabaseCode . . . 170
5.17.6 FindWidgetType . . . 170
5.17.7 VValue . . . 170
5.17.8 FindWidgetUID . . . 171
5.18 Script variables . . . 172

5.18.1 MakeLocalVariables . . . 172
5.18.2 KillLocalVariables . . . 173
5.19 Various functions . . . 173
5.19.1 PopMenu . . . 173
5.19.2 PushMenu . . . 176
5.19.3 GoScript . . . 176
5.19.4 FindFunction . . . 177
5.19.5 DoReturn . . . 177
5.20 More debugging . . . 178
5.20.1 StackDump . . . 178
5.20.2 ShowStackItem . . . 179
5.21 More utilities — reorganize me . . . 181
5.21.1 PeekLengthPlus . . . 181
5.21.2 QuickBy . . . 181
5.21.3 PeekType . . . 182

6 Main section 183
6.1 The PilotMain function . . . 183
6.2 Starting and stopping . . . 185
6.3 Event loop . . . 188
6.4 Individual form event handling . . . 195
6.5 A debugger function . . . 196
6.6 Library loader . . . 196
6.7 Creating an SQL instance, and deleting it . . . 197
6.7.1 Killing the program . . . 199
6.8 Static callback functions . . . 199
6.8.1 myLISTDRAWER callback . . . 200
6.8.2 CompareRowKeys callback . . . 200

7 The Makefile, and other frills 202
7.1 The Makefile . . . 202
7.2 Header files . . . 203
7.2.1 pain5.h . . . 203
7.2.2 palmsql3.h . . . 204
7.2.3 palmsql3A.h . . . 206
7.2.4 painrcp.h . . . 209
7.3 RCP file . . . 209
7.4 The DEF file . . . 210

8 Appendix: PalmOS programming **211**
8.1 PDA programming — an introduction 212
8.1.1 A first program 213
8.2 A larger application 218
8.2.1 The Makefile 218
8.2.2 The code 220
8.2.3 The resource compiler 223
8.3 Debugging 227
8.4 Communications 227
8.5 Profiling 229
8.6 Conclusion 231
8.6.1 A small note on binary files 231

1 Introduction

This document describes in some detail our principal PDA program, written in C++ for PalmOS. It follows on previous documents describing the design of our Pain Database, including associated paper forms and the desktop Perl program, all of which should be read prior to reading these pages. My C++ program makes extensive use of my PalmOS *library* routines, all written in C (not C++). These are discussed in detail in separate documents, one for each library, obtainable with all of the other documentation at our website anaesthetist.com.[1]

Apart from assuming that you've read the preceding documents in the series, we also assume that you understand the difference between little and big-endian data storage, know about ASCIIZ, and know something of C and C++. Some familiarity with PalmOS (and its representation of numbers, handles and so forth) is also assumed.

I agonized for some time over which platform and programming language to use. First, I chose the Palm PDA because I've previously witnessed the travail a competent colleague went through with Windows CE when he tried to implement a similar project on tablet PCs some years ago.[2] Next I excluded Java (because it's a dog in terms of efficiency when implemented on PDAs), and although Tcl was attractive, I fixed on C because it's the native language of PalmOS. In retrospect, choosing C++ was perhaps not very wise, because it has certain unattractive features, particularly on the PDA. Using the intrinsic *new* function results in a whole lot of unwanted side effects, including code bloat. In addition, because I'm a procedural programmer at heart, I have tended towards a more procedural orientation to code-writing, only using C++ to structure my code a bit better — the last mentioned could have been performed equally well in C, with fewer overheads and more ability to control the code.

1.1 Structure of this document

I've chosen first to document error-handling routines (these are fundamentally important, and most programmers seem to relegate them to footnotes after they've finished their main documentation). We then examine a simple method of wrapping almost every PalmOS function within 'wrapper' functions of our own (some-

[1] The libraries are called ERRDEBUG, SCRIPTING, SQL3, NUMERIC, and CONSOLE, the last-named being exclusively for debugging. The console can be viewed using the stand-alone OSBOX program.

[2] This despite alleged improvements in WinCE, and the concern that MS will assimilate the PDA market and extinguish PalmOS with Borg-like efficiency. I've just had too many baad experiences with MS 'products', and the 500 Meg download for the Windows CE development kit does not reassure! Be careful whom you go to bed with.

thing which might facilitate porting our program to other systems). We've kept these functions to what we consider a minimum. Next we look at a grouping of utility routines, and after this we consider higher-level routines which implement SQL functionality, menu-drawing and a complete scripting language. Finally, we consider the *main* PalmOS function which ties everything else together.

Because I've used C++, I've implemented each of the above major sections as a class, which inherits the properties of the previous (more fundamental) class. Here are the classes, from simplest to most complex, in order of inheritance:

1. *err.hpp*
2. *wraps.hpp*
3. *utility.hpp*
4. *sql.hpp*
5. *pain5.cpp* (Well, not actually a class, as it's the main program)

You'll see that all bar the main program (*pain5.cpp*) have the suffix '.hpp' as I've included the class definition within the code, rather than having a separate header file.

1.2 Obtaining code — DogWagger

As usual, our DogWagger program (written in Perl) can be used to extract the entire C++ source code from the LaTeX (.TEX) version of this document. We introduce a new option in DogWagger 2.0, where a section of text (several lines) can be marked with a starting keyword: `+OPTIONAL`, and terminated with a corresponding `-OPTIONAL`. The contents of the lines starting with these keywords are ignored, but the intervening lines can be *conditionally* inserted into the files generated by DogWagger. The main purpose of this frill is to allow us to produce two versions of our code — a debug version and a production version. We specify the 'debug' version by simply inserting the text `include='everything'` into the first DogWagger comment line of the LaTeX source — otherwise DogWagger leaves out the optional lines (thus making the 'production' version).

2 Error-handling routines

These routines are fundamental to the operation of the program. We've tried to structure them so that when an error occurs, it can be identified and characterised fairly easily. The most vexing errors on the Palm are memory leaks, so we've devoted some ingenuity to identifying which routines are responsible for such leaks. You can regard this section as a debugging section, as well as an error-handling section.

Because the wrapper class is higher up the food chain than the error class, we here individually wrap calls to PalmOS functions in a similar way to our use of wrappers in *wraps.hpp* later on.[3]

2.1 Error includes and constants

We kick off with some mandatory PalmOS includes, and a few constant definitions (which might be moved to a separate header file).

```
#include <PalmOS.h>
#include "err/ERRDEBUG.h"  // library

#define TWOBUFFERSIZE           128
#define ErrorBufferName         "CYCERROR"
#define CYCSIZE                 32752

#define ErrBadBase              200 // base for err codes
#define ERRNoNumber             ErrBadBase+1
  // failed to set aside buffer area for magic number
#define ERRCannotCreate         ErrBadBase+2
  // cannot create CYCERROR cyclical data buffer
#define ERRStillNotFound        ErrBadBase+3
  // despite creation, buffer wasn't found. Peculiar.
#define ERRCannotOpen           ErrBadBase+4
  // cannot open cyclical data buffer
#define ERRNoHandle             ErrBadBase+5
  // cannot create handle on cyclical data buffer
```

TWOBUFFER is simply a buffer used for string display (we allow up to 127 characters). We also define a few local constants such as the name of the PalmOS database which we use as a cyclical error buffer, its size, and a few error-related

[3] There's a certain irony here. Because I appreciated from the onset that in most programs (and indeed, most languages) error-handling is given a second-rate status, I initially concentrated on writing these routines, in an attempt to structure everything around them, and make them robust. Unfortunately, at that stage my PalmOS C++ skills and overall view of the program were less than complete, so the error routines could still 'use a little work', despite my noble goals!

constants of dubious utility. As can be seen from what follows, all of the ERR constants are trivial, ugly, and infrequently used relics. CYCSIZE must be divisible by eight, as per its usage below.

2.2 Error function headers

```
class err
{ public:
           err  () {};
           ~err () {};
    Int16  ErrStart(UInt16 libcode);
    Int16  ErrEnd();

    Int16  LibraryUnload (UInt16 libcode);
    void   ERRDisplay(Char * p, Int32 c);
    void   ERRSTRING(Char * p, Int16 plen);
    void   ERRmsg(Int16 code);
    void   ERRSHOW(Int16 code);
    void   ERRADD (Char * p, Int16 plen);
    void   ERRINT (Int32 i);
    void   ErrSetConsole (UInt16 cons);

private:
    UInt16    ERRORLIBCODE;
    Char *    TWOBUFFER;
    Int16     TWOTOP;
    MemPtr    MAGICNUMBER;

    void      ErrMsgAndNumber(const Char * msg, Int32 nmbr);
    MemPtr    e_New ( Int16 memsize);
    LocalID   e_DmFindDatabase (const Char *nameP);
    Int16     e_DmCreateDatabase (const Char *nameP);
    DmOpenRef e_DmOpenDatabase (LocalID dbID, UInt16 mode);
    Int16     e_Delete (MemPtr memP);
    MemHandle e_DmNewRecord (DmOpenRef dbP, UInt16 *atP, UInt32 size);
    Int16     e_DmWrite (void *recordP, UInt32 offset,
                 Char *srcP, UInt32 bytes);
    MemHandle e_DmGetRecord (DmOpenRef dbP, UInt16 index);
    Int16     e_DmReleaseRecord (DmOpenRef dbP, UInt16 index,
                 Boolean dirty);
    Int16     e_MemHandleUnlock (MemHandle h);
    Int16     e_DmCloseDatabase (DmOpenRef dbP);
};
```

The constructor and destructor are unused, because (as discussed later in Section 4.6.1) we have chosen to define our own version of *new*, and this choice has many benefits but the slightly unfortunate side-effect of rendering the traditional C++ constructors and destructors non-functional. We thus define our own versions: ErrStart and ErrEnd.

There's a very short list of functions, and of these, ErrDisplay is used strictly for debugging. In fact, the list could be made even shorter with some skillful

rewriting, as the error functions have become extensively modified from their primitive beginnings, as our requirements changed!

The private variables are used as follows:

- ERRORLIBCODE references the error library (a PalmOS convention)
- TWOBUFFER is used to display strings. Several component strings can be appended to this buffer before it's displayed. TWOTOP indicates the current top of this buffer.
- MAGICNUMBER is used to write an ASCII error number.

Interestingly enough, all of the private functions bar one are wrapper functions for PalmOS native functions. The sole exception (ErrMsgAndNumber) is a utility function which simply displays a message.

2.3 Error Initialisation and Exit

2.3.1 Initialisation

ErrStart is our 'constructor'. We pass it a reference (libcode) to our PalmOS error library module, and store the value in ERRORLIBCODE. We also initialise a buffer in which we write integers (MAGICNUMBER), and we then open or create our cyclical error buffer, using the wrapped functions described below.

```
Int16 err::ErrStart (UInt16 libcode)
{ ERRORLIBCODE = libcode;
  MAGICNUMBER = e_New(maxStrIToALen);
  if (! MAGICNUMBER) { return ERRNoNumber; };
```

The code which used to sit here has all been exported to the error library.

```
  TWOTOP = 0;
  TWOBUFFER = (Char *) e_New(TWOBUFFERSIZE);
  if (! TWOBUFFER) // nasty error
     { return -1;  // fail
     };
  return 0;
}
```

Zero signal success, any other number returned is an error code.

2.3.2 Clean-up and exit

The following 'destructor' must be called just before we terminate our PalmOS program. The code closes the error library, frees up memory, and closes the cyclical error buffer. See how we pass `numappsP` by reference, allowing ERRDEBUGClose to insert a value of over zero into it on failure (Hmm).

```
Int16 err::ErrEnd ()
{ UInt16 err=0;
  UInt16 numappsP=0;

  if (! e_Delete(TWOBUFFER))
     { err ++; // bit 0
     };
  if (! e_Delete(MAGICNUMBER))
     { err |= 2; // bit 1
     };
  err |= ERRDEBUGClose(ERRORLIBCODE, &numappsP);
   // err value returns bit flags (bits 3..7), and
   //   num apps still open (although unused at present).
  if (!LibraryUnload(ERRORLIBCODE))
     { err |= 4;  // bit 2
     };
  ERRORLIBCODE = 0; // braces
  return err;
}
```

Note the usage of ERRORLIBCODE by the function ERRDEBUGClose. This tells us that ERRDEBUGClose is a call to a library function, in this case, the error library we've created. The error library is discussed in detail in the accompanying document *ErrLib.pdf*.

Success is signalled by returning zero; any other number is an error code (Bit flags are used to signal several errors). The bit flags can be deduced from the above code, 1 for failure to delete TWOBUFFER, 2 for failure to delete the MAGICNUMBER buffer, and so forth. Such errors should occur very infrequently.

2.4 Error messaging — the meat!

This section contains several important error functions, ERRmsg which displays a coded error, ErrDisplay which displays a string and an associated error code (and should only be used internally), ERRADD which simply appends a message to the string currently in the buffer (without displaying anything), ERRINT (similar but appends an integer in ASCII format), and last but not least, ERRSHOW which actually displays the contents of the buffer painstakingly populated by ERRADD

and ERRINT. These last three capitalised functions were created later to allow us a bit more flexibility in composing an error message out of text strings and numbers.

2.4.1 ERRmsg: display coded error

Given an error code with a value under 10000, use our error library function `ErrorString` to retrieve an error string into TWOBUFFER, and then display the string using ErrMsgAndNumber.

```
void err::ERRmsg(Int16 code)
{ if (! ERRORLIBCODE)
     { return;
     };
  if (code > 10000)
     { ERRDisplay("Gen Err",code);
       return;
     };
  Int16 el;
  el = ErrorString (ERRORLIBCODE, TWOBUFFER, TWOBUFFERSIZE, code);
  if (! el)
    { ErrMsgAndNumber("Unknown error code", code);
      return;
    };
  ErrMsgAndNumber(TWOBUFFER, el);
  return;
}
```

If the error code is over ten thousand, simply display the code with a generic message; if the code isn't represented within the error library, give a message to this effect. We fail to do anything if the error library hasn't been installed.[4]

Here's the ErrMsgAndNumber function referred to — the only other function which uses this one is the frowned-upon ERRDisplay. We might put wrappers around the PalmOS functions StrIToA and FrmCustomAlert, which we haven't done. We make use of the simple hard-coded form `DebugAlert`, the specification of which can be seen in the file *pain5.rcp* and the associated *painrcp.h*, coded below (Section 7.2.4).

```
void err::ErrMsgAndNumber(const Char * msg, Int32 nmbr)
{
  StrIToA((Char *)MAGICNUMBER, nmbr);
  FrmCustomAlert (DebugAlert, msg, (Char *)MAGICNUMBER,
                  "Message:\n");
  return;
}
```

[4] Perhaps silly; might just as well display the code with a message that the library is not present!

2.4.2 ErrDisplay and ERRSTRING: avoid us!

Useful in internal debugging, ErrDisplay gives us access to ErrMsgAndNumber. Excessive use (with associated wordy strings) results in code bloat, so avoid me!

```
void err::ERRDisplay(Char * p, Int32 c)
{
 ErrMsgAndNumber(p, c);                // see this fx
}
```

ERRSTRING is a similar nasty which displays a string of a specified length, with *no* error code. This clumsily-written routine directly invokes FrmCustomAlert, and also has the unpleasant side effect of overwriting any current string present in TWOBUFFER.

```
void err::ERRSTRING(Char * p, Int16 plen)
{ if (plen > TWOBUFFERSIZE-1)
     { plen = TWOBUFFERSIZE-1;
     };
  Char * tb = TWOBUFFER;
  while (plen > 0)
    { * tb ++ = * p ++;
      plen --;
    };
  * tb = 0x0;  // asciiz
  FrmCustomAlert (DebugAlert, TWOBUFFER, "", "");
}
```

ERRSTRING should be used sparingly if at all: the major value of this function is where we wish to display a string which is not zero-terminated.

It's worthwhile noting at this point that wherever possible (within the constraints of the PalmOS system) we specify string lengths, and avoid use of ASCIIZ strings as the means of determining a string's length. The crazy ASCIIZ convention is always a silly idea, but becomes utterly insane when UNICODE characters are used.

2.4.3 ERRADD: Append error string

Here we append a string to the current string present within TWOBUFFER, *without* displaying the result. We prevent the writing of characters under hex 0A, replacing them with question marks, as otherwise the string ultimately displayed won't behave well. We avoid overruns by testing the length (plen).

```
void err::ERRADD (Char * p, Int16 plen)
{ if (TWOTOP + plen > TWOBUFFERSIZE-1)
     { plen = TWOBUFFERSIZE-1-TWOTOP;
     };
  Char * tb = TWOBUFFER+TWOTOP;
  TWOTOP += plen;
  while (plen > 0)
    { * tb ++ = * p ++;
      if (*(tb-1) < 0x0A)
         { *(tb-1) = '?';
         };
      plen --;
    };
}
```

2.4.4 ERRINT: Append integer to error string

ERRINT is similar to ERRADD (and in fact invokes it) — we simply use the PalmOS function StrIToA (which should really be wrapped) to write the integer into the MAGICNUMBER buffer, and then add this string to TWOBUFFER. We again cheat by not wrapping StrLen, which determines the *byte* length of the resulting number.

```
void err::ERRINT (Int32 i)
{ Int16 ilen;
  Char * p = ((Char *)MAGICNUMBER); // clumsy
  StrIToA(p, i);
  ilen = StrLen(p);
  ERRADD (p, ilen);
};
```

2.4.5 ERRSHOW: Display error string

ERRSHOW simply displays the contents of TWOBUFFER, as populated by the ERRADD and ERRINT functions. There's even the option of submitting an error code, which results in use of the ErrorString library function to fetch a title message (emsg)!

```
void err::ERRSHOW(Int16 code)
{ Char * tb = TWOBUFFER+TWOTOP;
  * tb = 0x0; // asciiz
  if (! code) // if zero, don't look up code string
     { FrmCustomAlert (DebugAlert, TWOBUFFER, "", "");
     } else   // if code was provided
     { Char * emsg;
       Int16 elen;
       emsg = TWOBUFFER+TWOTOP+1;
       elen = TWOBUFFERSIZE-2-TWOTOP;
       if(! ErrorString (ERRORLIBCODE, emsg, elen, code) )
         { TWOTOP = 0; // clear!
           return;     // fail.
         };
       FrmCustomAlert(DebugAlert, TWOBUFFER, "", emsg);
     };
  TWOTOP = 0; // clear this
}
```

Submitting a code of zero simply results in display of the text in TWOBUFFER, but otherwise the title string is fetched into `emsg` (which conveniently uses the area of TWOBUFFER after the end of the current message) and displayed. Again we use FrmCustomAlert without a wrapper (naughty).

2.4.6 ErrSetConsole

A trivial routine which merely passes the handle of our console to the error library, allowing the latter to write errors to the console whenever they occur!

```
void err::ErrSetConsole(UInt16 cons)
{ ErrPassConsole (ERRORLIBCODE, cons);
}
```

2.5 PalmOS error wrappers

In most of our code, we 'wrap' calls to PalmOS within our own wrapper functions, a habit which might conceivably allow us to export our code to other systems more easily. (It also straightens out a few wrinkles, that is, differences between the way we view things and PalmOS does). Here are the simple error wrappers, which all begin with the prefix "e_". Later on you'll find that other wrapper functions contained within *wraps.hpp* all begin with "w_", which makes such invocations readily identifiable.

In all of the following functions, a return value of zero signals *failure*. This convention (which is at variance with that of some of the PalmOS functions) allows us to easily test the result of an e_wrapped function.

2.5.1 Delete: free memory on exit

```
Int16 err::e_Delete (MemPtr memP)
{ if (MemPtrUnlock (memP) != errNone)
     { return 0; // fail
     };
  if (MemPtrFree (memP) != errNone)
     { return 0;
     };
  return 1; // success
}
```

2.5.2 Close Database

Straightforward. See how we negate the result, as in our book, zero signals an error.

```
Int16 err::e_DmCloseDatabase (DmOpenRef dbP)
{ return (! DmCloseDatabase (dbP));
}
```

2.5.3 Unlock memory

Likewise:

```
Int16 err::e_MemHandleUnlock (MemHandle h)
{ return (! MemHandleUnlock (h) );
}
```

2.5.4 Release record

Zero signals an error.

```
Int16 err::e_DmReleaseRecord (DmOpenRef dbP, UInt16 index,
                              Boolean dirty)
{ return (! DmReleaseRecord (dbP, index, dirty));
}
```

2.5.5 Reserve memory

We return a pointer to a locked area of memory (or zero if we failed).

```
MemPtr err::e_New ( Int16 memsize)
{
  MemHandle memH=0;
  MemPtr memP=0;
  memH = MemHandleNew(memsize);
  if (! memH)
     { return 0; // fail.
     };
  memP = MemHandleLock(memH); // MHL1
  if (! memP)
     { return 0;
     };
  return memP;
}
```

2.5.6 Locate database

Given the ASCIIZ name of database, find it, returning its local ID (as per odd PalmOS conventions). We always submit a card value of zero, the first (UInt16) argument for DmFindDatabase.[5] A return value of zero means failure.

```
LocalID err::e_DmFindDatabase (const Char *nameP)
{ return DmFindDatabase (0, nameP);
}
```

2.5.7 Create Database

```
Int16 err::e_DmCreateDatabase (const Char *nameP)
{ return (! DmCreateDatabase (0, nameP, DBCREATOR, DBTYPE, false));
}
```

[5] Arguably a bit silly, could be modified by those wishing to use other cards.

Straightforward, once you realise that we again force the card number to zero, as for e_DmFindDatabase, and that the constants DBCREATOR and DBTYPE are specified in the header file *palmsql3A.h*.[6] The values are 'JovS' and 'DATA' respectively. The zero final parameter for DmCreateDatabase tells us that we are *not* creating a resource database. A return value of zero signals success.

2.5.8 Open Database

Given a local ID (See e_DmFindDatabase above) return a reference to the opened database, or zero on failure.

```
DmOpenRef err::e_DmOpenDatabase (LocalID dbID, UInt16 mode)
{
  return DmOpenDatabase (0, dbID, mode);
}
```

2.5.9 Create new record

Given an open database, a *pointer to* the index of the new record, and its size, create a new record, returning a handle to this record (or zero on failure).

```
MemHandle err::e_DmNewRecord (DmOpenRef dbP, UInt16 *atP, UInt32 size)
{
  return DmNewRecord (dbP, atP, size);
}
```

2.5.10 Get record

Return the handle to a record, given an open reference to the database, as well as the known index of the record.

```
MemHandle err::e_DmGetRecord (DmOpenRef dbP, UInt16 index)
{
  return DmGetRecord (dbP, index);
}
```

2.5.11 Write to a database record

Given a pointer to the record, the offset within the record, a source string and its length, write these data to the record. As PalmOS signals success with zero (errNone), we invert this, so zero signals failure on return from our wrapper.

[6] See Section 7.2.3

```
Int16 err::e_DmWrite (void *recordP, UInt32 offset, Char *srcP,
                      UInt32 bytes)
{ return (! DmWrite (recordP, offset, srcP, bytes));
}
```

2.5.12 Unload a library

Given the code of a library already loaded, unload it, returning zero from our wrapper only if this unloading failed.

```
Int16 err::LibraryUnload (UInt16 libcode)
{
  return (! SysLibRemove(libcode));
}
```

3 Wrappers for PalmOS functions

The following section is, I am afraid, excruciatingly boring. This is because it mostly provides wrappers for a fairly large number of PalmOS system functions. As explained in the preceding section (2.5), there are good reasons for the wrapping, but the existence of reasons doesn't necessarily alleviate the pain.

Some of the functions are similar to, or duplicates of the wrapped error functions above, so it would be possible to save a few bytes here and there by making the error functions public and referring to them directly. This approach would be ugly and probably even unwise.

3.1 Included files and class definition

```
#include <PalmOS.h>
#include "err.hpp"
#include "err/ERRDEBUG.h"
```

ERRDEBUG.h is included because it contains relevant error constants. All wrapped functions begin with the prefix "`w_`", and otherwise should be identical in name to the corresponding PalmOS function.[7] As with the error section, all wrapped functions return zero if and only if they *fail*. Some of the PalmOS functions return Err (or nothing), but most of our wrappers return Int16. Exceptions to the last rule are where a specific data type must be returned. Here's the complete list, within the class definition. As before, the C++ constructor and destructor are unused.

You can see that, as promised, `wraps` inherits properties from the `err` class, which is of course why we had to include *err.hpp* above.

```
class wraps : public err {
public:
          wraps                 () { }
          ~wraps                () { }
MemHandle w_MemHandleNew        (UInt32 size);
Int16     w_MemHandleFree       (MemHandle h);
MemPtr    w_MemHandleLock       (MemHandle h);
Int16     w_MemHandleUnlock     (MemHandle h);
Int16     w_MemPtrFree          (MemPtr p);
Int16     w_MemPtrUnlock        (MemPtr p);

WChar     w_TxtGetChar          (const Char* inText, UInt32 inOffset);
UInt16    w_TxtSetNextChar      (Char* ioText, UInt32 inOffset,
                                 WChar inChar);
```

[7] There is one exception. See Section 3.7.7

```
UInt16    w_TxtCharSize        (WChar inChar);

MemHandle w_DmGetRecord        (DmOpenRef dbP, UInt16 index);
Int16     w_DmGetLastErr       ();
Int16     w_DmReleaseRecord    (DmOpenRef dbP, UInt16 index,
                                Boolean dirty);
LocalID   w_DmFindDatabase     (Char *nameP, Int16 nlen,
                                Char * CRUTCH);
DmOpenRef w_DmOpenDatabase     (LocalID dbID, UInt16 mode);
Int16     w_DmCloseDatabase    (DmOpenRef dbP);
MemHandle w_DmNewRecord        (DmOpenRef dbP, UInt16 *atP,
                                UInt32 size);
Int16     w_DmCreateDatabase   (Char *nameP, Int16 namlen,
                                Char * CRUTCH);
Int16     w_DmDeleteDatabase   (LocalID dbID);
Int16     w_DmWrite            (void *recordP, UInt32 offset,
                                const void *srcP, UInt32 bytes);
UInt16    w_DmFindSortPosition (DmOpenRef dbP, void *newRecord,
                                SortRecordInfoPtr newRecordInfo,
                                DmComparF *compar, Int16 other);
MemHandle w_DmQueryRecord      (DmOpenRef dbP, UInt16 index);
Int16     w_DmRemoveRecord     (DmOpenRef dbP, UInt16 index);
MemHandle w_DmResizeRecord     (DmOpenRef dbP, UInt16 index,
                                UInt32 newSize);
Int16     w_DmDatabaseSize     (LocalID dbID, UInt32 *numRecordsP,
                          UInt32 *totalBytesP, UInt32 *dataBytesP);

UInt16    w_StrLen             (const Char *src);
Char *    w_StrCopy            (Char *dst, const Char *src);
Char *    w_StrCat             (Char *dst, const Char *src);
Int32     w_StrAToI            (Char * src);
Int16     w_StrIToA            (Char *s, Int16 slen, Int32 i);

ControlType * w_CtlNewControl  (void **formPP, UInt16 ID,
                        ControlStyleType style, const Char *textP,
                        Coord x, Coord y, Coord width,
                        Coord height, FontID font,
                        UInt8 group, Boolean leftAnchor);
void          w_CtlSetLabel    (ControlType * id, const Char * newtxt);
void          w_CtlSetValue    (ControlType *controlP,Int16 newValue);

void          w_FrmReturnToForm(UInt16 formId);
FormType *    w_FrmNewForm     (UInt16 formID, const Char *titleStrP,
                        Coord x, Coord y, Coord w, Coord h,
                        Boolean modal, UInt16 defaultButton,
                        UInt16 helpRscID, UInt16 menuRscID);
void          w_FrmDeleteForm  (FormType *formP);
void          w_FrmEraseForm   (FormType *formP);
```

```
UInt16        w_FrmDoDialog    (FormType *formP);
FormType *    w_FrmGetActiveForm(void);
UInt16        w_FrmGetActiveFormId(void);  // clumsy
void          w_FrmDrawForm    (FormType *formP);
void          w_FrmGotoForm    (UInt16 formId);
FormLabelType* w_FrmNewLabel   (FormType **formPP, UInt16 ID,
                                const Char *textP,
                                Coord x, Coord y, FontID font);
Int16        w_FrmGetObjectIndex(const FormType *formP,
                                UInt16 objID);
void *        w_FrmGetObjectPtr(const FormType *formP,
                                UInt16 objIndex);
UInt16        w_FrmGetNumberOfObjects (const FormType *formP);
UInt16        w_FrmGetFormId   (const FormType *formP);
void          w_FrmPopupForm   (UInt16 formId);
Int16         w_FrmRemoveObject(FormType **formPP, Int16 objIndex);
void          w_FrmShowObject  (FormType *formP, UInt16 objIndex);
void          w_FrmSetTitle    (FormType *formP, Char * newtitl);
Char *        w_FrmGetTitle    (FormType *formP);

FieldType *   w_FldNewField    (void **formPP, UInt16 id,
     Coord x, Coord y, Coord width, Coord height, FontID font,
     UInt32 maxChars, Boolean editable, Boolean underlined,
     Boolean singleLine, Boolean dynamicSize,
     JustificationType justification, Boolean autoShift,
     Boolean hasScrollBar, Boolean numeric);
MemHandle     w_FldGetTextHandle(const FieldType *fldP);

Int16         w_LstNewList     (void **formPP, UInt16 id, Coord x,
     Coord y, Coord width, Coord height, FontID font,
     Int16 visibleItems, Int16 triggerId);
void          w_LstSetListChoices(ListType *listP,
              Char **itemsText, Int16 numItems);
void          w_LstSetDrawFunction(ListType *listP,
              ListDrawDataFuncPtr func);

MemHandle     w_SysFormPointerArrayToStrings (Char *c,
              Int16 stringCount);
void          w_FlpBufferAToF  (FlpDouble *result, const Char *s);

Int16         xCopy (Char * dest, Char * xsrc, Int16 maxlen);
};
```

We have done some wicked things. For example, w_DmFindDatabase allows us to submit a file name and length, rather than the clumsy use of ASCIIZ.[8] The

[8] But we in turn have to introduce our own clumsiness to get around this! Clumsy design begets clumsy design.

final function (xCopy) is perhaps the only interesting function in the whole list, as it is not a wrapped representation of a PalmOS function — we use it to relieve ourselves of ASCIIZ dependency.

The following pages discuss the wrapped functions, grouped by their PalmOS functionality.

3.2 Memory wrappers

3.2.1 Requesting memory

First make sure that we're not requesting no memory (zero length), then return a handle to the allocated memory. See how we use ERRmsg to generate an error message if a rubbish value is submitted.

```
MemHandle wraps::w_MemHandleNew (UInt32 size)
{ if (size == 0)
   { ERRmsg(ErNullMemoryRequest);
     return 0;
   };
  return MemHandleNew (size);
}
```

3.2.2 Freeing memory

Free a valid handle, returning zero on failure.

```
Int16 wraps::w_MemHandleFree( MemHandle h)
{ return ( ! MemHandleFree (h) );
}
```

3.2.3 Locking memory

Lock a memory handle.

```
MemPtr wraps::w_MemHandleLock (MemHandle h)
{ if (h == 0)
   { ERRmsg(ErLockNullHandle);
     return 0;
   };
  return MemHandleLock (h);
}
```

Okay, in this and subsequent code, I'll often use ugly code such as `h == 0` when `!h` might suffice. My idiosyncrasy, which shouldn't affect the final, compiled code.

3.2.4 Unlocking memory

Given a valid handle (zero isn't valid) unlock the associated memory.

```
Int16 wraps::w_MemHandleUnlock (MemHandle h)
{ Int16 ok;
  if (h == 0)
   { ERRmsg(ErUnlockNullHandle);
     return 0;
   };
  ok = MemHandleUnlock (h);
  if (!ok) { return 1; };
  ERRmsg(ErFailUnlockNulHandle);
  return 0;
}
```

3.2.5 Freeing a pointer

Free a chunk of memory, given the memory pointer.

```
Int16 wraps::w_MemPtrFree (MemPtr p)
{ Int16 ok;
  if (p == 0)
   { ERRmsg(ErNoFreeNulPtr);
     return 0;
   };
  ok = MemPtrFree (p);
  if (!ok) { return 1; };
  ERRmsg(ErFailFreeMemPtr);
  return 0;
}
```

3.2.6 Unlocking a pointer

Unlock (but don't free) a locked memory chunk. Clumsy coding.

```
Int16 wraps::w_MemPtrUnlock (MemPtr p)
{Int16 ok;
  if (p == 0)
   { ERRmsg(ErCannotUnlockNulPtr);
     return 0;
   };
  ok = MemPtrUnlock (p);
  if (!ok) { return 1; };
  ERRmsg(ErFailUnlockMemPtr);
  return 0;
}
```

3.3 Text wrappers

Text handling is relatively sluggish and ugly, largely because of the use by PalmOS of ASCIIZ strings with UNICODE representation of characters.

3.3.1 Getting a text character

Get a character from a given offset, made clumsy due to UNICODE.

```
WChar wraps::w_TxtGetChar (const Char* inText, UInt32 inOffset)
{
  return TxtGetChar (inText, inOffset);
}
```

3.3.2 Writing a text character

Similarly, write a wide (UNICODE) character to an offset. Eugh.

```
UInt16 wraps::w_TxtSetNextChar (Char* ioText, UInt32 inOffset,
                                WChar inChar)
{
  return TxtSetNextChar (ioText, inOffset, inChar);
}
```

3.3.3 Finding the size of a character

Self-explanatory, once one realises that the character size is in bytes.[9]

```
UInt16 wraps::w_TxtCharSize (WChar inChar)
{
  return TxtCharSize (inChar);
}
```

3.4 Database wrappers

Palm ‘file’ (aka database) management is cumbersome. There is a fair number of different ways of looking at more-or-less the same thing, with confusion and a proliferation of functions.

[9] Biig problem if this can ever return a value of zero. Can it? See our usage!

3.4.1 Finding a database record

Get a handle on a record, given a handle on the opened database, and the index of the record.

```
MemHandle wraps::w_DmGetRecord (DmOpenRef dbP, UInt16 index)
{
  return DmGetRecord (dbP, index);
}
```

A useful routine (that we might as well put here) is the PalmOS hack to determine an error code, where a routine (like DmGetRecord) returns NULL on error:

```
Int16 wraps::w_DmGetLastErr ()
{
  return (Int16) DmGetLastErr ();
}
```

The value returned from the above routine is only meaningful if a database routine returned NULL as a signal that it failed.

3.4.2 Releasing a database record

Release a record, given the database and the index of the record. Flag the record as dirty or not, depending on the Boolean value in `dirty`.

```
Int16 wraps::w_DmReleaseRecord (DmOpenRef dbP, UInt16 index,
                                Boolean dirty)
{ Int16 ok;
  if (dbP == 0)
   { ERRmsg(ErCantReleaseNulRec);
     return 0;
   };
  ok = DmReleaseRecord (dbP, index, dirty);
  if (!ok) { return 1; };
  ERRmsg(ErFailReleaseRec);
  return 0;
}
```

3.4.3 Finding a named database

Locate a database. In a desperate attempt to get rid of ASCIIZ, we submit the length of the name, but this forces us to either retain a local buffer in which we store the temporary ASCIIZ string required by PalmOS, or to initialise and later release an internal buffer string [we should probably do the lattter, and fix up our function!! FIX ME]

```
// ---------FIX ME!-------------------------------------------------------------
LocalID wraps::w_DmFindDatabase (Char *nameP, Int16 namlen,
                                 Char * CRUTCH)
{ // UInt16 cardNo is always zero, in our current schema.
  // LocalID ok;
  xCopy (CRUTCH, nameP, namlen);
  *(CRUTCH+namlen) = 0x0;
  return( DmFindDatabase(MAINCARD, CRUTCH));
}
```

As things stand, we copy and zero-terminate the string, and then pass this onto DmFindDatabase. Eugh.

3.4.4 Opening a database

Given the local ID of the database, open it. There are several possible modes, notably read, write or both, but also options for preventing access by others, and leaving the database open after the program quits (!). We don't use the funny options.

```
DmOpenRef wraps::w_DmOpenDatabase (LocalID dbID, UInt16 mode)
{
  return DmOpenDatabase (MAINCARD, dbID, mode);
}
```

3.4.5 Closing a database

Self-explanatory, clumsily written.

```
Int16 wraps::w_DmCloseDatabase (DmOpenRef dbP)
{ Int16 ok;
  if (dbP == 0)
   { ERRmsg(ErCloseNulDb);
     return 0;
   };
  ok = DmCloseDatabase (dbP);
  if (!ok) { return 1; };
  ERRmsg(ErFailCloseDb);
  return 0;
}
```

3.4.6 Create a new record

Similar to the previous error version (Section 2.5.8).

```
MemHandle wraps::w_DmNewRecord (DmOpenRef dbP, UInt16 *atP,
                                UInt32 size)
{
  return DmNewRecord (dbP, atP, size);
}
```

3.4.7 Create a new database

In an analogous fashion to the way we find a database (Section 3.4.3), we create a new database given a name and length. We should fix the CRUTCH requirement. We don't make resource databases.

```
Int16 wraps::w_DmCreateDatabase (Char *nameP, Int16 namlen,
                                 Char* CRUTCH)
{ if (nameP == 0)
   { ERRmsg(ErNulDbName);
     return 0;
   };
  Int16 fail;
  xCopy (CRUTCH, nameP, namlen); // copy string
  *(CRUTCH+namlen) = 0x0;        // asciiz
  fail = DmCreateDatabase (MAINCARD, CRUTCH, DBCREATOR,
                           DBTYPE, false);
  if (!fail) { return 1; };      // success
  ERRmsg(ErFailMakeDb);
  return 0;
}
```

3.4.8 Delete a database

The following database deletion function is provided for completeness' sake. MAINCARD is defined as zero in the header file *palmsql3A.h* (Section 7.2.3).

```
Int16 wraps::w_DmDeleteDatabase (LocalID dbID)
{ Int16 ok;
  if (dbID == 0)
   { ERRmsg(ErDelNulDb);
     return 0;
   };
  ok = DmDeleteDatabase (MAINCARD, dbID);
  if (!ok) { return 1; };
  ERRmsg(ErDelDbFail);
  return 0;
}
```

3.4.9 Write to a record

Write data from a string to a record within a database at the specified offset in the record. We have several sanity checks. We permit writing of zero bytes (important) in which case we simply 'succeed'!

```
Int16 wraps::w_DmWrite (void *recordP, UInt32 offset,
                        const void *srcP, UInt32 bytes)
{ Int16 ok;
  if (recordP == 0)
   { ERRmsg(ErWriteNulRec);
     return 0;
   };
  if (srcP == 0)
   { ERRmsg(ErWriteNulSrc);
     return 0;
   };
  if (bytes == 0)
   { return 1; // permit 0 byte write!
   };
  ok = DmWriteCheck(recordP, offset, bytes);
  if (ok != errNone)
     { ERRmsg(ErFailRecCheck);
       return 0; // fail
     };
  ok = DmWrite (recordP, offset, srcP, bytes);
  if (!ok) { return 1; };
  ERRmsg(ErFailRecWrite);
  return 0;
}
```

I'm not certain whether the DmWriteCheck is a good idea, and suspect that it may slow things down a fair bit [CHECK ME]! It may however prevent major stuffups. Possible consequences of this check are dmErrNotValidRecord or dmErrWriteOutOfBounds.

3.4.10 Search for a sorted record

Given the handle on an open, sorted database and a potential record to find/insert, apply a comparison function repeatedly in order to locate the correct insertion position for the 'new' record. This PalmOS functionality is a little cumbersome. We add a few simple sanity checks.

```
UInt16 wraps::w_DmFindSortPosition (DmOpenRef dbP, void *newRecord,
                            SortRecordInfoPtr newRecordInfo,
                            DmComparF *compar, Int16 other)
```

```
{ if (dbP == 0)
   { ERRmsg(ErSortNulDb);
     return 0;
   };
  if (newRecord == 0)
   { ERRmsg(ErSortNoRec);
     return 0;
   };
  if (compar == 0)
   { ERRmsg(ErSortNoFx);
     return 0;
   };
  return DmFindSortPosition (dbP, newRecord, newRecordInfo,
                             compar, other);
}
```

3.4.11 Open a record for reading only

Given a database and the index of a record within that database, open the record (read only). Don’t set its PalmOS ‘busy bit’.

```
MemHandle wraps::w_DmQueryRecord (DmOpenRef dbP, UInt16 index)
{
  return DmQueryRecord (dbP, index);
}
```

3.4.12 Remove a record

We don’t yet use this deletion function.

```
Int16 wraps::w_DmRemoveRecord (DmOpenRef dbP, UInt16 index)
{ Int16 ok;
  if (dbP == 0)
   { ERRmsg(ErNotRemoveNulRec);
     return 0;
   };
  ok = DmRemoveRecord (dbP, index);
  if (!ok) { return 1; };
  ERRmsg(ErFailRecRemove);
  return 0;
}
```

3.4.13 Resize a record

Given the index of a record, resize it.

```
MemHandle wraps::w_DmResizeRecord (DmOpenRef dbP, UInt16 index,
                                   UInt32 newSize)
{
  return DmResizeRecord (dbP, index, newSize);
}
```

3.4.14 Determine the size of a database

This clumsy function accepts pointers to various data dwords. The PalmOs function returns zero (ErrNone) on success; we negate this to return zero on *failure*.

```
Int16 wraps::w_DmDatabaseSize (LocalID dbID, UInt32 *numRecordsP,
                       UInt32 *totalBytesP, UInt32 *dataBytesP)
{
 return (! DmDatabaseSize (MAINCARD, dbID, numRecordsP,
                           totalBytesP, dataBytesP));
   // for now, always use card 0.
   // should invert sense [fix me!]
}
```

3.5 String wrappers

These tedious string functions are more trouble than they're worth.

3.5.1 Find the length of a string

Another aspect of the hideousness of ASCIIZ. For PalmOS version between 3.5 and five, an Int16 was returned, but otherwise an unsigned integer is returned.

```
UInt16 wraps::w_StrLen (const Char *src)
{  return StrLen (src);
}
```

3.5.2 Copy a string

Copy an ASCIIZ string from source to destination.

```
Char * wraps::w_StrCopy (Char *dst, const Char *src)
{
  return StrCopy (dst, src);
}
```

3.5.3 Concatenate two ASCIIZ strings

Note the requirement for ASCIIZ.

```
Char * wraps::w_StrCat (Char *dst, const Char *src)
{
  return StrCat (dst, src);
}
```

3.5.4 Convert ASCII to an Integer

Generate a signed 32-bit integer.

```
Int32  wraps::w_StrAToI (Char * src)
{
  return StrAToI (src);
}
```

3.5.5 Convert integer to ASCII

The converse of the preceding function. Make sure there is enough space for the result (sign + 9 digits + ASCIIZ zero terminator)? The PalmOS function can fail but we cannot know this. We return the length of the string, or zero if *we* detected 'failure'.

```
Int16 wraps::w_StrIToA (Char *s, Int16 slen, Int32 i)
{  if (slen < 11)
      { return 0;
      };
   StrIToA (s, i);   // can this fail?
   return ((Int16) StrLen(s));
}
```

3.6 Control wrappers

Controls appear within PalmOS forms.

3.6.1 Create a new control

Create a control, in all of its bizarre complexity.

```
ControlType * wraps::w_CtlNewControl (void **formPP, UInt16 ID,
                  ControlStyleType style, const Char *textP,
                  Coord x, Coord y, Coord w, Coord h,
                  FontID font, UInt8 group, Boolean leftAnchor)
```

```
{
  return CtlNewControl ( formPP, ID, style, textP, x, y, w, h,
                         font, group, leftAnchor);
}
```

There is a variety of possible control styles: buttonCtl, pushButtonCtl, checkboxCtl, popupTriggerCtl, as well as the (as yet unused by us) selectorTriggerCtl, repeatingButtonCtl, sliderCtl, and feedbackSliderCtl. One other thing you might consider a 'control' — the text box — is termed a *Field* with its own baroque set of associated functions.

3.6.2 Set the label of a control

This too has ASCIIZ dependency. You must know the identifier of the control. In addition certain controls (graphical and slider ones) will crash the whole application if you apply this function. Furthermore, you have to keep a reference to the new label, as it's your responsibility to free the associated memory, or you'll have a leak!

```
void wraps::w_CtlSetLabel ( ControlType * id,
                            const Char * newtxt)
{ return CtlSetLabel (id, newtxt);
}
```

3.6.3 Set the value of a control

This only works with some controls, for example you can set the value of a checkbox to on or off, and likewise for graphical controls and push buttons. When applied to sliders, it moves the slider along the requisite part of its range.

```
void wraps::w_CtlSetValue (ControlType *controlP, Int16 newValue)
{
  return CtlSetValue(controlP, newValue);
}
```

3.7 Form wrappers

3.7.1 Kill current form, reactivate another

Self-explanatory, knowing the ID of the desired form.

```
void wraps::w_FrmReturnToForm (UInt16 formId)
{
  FrmReturnToForm (formId);
}
```

3.7.2 Create a new form

Complex dynamic creation of a form. Although this is meant to work in version 3.0 and more, it's broken in some versions of PalmOS under 3.5. You'll run into trouble[10] if you're not careful with the dimensions, despite their apparent flexible range of 1–160 pixels. Nasty.

```
FormType * wraps::w_FrmNewForm (UInt16 formID, const Char *titleStrP,
                Coord x, Coord y, Coord w, Coord h,
                Boolean modal, UInt16 defaultButton,
                UInt16 helpRscID, UInt16 menuRscID)
{ return
  FrmNewForm (formID, titleStrP, x, y, w, h,
              modal, defaultButton,
              helpRscID, menuRscID);
}
```

3.7.3 Delete a form

Releases the memory occupied by the form. God help you if you invoke this before first saying FrmEraseForm!

```
void wraps::w_FrmDeleteForm (FormType *formP)
{
  FrmDeleteForm (formP);
}
```

3.7.4 Erase a form

Erase a displayed form. See FrmDeleteForm above (Section 3.7.3).

```
void  wraps::w_FrmEraseForm  (FormType *formP)
{
  FrmEraseForm(formP);
};
```

3.7.5 Display a modal dialogue

'Modal' means that the dialogue won't release control unless the user hits something. Returns the resource ID of the button which was tapped. Requires a prior FrmInitForm, and possibly an event handler.

```
UInt16 wraps::w_FrmDoDialog (FormType *formP)
{
   return FrmDoDialog (formP);
}
```

[10] At least, with debug ROMs

3.7.6 Get the current, active form

Returns the form. Enough said.

```
FormType * wraps::w_FrmGetActiveForm (void)
{
  return FrmGetActiveForm ();
}
```

3.7.7 Get the ID of the active form

Returns the ID of the form. Note that the PalmOS function ends in ID, *not* Id. Compare this with the PalmOS function GetFormId, and you too will be confused.

```
UInt16 wraps::w_FrmGetActiveFormId (void)
{
  return FrmGetActiveFormID ();
}
```

3.7.8 Draw the form

Thus:

```
void wraps::w_FrmDrawForm (FormType *formP)
{
   FrmDrawForm (formP);
}
```

3.7.9 Go to a form

Close the current form and load the specified form.

```
void wraps::w_FrmGotoForm (UInt16 formId)
{
  FrmGotoForm (formId);
}
```

3.7.10 Create a form label

Dynamically create a new label object within a form. No, a label isn't a control.[11]

[11] This distinction might make semantic sense, but to me at least, it's not sensible programming!

```
FormLabelType * wraps::w_FrmNewLabel (FormType **formPP,
                      UInt16 ID, const Char *textP,
                      Coord x, Coord y, FontID font)
{
   return FrmNewLabel (formPP, ID, textP,
                       x, y, font);
}
```

3.7.11 Get the index of an object in a form

Given the ID of an object present in a form, determine its corresponding *index*. The index can then be used to obtain a pointer to the object (I hope this makes sense to somebody apart from the PalmOS designers)!

Unfortunately the index of the first item in a form is actually zero, so we make a kludge and return -1, not zero, if things fail. Ugh.[12]

```
Int16 wraps::w_FrmGetObjectIndex (const FormType *formP,
                                  UInt16 objID)
{ UInt16 idx;
  idx = FrmGetObjectIndex (formP, objID);
  if (idx != frmInvalidObjectId)
     { return (Int16) idx;
     };
  return -1; // BUGGER
}
```

3.7.12 Get a pointer to a form object

Well, here we obtain the pointer to the object, given the index obtained using FrmGetObjectIndex (Section 3.7.11).

```
void * wraps::w_FrmGetObjectPtr (const FormType *formP,
                                 UInt16 objIndex)
{
  return FrmGetObjectPtr (formP, objIndex);
}
```

3.7.13 Count objects within a form

We submit a pointer to the form object.

```
UInt16 wraps::w_FrmGetNumberOfObjects (const FormType *formP)
{
  return FrmGetNumberOfObjects(formP);
}
```

[12] Smarter might be to return the index incremented, and then mandate that those using the value decrement it, but this too is nasty!

3.7.14 Get the ID of a form

Given the form pointer, we obtain its ID. This ID is used by FrmPopupForm; conversely, the pointer can be obtained from the ID using FrmGetFormPtr, not that we use this fact.

```
UInt16 wraps::w_FrmGetFormId (const FormType *formP)
{
  return FrmGetFormId (formP);
}
```

3.7.15 Load and open form (without closing current)

Although we initially found this functionality attractive, we've now moved away from it, as it adds little apart from complexity. We now tend to close the background form, and then just open up the whole darn thing again. Sure it's slower, but it's also surer! (But see FormActivate 4.7.2).

```
void wraps::w_FrmPopupForm (UInt16 formId)
{
  return FrmPopupForm (formId);
}
```

We still however use FrmPopupForm once in our program.

3.7.16 Remove object from form

We must know the index of the object, and point to a reference to the form. See FrmGetObjectIndex (Section 3.7.11). The form reference may be modified by PalmOS, to the discomfiture of the unwary user! Our return value of zero signifies failure.

```
Int16 wraps::w_FrmRemoveObject (FormType **formPP, Int16 objIndex)
{
  return (! FrmRemoveObject (formPP, (UInt16) objIndex) );
}
```

3.7.17 Show form object

Again, you must know the index of the object. See FrmGetObjectIndex (Section 3.7.11).

```
void wraps::w_FrmShowObject  (FormType *formP, UInt16 objIndex)
{
  return FrmShowObject  (formP, objIndex);
}
```

3.7.18 Set form title

This function uses and does not copy the character string passed. Be very scared.

```
void wraps::w_FrmSetTitle (FormType *formP, Char * newtitl)
{
  FrmSetTitle(formP, newtitl);
}
```

Here's the retrieval routine.

```
Char * wraps::w_FrmGetTitle (FormType *formP)
{
  return (Char *) FrmGetTitle (formP);
}
```

3.8 Field wrappers

3.8.1 Create a new field

A 'field' is a text box. We create one dynamically, using a host of supplied parameters, and return a handle on the field. Note that the containing form will be altered. Dimensions are in pixels. You can create a non-editable field (pass `false` in `editable`); dynamicSize resizes dynamically depending on the contents, probably something you would wish to avoid. For all the other messy options, consult the PalmOS Reference guide.

```
FieldType * wraps::w_FldNewField (void **formPP, UInt16 id,
           Coord x, Coord y, Coord width, Coord height,
           FontID font, UInt32 maxChars, Boolean editable,
           Boolean underlined, Boolean singleLine,
           Boolean dynamicSize, JustificationType justification,
           Boolean autoShift, Boolean hasScrollBar,
           Boolean numeric)
{
 return FldNewField ( formPP, id,
                      x, y, width, height,
                      font, maxChars, editable,
                      underlined, singleLine,
                      dynamicSize, justification,
                      autoShift, hasScrollBar,
                      numeric);
}
```

3.8.2 Get a handle on the text in a field

Looks simple, but is insanely complex, depending on the fiddles you have unwisely done (RTM). Do *NOT* edit the text unless you've first 'removed the handle from the field' by setting the field text handle to null (FldSetTextHandle). Once you've edited the little bugger, you have to set the text handle back to the field, and then redraw the field (FldDrawField). It hardly seems worthwhile.

```
MemHandle wraps::w_FldGetTextHandle (const FieldType *fldP)
{ if (! fldP)
     { return 0;
     };
  return FldGetTextHandle (fldP);
};
```

3.9 List wrappers

This list 'functionality' (we use the word loosely) is madly complex.

3.9.1 Create a new list

We dynamically create a new list, imperilling life and limb. The original form is modified. It is said you can create both the popup trigger and its associated list by specifying a non-zero value for `triggerId`. This is the theory. In practice, use CtlNewControl to create the popup trigger, and then specify the ID of this popup trigger in `triggerId` when creating the list. PalmOS returns ErrNone (zero) on success; we invert the value, returning zero on failure.

```
Int16 wraps::w_LstNewList (void **formPP, UInt16 id,
                     Coord x, Coord y, Coord width, Coord height,
                     FontID font, Int16 visibleItems,
                     Int16 triggerId)
{ return ! LstNewList (formPP, id,
                     x, y, width, height,
                     font, visibleItems,
                     triggerId);
}
```

3.9.2 Set the choices for a list

Theoretically you can pass any old pointer to a pointer in **itemsText. The manual says "If you use a callback routine to draw the items in your list, the itemsText pointer is simply passed to that callback routine and is not otherwise used by the List Manager code." This is in fact now a lie, and more recent versions of PalmOS

will die quite nicely if you try such a trick. Don't. Be a good boy/girl and use SysFormPointerArrayToStrings (Section 3.10.1) to create a pointer array to some nice, appropriate ASCIIZ strings. Apply MemHandleLock to this array pointer, and finally cast the result as (Char **), supplying this to LstSetListChoices. Easy, innit? Oh yes, you should retain a pointer to the array of list items as the system won't automatically free up the memory used, and you'll be left with a memory leak. A linked list is good for such garbage collection. See the usage in our ugly FormNewControl (Section 4.7).

```
void wraps::w_LstSetListChoices (ListType *listP,
                 Char **itemsText, Int16 numItems)
{  LstSetListChoices (listP, itemsText, numItems);
}
```

3.9.3 Set the drawing function for a list

Set up the callback function which actually draws the list. Because the callback must be static, we must define it in the main section and pass its handle down the food chain to where it's needed!

```
void wraps::w_LstSetDrawFunction (ListType *listP,
                                  ListDrawDataFuncPtr func)
{  LstSetDrawFunction (listP, func);
}
```

3.10 System and other wrappers

Various ugly but useful functions.

3.10.1 Create an array of pointers to some ASCIIZ strings

Given an array of contiguous ASCIIZ strings, create a pointer array to them. Used in list display. Nasty.

```
MemHandle wraps::w_SysFormPointerArrayToStrings (Char *c,
                                                 Int16 stringCount)
{
  return (MemHandle) SysFormPointerArrayToStrings (c,stringCount);
}
```

3.10.2 Clumsily turn ASCIIZ into floating point double

```
void wraps::w_FlpBufferAToF (FlpDouble *result, const Char *s)
{
  FlpBufferAToF(result, s);
}
```

The PalmOS routine is mildly crippled, and doesn't appear to return a success/failure value. We really should make use of Rick Huebner's library someday, and generally fix up the floating point stuff. We should then return an Int16 signalling success/failure.

Because of a design 'feature' in PalmOS, the PalmOS FlpAToF function (which returns a floating point double) is incompatible with GCC, and fails miserably. Anyway, it's rather silly to return floating point doubles on the stack under most circumstances, so it would be wise to make use of FlpBufferAToF, even without the crippling incompatibility which exists!

3.11 xCopy — a useful function

We need to be able to copy a string of a specified length (not just silly old ASCIIZ) from a source to a destination buffer. Here it is, with a few sanity checks too:

```
Int16 wraps::xCopy (Char * dest, Char * xsrc, Int16 maxlen)
{ Int16 cnt;
  if (! dest)
     { ERRmsg(ErNulCopyDestin);
       return 0;
     };
  if (! xsrc)
     { ERRmsg(ErNulCopySrc);
       return 0;
     };
  cnt = maxlen;
  while (cnt > 0) // copying NO bytes is OK too.
    { * dest++ = *xsrc++;
      cnt --;
    };
  return 1;
}
```

This routine will still crash horribly if there's a buffer overrun (maxlen is incorrectly specified), but in the context of C/C++ extensive redesign is required to prevent such problems. Not really worth it, short of writing your own low-level language. We previously had this function in *utility.hpp*, but need it here for the non-ASCIIZ name handling above.

4 Fundamental routines

In this section we describe routines contained in the `utility` class, contained in the file *utility.hpp*. Of necessity, a large number of routines previously in this section have been moved out to libraries.

4.1 Utility includes

The `utility` class inherits properties from the `wraps` class, and thence the `err` class. Because the routines contained in this class invoke a variety of library routines, we must reference these. The libraries include ERRDEBUG, SCRIPTING, NUMERIC and SQL3. We've recently added references to the developmental CONSOLE library as well.

```
#include <PalmOS.h>
#include "wraps.hpp"
#include "err/ERRDEBUG.h"
#include "scripting/SCRIPTING.h"
#include "numeric/NUMERIC.h"
#include "console/CONSOLE.h"
#include "sql3/SQL3.h"

#define MAXFIELDSNAPSHOT 256
#define MAXCRUTCH 64
```

SCRIPTING.h, which you'll find in the scripting library, contains vital communication codes like iSKIP and iRETURN. We also include information about the clumsy CRUTCH buffer, and the maximum length of the even uglier SNAPSHOT buffer, used to test whether a recently used field has been altered. We currently limit the size of a text entry to 256 characters, the value specified in MAXFIELDSNAPSHOT.

The Debug_... values are used only as mask parameters passed to the various debugging routines. *Mask parameters* are bit flags which are used to decide whether a particular value is 'printed' to the debugging console.

4.2 Utility class definition

The class contains a large number of general-purpose functions. As usual, the constructor and destructor are replaced by custom functions, SetupUtility and KillUtility.

```
class utility : public wraps {
public:
```

```
        utility  () { };
        ~utility () { };
Int16   SetupUtility(ListDrawDataFuncPtr LD,
                     DmComparF * j, UInt16 scriptlibcode,
                     UInt16 numericcode, UInt16 sql3code,
                     UInt16 consolecode, UInt16 elibcode,
                     UInt16 idxcode, UInt16 cachecode);
Int16   MirrorDebug(Int16 dbg);
Int16   KillUtility(void);
Int16   UnloadAllLibraries(Int16 cumerr); // 20080510

Char *  xNew ( Int16 memsize, Int16 e);
Int16   Delete (MemPtr memP);
Int16   xSame (Char * p0, Char * p1, Int16 slen);
Int16   xCompare (Char * p0, Char * p1, Int16 slen);
Int16   xFill (Char * p0, Int16 slen, Char c);
Int16   Advance (Char * myptr, Int16 limit, WChar target);
Int32   ReadInt32 (Char * myptr);
Int16   WriteInt32 (Char * myptr, Int32 datum);
Int16   Shorter32 (Char * srcp, Int16 lgth);
LocalID u_DmFindDatabase (Char *nameP, Int16 nlen);
Int16   PalmFileCreate (Char *nameP, Int16 namlen);

FormPtr FormMakeDynamic (Char * fTitl,
        Int16 x, Int16 y, Int16 w, Int16 h);
Int16   FormActivate (FormPtr frm);
Int16   InsertWidget (FormPtr * frmP, Int16 widgetType,
        Int16 x, Int16 y, Int16 w, Int16 h,
        Char * wtext, Int16 grp, ActualList * aux,
        Int16 ilines, Int16 xtracode,
        Int16 able);
Int16   DidLastFieldChange();
Char *  GetFieldText(Int16 * fldlen);
Int16   SetActiveField (FieldType * fld, Int16 fldid);
Int16   GetFieldID ();
DmOpenRef PalmFileOpen (Char *databaseName, Int16 nlen);
Int16   PalmFileClose (DmOpenRef myDBref);

Int16   SQLselect (Char * selectstring, Int16 selen, Int16 onlyone);
Int16   SQLinsert (Char * srcp, Int16 slen);
Int16   DataRowInsert (DmOpenRef pdb, char * dat, Int16 dlen);
Int16   SQLupdate (Char * stmt, Int16 slen);
// Int16   SQLcreateTable (Char * srcp, Int16 slen);
Int16   MakeEmptyRecord (DmOpenRef myDB, UInt16 idx,
        Int16 datalen);
Int16   MakeDiskBuffer (Char * fname, Int16 fnlen,
        Int16 reccount, Int16 recsize);
Int16   PalmFileFind (Char * databaseName, Int16 namlen);
```

```
Int16   OpenStack ();
Int16   CloseStack();
Int16   ReconstituteRecord(DmOpenRef dbref, Int16 recnum); // v0.95

void * NewListNode (Int16 id, ActualList * alist);
void * FindListNode (Int16 id);
void * FindListByPopper (Int16 id);
Int16   InsertListNode (Int16 id, Char * txtlist,
        MemHandle keepme, Int16 choices,
        ActualList * alist, Int16 popper);
Int16   KillListNodes ();
Int16   LaunchConsole ();

void    WriteConsoleText (Int16 dbg, Char * txt, Int16 txlen);
void    WriteConsoleInteger (Int16 dbg, Int32 i);
void    WriteConsoleAsciiz (Int16 dbg, Char * txt);
void    WriteConsoleFloat (Int16 dbg, Char * d);
void    WriteConsoleErr (Int16 dbg, Char * d, Int32 i);
// We might make all of the above console writes contingent upon
// debug flag status!

Int16   ConsoleDump(Int16 dbg, Int16 icount);
Int16   DumpItem(Int16 dbg, Char * ISRC, Int16 icount,
                        Boolean recur);
Int16   UnFloat (Char * dest, Int16 dlen, Char * srcp);
Char *    FetchListItem( Int16 popid, Int16 idx, Int16 * ilen);

// The following is only to allow clickable labels [ugh]!
Char * InsertTextNode (Int16 code, Char * p, Int16 len); // modelled on InsertList
Int16 KillAllTextNodes();

// ----------------------------------------------------:
// The following should NOT be public but are made so to help out in sql.hpp.
// (clumsy, fix me!)
  Char *    STACK;
  Char *    STACKSTRING;
  DmOpenRef pdbSTACK;
  UInt16    ELIBCODE;
  UInt16    SCRIPTLIBCODE;
  UInt16    NUMERICCODE;
  UInt16    SQL3CODE;
  UInt16    CONSOLECODE;
  UInt16    IDXCODE;
  UInt16    CACHECODE;
  Int16    TOGGLED;
```

TOGGLED is something of a hack. If it's set, then we swop the background

and foreground colours of the widget being created. After creation of each widget, we reset TOGGLED to zero; it is only set by the TOGGLE command, which allows us to highlight buttons.

4.3 Utility: private functions and data

Here we have a whole bunch of ugly variables. DEBUGFLAGS is used in debugging, as the name suggests. We keep a record of the current text field in ACTIVEFIELD and ACTIVEFIELDID,[13] and also root our linked list of nodes (which keep tabs on actively used memory). The snapshot variables keep a record of the most recent field value, and its length (See MAXFIELDSNAPSHOT above). The `janet` callback (for search/sort) is also kept here. The NextItemID is our generator for new IDs (These start at 2000 and wrap just before reaching 10000; we need to look into this wrap around as it's a potential source of error).[14] A recent introduction (4/2007) is that the 'usual' item id generated has a value which is divisible by 8, so to generate the next item ID we add eight every time. The reason for this unusual requirement is that functions which are passed the item ID can now alter the last 3 bits, using these to signal whether a text box (field) is an ordinary text field (last 3 bits all zero), a date with the second bit set ie 01x, or a numeric, with the highest bit set i.e. 1xx.[15] For what it's worth, we also record width, height etc of the current menu.

```
private:

Int16 DEBUGFLAGS;
Char *      CRUTCH;       // uncripple ASCIIZ
FieldType * ACTIVEFIELD;  // current text field
Int16 ACTIVEFIELDID;      // corresponding id number.
Char * FIELDSNAPSHOT;     // ASCIIZ
Int16  SNAPLENGTH;        // with length.
ListLink  * ROOTLINK;     // root node
DmComparF * janet;              // callback
ListDrawDataFuncPtr LISTDRAWER; // a callback
Int16       NextItemID;   // Generator for IDs
Int16 CURRENTMENUX;       // pixel values..
Int16 CURRENTMENUY;
Int16 CURRENTMENUW;
Int16 CURRENTMENUH;
```

[13] zero if unoccupied

[14] However remote!

[15] This convenient design feature limits us to 1000 active widgets, a number we are unlikely to exceed on a PDA.

```
// for our linked text list we have:
TextNode * TEXTROOT;
  // we initialise this to null at kickoff.
```

Next, several private functions used by the above public ones. Many of these are database-related. They are all discussed in the relevant sections.

```
Int16     GenerateUniqueItemId ();
Int16     FormNewControl ( Int16 ctlcode, FormPtr * frmP,
          Int16 widgetType, Char * ctltitl,
          Int16 x, Int16 y, Int16 w, Int16 h,
          FontID fnt, Int16 grp, ActualList * aux,
          Int16 ilines,Int16 listcode, Int16 able);
Int16     CreateNewList(Int16 ctlcode, FormPtr * frmP,
           Int16 x, Int16 y, Int16 w, Int16 h,
           Char * ctltitl, FontID fnt, ActualList * aux,
           Int16 ilines);

Int16     SQLctbl (Char* destbuf, Int16 destlen,
          Char * srcp, Int16 slen);
Int16     FindNextComma( Char * datum, Int16 maxlen);
Int16     TransferFormatDatum(Char * newdata, Int16 freesize,
          Char * cDescrip, Char *listIdx, Int16 cc,
          Char *  colnames, Char * startdata);
//Int16     XFormat(Char * destin, Int16 freesize,
//          Char * datum, Int16 datlen, Char coltype,
//          Int16 colscale, Int16 maxcolsize);
Int16     SQLins (Char * srcp, Int16 slen, DmOpenRef pdb,
          Char * hdrP, Char * listIdx, Char * newrow,
          Int16 newrowlen);
};
```

4.4 Utility ‘constructor’ and destructor

These replace the functionless C++ constructor and destructor, and must be manually invoked on startup and exit. For SetupUtility usage see section 6.7, for KillUtility, section 6.7.1.

4.4.1 constructor: SetupUtility

First the constructor, which initialises several of the above variables, including the next item ID, the callback functions janet and LISTDRAWER (being passed these from above), and the linked list which keeps track of memory utilisation.

We also set up the CRUTCH buffer (used in opening files with non-ASCIIZ names), and the SNAPSHOT buffer (used to uncripple other PalmOS functionality). Finally we fill in various library references to permit scripting, sql, numeric processing and console management, and exit after setting up our stack!

```
Int16 utility::SetupUtility(ListDrawDataFuncPtr LD, DmComparF * j,
               UInt16 scriptlibcode, UInt16 numericcode,
               UInt16 sql3code, UInt16 consolecode, UInt16 elibcode,
               UInt16 idxcode, UInt16 cachecode)
{
  DEBUGFLAGS = 0;

  if (! elibcode)
     { return -10;                                  // fail.
     };
  ELIBCODE = elibcode; // early on
  if (! consolecode)
     { return -14;
     };
  CONSOLECODE = consolecode;
  ErrSetConsole(CONSOLECODE);

  IDXCODE = idxcode; // 0 = not loaded
  CACHECODE = cachecode; // likewise

  janet = j;
  ACTIVEFIELD = 0;
  NextItemID = KICKOFFITEMID;
  LISTDRAWER = LD;
  ROOTLINK = 0;
  TEXTROOT = 0; // also clear text list
  TOGGLED = 0; // no widget colour reversal (default)

  if (! scriptlibcode)
     { return -11; // fail.
     };
  if (! numericcode)
     { return -12; // fail.
     };
  if (! sql3code)
     { return -13;
     };
  SCRIPTLIBCODE = scriptlibcode;
  NUMERICCODE = numericcode;
  SQL3CODE = sql3code;

  CRUTCH = xNew(MAXCRUTCH, 2); // XNW1
  CURRENTMENUX=2;
```

```
  CURRENTMENUY=2;
  CURRENTMENUW=SCREENWIDTH-4;
  CURRENTMENUH=SCREENHEIGHT-4;
  FIELDSNAPSHOT = xNew(MAXFIELDSNAPSHOT, 0x1B); // XNW2
  SNAPLENGTH = 0;

  PassConsole(SQL3CODE, CONSOLECODE, ELIBCODE, IDXCODE, CACHECODE);
    // pass console code (etc) to sql3 lib module!!
  PassConsoleScripting(SCRIPTLIBCODE, CONSOLECODE, ELIBCODE);

  return (OpenStack());
}
```

In the above, see how we pass the handle of the console library to the SQL3 library module for later writing of debug comments to the console.

MirrorDebug is a tiny function which copies the debug flags from the caller (in *sql.hpp*) into DEBUGFLAGS, and *also* passes these debug flags to the SQL3 library, keeping the value there current.[16]

```
Int16 utility::MirrorDebug(Int16 dbg)
{
  DEBUGFLAGS = dbg;
  PassBug(SQL3CODE, DEBUGFLAGS);
  PassBugScripting(SCRIPTLIBCODE, DEBUGFLAGS);
    return 1;
}
```

4.4.2 Destructor: KillUtility

KillUtility reverses a lot of what was done by SetupUtility. Various memory buffers are deleted, libraries are unloaded, and (after a temporary debug write to the console) we exit. Note the use of the cumulative error flag passed to and returned by UnloadAllLibraries. KillUtility itself should not alter the bit flags 0–8 in cumerr.

```
Int16  utility::KillUtility(void)
{ Int16 cumerr = 0; // cumulative error

  Delete(FIELDSNAPSHOT);
  CloseStack();
  if (! KillListNodes() )
     { cumerr |= 1024;
     };
  if (! KillAllTextNodes() )
```

[16] Later we plan to do the same for all libraries!

```
     { cumerr |= 2048;
     };
  Delete (CRUTCH);
  return UnloadAllLibraries(cumerr); // 0 = success
}
```

Here's the subsidiary UnloadAllLibraries routine. It accepts, may modify, and returns the cumulative error word cumerr. UnloadAllLibraries should not alter any of the flags in cumerr apart from those in bits 0–8.

```
Int16  utility::UnloadAllLibraries(Int16 cumerr)
{
  if (!LibraryUnload(SCRIPTLIBCODE))
     { cumerr = 1;
     };
  SCRIPTLIBCODE = 0;

  if (!LibraryUnload(NUMERICCODE))
     { cumerr |= 2;
     };
  NUMERICCODE = 0;

  if (!LibraryUnload(SQL3CODE))
     { cumerr |= 4;
     };
///  if (!LibraryUnload(IDXCODE))  /// [unused at present]
///     { cumerr |= 64;
///     };

  if (!LibraryUnload(CACHECODE))
     { cumerr |= 8;
     };

  SQL3CODE = 0;
  UInt16 concnt;
  Int16 ok=0;
  concnt = 0;
  ok = CONSOLEClose (CONSOLECODE, &concnt);
  if ((ok) || (concnt))
     { // ERRDisplay("Con?", ok);
       cumerr |= 16;
     };
  if (!LibraryUnload(CONSOLECODE))
     { cumerr |= 32;
     };
  CONSOLECODE = 0;
  return cumerr; // 0 for success!
}
```

4.5 Console writing — a frill

The following permits writing a string to the console, without having to worry about CONSOLECODE, which we retain within ‘utility’.

```
void utility::WriteConsoleText(Int16 dbg, Char * txt, Int16 txlen)
{ if( (dbg == fDEBUG_ALWAYS) || (dbg & DEBUGFLAGS) )
     { ConWrite(CONSOLECODE, txt, txlen);
     };
}
```

We might consider having general purpose writes of integers, floats and so forth to the console, even perhaps a stack debugger. Let’s try a few of these

```
void utility::WriteConsoleInteger (Int16 dbg, Int32 i)
{ Int16 ilen;
   if( (dbg != fDEBUG_ALWAYS) && (! (dbg & DEBUGFLAGS)) )
      { return;
      };
   ilen = w_StrIToA (CRUTCH, maxStrIToALen+1, i);
  ConWrite (CONSOLECODE, CRUTCH, ilen);
}
```

Using the CRUTCH buffer is particularly ugly. Now let’s try this for float.

```
void utility::WriteConsoleFloat (Int16 dbg, Char * d)
{
   if( (dbg != fDEBUG_ALWAYS) && (! (dbg & DEBUGFLAGS)) )
      { return;
      };
  Int16 flen;
  flen = UnFloat (CRUTCH, MAXCRUTCH, d); // check me [??? crutch length, &d]
  ConWrite(CONSOLECODE, CRUTCH, flen);
}

// here’s a little hack to help us:
Int16 utility::UnFloat (Char * dest, Int16 dlen, Char * srcp)
{  // temporarily stolen from scripting/scripting.c; nasty!
   // might check dlen, I suppose.
   FlpDouble f;
   Int16 flen;
   xCopy ( (Char *)&f, srcp, 8);
   if ( FlpFToA (f, dest) == errNone )
      { flen = w_StrLen(dest);
        return flen;
      };
   return 0;
}
```

Next, we'll do the same for an asciiz string:

```
void utility::WriteConsoleAsciiz(Int16 dbg, Char * txt)
{
   if( (dbg != fDEBUG_ALWAYS) && (! (dbg & DEBUGFLAGS)) )
      { return;
      };
  ConWrite(CONSOLECODE, txt, w_StrLen(txt));
}
```

Similar is an 'ErrDisplay' substitute, which writes to the console:

```
void utility::WriteConsoleErr (Int16 dbg, Char * d, Int32 i)
{
  WriteConsoleAsciiz(dbg, d);
  WriteConsoleInteger(dbg, i);
  WriteConsoleAsciiz(dbg, " "); // space things!
}
```

Finally, a routine to dump the top portion of the stack. The argument `icount` is the number of items to dump.

```
Int16 utility::ConsoleDump(Int16 dbg, Int16 icount)
{  Int16 top;
   Int16 bot;
   Int16 max;
   Int16 itms;
   top = *((Int16*)(STACK+oTOP));  // stack first..
   bot = *((Int16*)(STACK+oSTART));
   max = *((Int16*)(STACK+oMAX));
   WriteConsoleAsciiz(dbg, "\x0A" "Dump: ");
   WriteConsoleInteger(dbg, bot/16);
   WriteConsoleAsciiz(dbg, ",");
   WriteConsoleInteger(dbg, max/16);
   itms = (top-bot)/16;
   WriteConsoleAsciiz(dbg, " i=");
   WriteConsoleInteger(dbg, itms);

   Int16 SStop; // stackstring next
   Int16 SSbot;
   Int16 SSmax;
   SStop = *((Int16*)(STACKSTRING+oTOP));
   SSbot = *((Int16*)(STACKSTRING+oSTART));
   SSmax = *((Int16*)(STACKSTRING+oMAX));
   WriteConsoleAsciiz(dbg, "[");
   WriteConsoleInteger(dbg, SSbot);
   WriteConsoleAsciiz(dbg, ",");
   WriteConsoleInteger(dbg, SStop);
```

```
   WriteConsoleAsciiz(dbg, ",");
   WriteConsoleInteger(dbg, SSmax);
   WriteConsoleAsciiz(dbg, "]");

   if (itms < icount) { icount = itms; };
   while (icount > 0)
     { top -= 16;  // move down to (first) item
       DumpItem(dbg, STACK+top, icount, 1);
       icount --;
     };

  return 1;
}
```

Here's how we actually write each *item* to the console:

```
Int16 utility::DumpItem(Int16 dbg, Char * ISRC, Int16 icount,
                        Boolean recur)
{ Char c;
  Int16 ilen;
  Int16 ilentrue=0;
  Int16 ioff;
//  Int16 insertions=0;
//  Int16 offxstring=0; // keep compiler happy
  Int16 showlen;
  Int16 ival;
  Char * S;
  Boolean more;

  WriteConsoleAsciiz(dbg, "\x0A" " ");
  ioff = 0;
  c = *(ISRC+15);  // get item type
  if ((c < 'A') || (c > 'X'))
     { WriteConsoleAsciiz (dbg, "[");
       WriteConsoleInteger(dbg, c);
       WriteConsoleAsciiz(dbg, "]");
       c = '?';
     };
  ilen = 0x0F & (*(ISRC+14));
  ilentrue = ilen;
  if (ilen > 14)
     { ilentrue = *((Int16 *)(ISRC+0));
       ioff = *((Int16 *)(ISRC+2));
     };
  WriteConsoleText(dbg, &c, 1);  // show type
  WriteConsoleAsciiz(dbg, "=");

/*
```

```
  if (ioff) // if eXtended
     { ERRADD("[",1);
       ERRINT(ilentrue);
       ERRADD(",",1);
       ERRINT(ioff);
       if (c == 'X')  // if type X:
          { insertions = *((Int16 *)(ISRC+4));
            offxstring = *((Int16 *)(ISRC+6));
            ERRADD(",",1);
            ERRINT(insertions);
            ERRADD(",",1);
            ERRINT(offxstring);
          };
       ERRADD("]",1);
     } else // otherwise simply show length
     { ERRINT(ilen);
     };
*/

  S = ISRC;
  if (ilen > 14)
     { S = STACKSTRING+ioff;
     };
  switch (c)
    { case 'V':
      case 'N':
      case 'D':
      case 'T':
      case 'S':
      showlen = ilentrue;
      more = 0;
      if (showlen > 15)
         { showlen = 13;
           more = 1;
         };
      WriteConsoleText(dbg, S, showlen); // display string
      if (more) { WriteConsoleAsciiz(dbg, ".."); }; // signal stuff left out
      break;

      case 'I':
      ival = *((Int32*)(S));
      // watch out but if we store std on ARM, still ok!
      WriteConsoleInteger(dbg, ival);
      break;

      default:
      WriteConsoleAsciiz(dbg, "...");
    };
```

```
/*
  if (c != 'X')
     { return 1;
     };

  if (! recur)  // one deep only
     { ERRmsg(ErDeepRecurStkShow);
       return 0;
     }; // error: X inside X!

  Char * pX;
  pX = STACKSTRING + offxstring; // first x-item
  while (insertions > 0)
    { ERRADD("->",2); // indicator at start of x-item!
      ShowStackItem(pX, insertions, 0);  // recur=0
      pX += 16;  // next x-item
      insertions --;
    };
  ERRADD("INTO:" "\x0A" ,6); // 'insertee'!
  ShowStackItem(pX, 0, 0);
*/

  return 1; // ok
}
```

4.6 General-purpose utility functions

A bunch of important utility functions, the most important of all being xNew and Delete.

4.6.1 xNew: request memory

xNew reserves and locks a memory block, returning a character pointer to the start of this memory.

In order to be able to resolve pernicious problems with debugging memory allocation/deallocation, we radically altered xNew. We added an 'e' parameter, indicating the source of the xNew. We write this to the cyclical error buffer using ErrorWrite and similarly modify Delete to pull out and record this value, because we not only store the number in the error buffer, we also set aside 2 bytes of memory to keep it in the reserved memory. We can then go through the error buffer and find undeleted items by their absence — each allocation should have a corresponding de-allocation!

Because the number of allocations becomes very large, we have recently modified this routine to *ignore* e-values of zero (not writing them using ErrorWrite).

This adjustment is also made to the memory-releasing routine Delete. It is wise to use an e-value of zero only in circumstances where memory is rapidly released with no potential for error in between xNew and Delete.

```
Char * utility::xNew ( Int16 memsize, Int16 e)
{ MemHandle memH=0; // clumsy.
  MemPtr memP=0;
  memH = w_MemHandleNew(memsize+2); // 2 more bytes!
  if (! memH)
     { ERRmsg(ErNoMemory);
       return 0;
     };
  memP = w_MemHandleLock(memH); // MHL2
  if (! memP)
     { ERRmsg(ErNoMemLock);
       return 0; //
     };
  if (e)
     { ErrorWrite(ELIBCODE, e);
     };
  *((Int16 *)memP) = e;
  return (((Char *) memP)+2);
}
```

4.6.2 Delete: free memory

In freeing memory using Delete, we reverse the above process, again pulling out the e-code, but this time flagging it using an OR with hex `8000` before using ERRmsg to write to the error buffer.

```
Int16 utility::Delete (MemPtr memP)
{ if (! memP)
     { ERRmsg(ErDelNulPtr);
       return 1; // recover (?)
     };
  Char * p;
  p = ((Char *)memP)-2;
  Int16 i;
  i = *((Int16 *)p);
  if (i)
     { i |= 0x8000;
       ErrorWrite(ELIBCODE, i);
     };
  memP = (MemPtr) (p);
  if (! w_MemPtrUnlock (memP))
     { ERRmsg(ErMemoryNoUnlock);
       return 0; // fail
```

```
     };
  if (! w_MemPtrFree (memP))
     { ERRmsg(ErMemoryNoFree);
       return 0;
     };
  return 1; // success
}
```

4.6.3 Same: compare strings

This simple comparison returns just yes (1 = identical strings), or zero. There is a potential problem here, in that we don't know the length of each string, so the parameter slen should always contain the lesser length. Rather artificial, really.

```
Int16 utility::xSame (Char * p0, Char * p1, Int16 slen)
{ while (slen > 0)
    { if (*p0 != *p1)
         { return 0; //fail
         };
      slen --;
      p0 ++;
      p1 ++;
    };
  return 1; // identical strings
}
```

4.6.4 xCompare: odd man out

This function is somewhat odd. It returns *zero* if the strings are *the same*, a gross divergence from our usual practice, and even more remarkably returns -1 if the string at p0 is less than that at p1, and +1 if the reverse is true. The coding is clumsy. It uses a case-sensitive standard ASCII collation. The same limitation of `slen` applies as was the case for xSame above.

```
Int16 utility::xCompare (Char * p0, Char * p1, Int16 slen)
{ while (slen > 0)
    { if (*p0 != *p1)
         { if (*p0 > *p1) { return 1; };
           return -1;
         };
      slen --;
      p0 ++;
      p1 ++;
    };
  return 0; // identical strings.
}
```

4.6.5 Fill string with a byte value

Given an 8-bit character c, fill the string for the requisite length with that character.[17] (There is a faster way — see the PalmOS documentation).

```
Int16 utility::xFill (Char * p0, Int16 slen, Char c)
{ while (slen > 0)
    { * p0++ = c;
      slen --;
    };
 return 1; // success
}
```

4.6.6 Advance: find position of character

This useful function scans ahead through a string, looking for a particular wide character, up to a stated limit. As Advance always returns the offset of the character immediately *after* the target character, we can return 0 on failure! We can also then identify characters at the start of the string.

```
Int16 utility::Advance (Char * myptr, Int16 limit, WChar target)
{ WChar ch;  // NOTE: limit is _signed_
  UInt16 chlen;
  Int16 howfar = 0;
  while (limit > 0)
    { ch = w_TxtGetChar(myptr, 0);
      chlen = w_TxtCharSize(ch);
      howfar += chlen;
      if (ch == target)
         { return howfar;
         };
      myptr += chlen; //bump ptr past character
      limit -= chlen;
    };
  return 0; // fail.
}
```

4.6.7 ReadInt32: endian read integer from pointer

Our convention is always to use *big-endian* numbers. The following function should thus be replaced on an ARM (little-endian) processor, unless that processor is operating in 68K emulation mode. [DANGER POINT: FIX ME!]

[17] We might do a sanity check to see that slen isn't negative, and report such a misadventure.

```
// ------------------------------------------------------------------------------
// 6. Read integers from pointer
Int32 utility::ReadInt32 (Char * myptr)
{                                // hi byte stored first
 return * ((Int32 *) myptr);   // FIX IF ARM PROCESSOR
}
```

4.6.8 WriteInt32: endian write 32 bit integer from string

A PDA problem will arise with this clumsily-written write if the destination address is not on a dword (four byte) integral boundary. It is the responsibility of the invoker to ensure this state. There's another version in the numeric library, unused at present.

ALERT: We assume a big-endian write, so an ARM processor working in native mode will need a different routine, as the ARM is little endian.

We check that we're not writing to a null pointer, before the write.

```
Int16 utility::WriteInt32 (Char * myptr, Int32 datum)
{ if (! myptr)
     { ERRmsg(ErNulInt32Write);
       return 0; // aagh. Write to null ptr!
     };
Int32 * p2;
p2 = (Int32 *) myptr;
* p2 = datum;
return 1;
}
```

4.6.9 Clumsy short integer read from text

Shorter32 reads a 16 bit integer, despite the internal mechanics. For now, we permit no negatives [FIX ME]! ReadInt32X has been moved to the numeric library; ReadInt16X is similar. This function is only used by the SQL create table functions, which have in any case been temporarily disabled (Section 5.11.2).

```
Int16 utility::Shorter32 (Char * srcp, Int16 lgth)
{
 Int32 i;
 i = Asc2Int32(NUMERICCODE, srcp, lgth);
 if (i >= 0) // for now, allow NO negatives
    { return ((Int16) i);
    };
 ERRmsg(ErBadAscIntString);
 return 0; // fail
}
```

4.6.10 Generate ‘unique’ ID

Each PalmOS menu item requires a unique ID: here we sequentially generate these. There is a little problem if the program has been generating and deleting zillions of items, as it’s remotely possible that one such item would have been generated and kept, and then we wrap around to the same item.[18]

Note that the current stored value is obtained and returned, after bumping the stored value. Because of the nature of the PDA, we (of course) don’t need to worry about multiple access to this function.

```
Int16 utility::GenerateUniqueItemId ()
{ Int16 nid = NextItemID;
  NextItemID += 8; // allow space for bit-tweaking!
  if (NextItemID >= 10000)
     { NextItemID = KICKOFFITEMID; // wrap.
     }; // hmm. check this!
  return (nid);
}
```

In generating the ID we leave the three lowest order bits clear (by starting with a number divisible by 8 and adding eight each time). These bits are used as follows:

bit 0 Used in a field ID to indicate that this isn’t actually a field at all, but a clickable ‘label’ (We implement such a label as a special field);

bit 1 Signals that the field is a date, and if clicked on, a date picker must be invoked;

bit 2 Signals a number. Invoke the number picker in a similar fashion to invocation of a date picker.

See page 191 et seq.

4.6.11 Crutch functions

We find it very desirable not to use ASCIIZ (null terminated) strings. Unfortunately PalmOS uses these. In a rather pathetic attempt to avoid ASCIIZ, we replace certain ASCIIZ dependent functions with name/length dependent ones, adding in a supplementary buffer (CRUTCH) to permit writing of the name to the buffer before PalmOS function invocation.

[18] Although the chances are remote because we’ve changed our paradigm and now don’t retain old menus in the background, deleting them when we move up.

This approach is really clumsily implemented. We should perhaps move xNew and Delete to the `wraps` class, or export the buffer to that class, rather than passing it from here! [FIX ME: have wrapper constructor, and invoke it from above, passing the requisite buffer, and then using it internally]

```
LocalID utility::u_DmFindDatabase  (Char *nameP, Int16 nlen)
{
  LocalID ok;
  ok = w_DmFindDatabase (nameP, nlen, CRUTCH);
  return ok;
};

Int16   utility::PalmFileCreate (Char *nameP, Int16 namlen)
{ Int16 ok;
  ok = w_DmCreateDatabase (nameP, namlen, CRUTCH);
  return ok;
};
```

4.7 Utility form functions

PalmOS forms are complex to manage. The FormNewControl is our main function, a bit of a misnomer from a PalmOS point of view, as we refer to all objects within forms as 'controls',[19] whereas PalmOS makes rather arbitrary distinctions between controls, text fields, and labels, and thus generates multiple rather unnecessary functions to manage components based on these distinctions.

4.7.1 Dynamically create form

We wrap the PalmOS FrmNewForm to make a new form. Our form is 'modal', so it always retains the focus. The last three parameters of FrmNewForm are zero, so there is no ID for a default action, no online help, and no menu resource number.

```
FormPtr utility::FormMakeDynamic (Char * fTitl,
        Int16 x, Int16 y, Int16 w, Int16 h)
{ FormPtr   frm;
  Int16     frmcode;
  frmcode = GenerateUniqueItemId();
  frm = w_FrmNewForm( frmcode, fTitl, x, y, w, h,
                    true,     // modal
                    0, 0, 0);
  return frm; // return the form
}
```

[19] Sometimes we even use the term 'widgets'. We're really rather inconsistent, I'm afraid!

4.7.2 Activate dynamic form

Next, we have a routine to activate the form using FrmPopupForm, which simply posts a frmLoadEvent and a frmOpenEvent, returning void. Note that we *must* create all contained objects before the frmOpenEvent is posted. This routine does not close the current form. (Which FrmGotoForm would do).

```
Int16 utility::FormActivate (FormPtr frm)
{
  UInt16    frmid;
  frmid = w_FrmGetFormId(frm);
  if (! frmid) { return 0; };  // fail
  w_FrmPopupForm(frmid);
  return 1; // success.
}
```

4.7.3 Insert a new widget

Table 1 lists the types of item we can create within any form.

Item type	*Menu Code*	*Meaning*
BUTTONCTL	2	Simple button
PUSHBUTTONCTL	4	Pushbutton (toggles state)
CHECKBOXCTL	3	Check box (tick box)
POPUPTRIGGERCTL	6	popup trigger for list
LABELNOTCTL	1	text label
FIELDNOTCTL	10	text field
DATEFIELD	12	‘text’ field (date)
NUMBERFIELD	14	‘text’ field (number)
LISTNOTCTL	(18)	list of items
MENUITSELF	(50)	used for click on list header

Table 1: Our types of menu item

We have removed the SLIDERCTL and FEEDBACKSLIDERCTL controls, at least for now.

The above codes are defined in *palmsql3A.h*. Unfortunately we use different codes within menus, contained in the second numeric column. Different PalmOS functions are required to dynamically create the various ‘controls’ — FrmNewLabel (Label), FldNewField (text field), and CtlNewControl (the rest). Note that we *never* use the PalmOS facility to automatically associate text with a check box: such text must be created and positioned as a separate text label.

At some stage we need to implement graphic images. We also haven't yet implemented the slider controls. A feedback slider control sends repeating events as the slider is moved, not just when the user releases the stylus. We might also consider implementing a selector trigger (a control around a text label which contracts/expands depending on the size of the contents), and even a repeating button (sends repeated events if the user holds the stylus on it).

The InsertWidget function is simply a wrapper for FormNewControl below. Later on we might need to modify the submitted parameters, even adding more (ugly), or keeping local values for current font, etc.

```
Int16 utility::InsertWidget (FormPtr * frmP, Int16 widgetType,
                             Int16 x, Int16 y, Int16 w, Int16 h,
                             Char * wtext, Int16 grp,
                             ActualList * aux, Int16 ilines,
                             Int16 xtracode, Int16 able)
{ Int16 icode;

  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_WIDGET, "("); //
  -OPTIONAL

  icode = GenerateUniqueItemId();
  icode = FormNewControl (icode, frmP,
                          widgetType, wtext,
                          x, y, w, h,
                          (FontID) 0, grp,
                          aux, ilines, xtracode, able);
  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_WIDGET, ")\x0A");
  -OPTIONAL
  return (icode);
}
```

InsertWidget is widely used, by CreateOneWidget, CreateOneTable, CreatePolyTable and its subsidiary MakeWholeColumn, and the more complex MakeTextForm. The most important usage is the first. The `aux` variable is of particular interest in creation of lists, and only has a non-null value when InsertWidget is invoked by either CreateOneWidget or MakeWholeColumn.

4.7.4 Create a new form 'control'

Here we use the term 'control' loosely as any form item, and not in the restrictive PalmOS sense. Note the use of a pointer to a form pointer, so that the form itself can be altered!

There are a few other wrinkles in PalmOS. Fields are enabled by setting or resetting the editable flag on creation, whereas controls are enabled using CtlSetEnabled. We submit an `xtracode` value to turn on a checkbox or pushbutton as it's created.

This complex function returns the unique control code of the created object, or zero on failure. We allow some tweaking of the unique value submitted in **ctlcode**. Ordinarily this number is always divisible by 8, leaving the last 3 bits to signal the subtype of the 'control'! A text box with all three bits reset is an ordinary text box, but if the second last bit is set (010 or 011 ie. 01x) then the control is a date, and if the highest bit is set (1xx) then the control is numeric!

Let's break the function into component parts. First, the entry section:

```
Int16 utility::FormNewControl ( Int16 ctlcode,
           FormPtr * frmP, Int16 widgetType,
           Char * ctltitl,
           Int16 x, Int16 y, Int16 w, Int16 h,
           FontID fnt, Int16 grp,
           ActualList * aux, Int16 ilines,
           Int16 xtracode, // turn on?
           Int16 able         )
{
  Int16 ctlstyle;
  FieldType * okft1;
  ControlType * ctp = 0; // default null
```

Take particular note of the `aux` variable. This contains an ASCIIZ string which is *only* used in list creation. The string contains a pipe-delimited list of items in *pairs*. The first item in each pair is a unique ID for that item, and the second is an associated text string.

Next, if we're dealing with a label, we simply create a new label using FrmNewLabel.

```
  if (widgetType == LABELNOTCTL)
     {
       +OPTIONAL
       WriteConsoleAsciiz(fDEBUG_WIDGET, "Label:"); //
       WriteConsoleAsciiz(fDEBUG_WIDGET, ctltitl);
       -OPTIONAL

     if (! ctltitl)
        { return 0;  // 2007-9-30: IGNORE IF NULL!
        };

  // In order to allow clickable labels, we have to do
  // a lot of fiddling. First thing we do is only allow
```

```
// a label to be clickable if the group (grp) is nonzero.
// Otherwise we fall back on old style [frowned on?]
// Note that many of our older label WIDTHS are rubbish,
// and we must fix this in the PainForm database (AnalgesiaDB2.tex) % [FIX ME!]

     if (! grp)  // ??
        {  w_FrmNewLabel ( (FormType **)frmP,
                    ctlcode, ctltitl, x, y, fnt);
        } else
        { if (h < 15) { h = 15; }; // [hack]
          Int16 fldlenx;
          fldlenx = w_StrLen(ctltitl);
          ctlcode |= 1; // [see note below]
          okft1 = w_FldNewField ( (void **)frmP,
                        (UInt16) ctlcode, x, y, w, h, (FontID) fnt,
                        (UInt32) 1+fldlenx, // maxChars
                        (Boolean) 0, // editable <== NB reset.
                        (Boolean) noUnderline,
                        (Boolean)1,  // singleline
                        (Boolean)0,  // dynamicSize
                        (JustificationType) leftAlign,
                        (Boolean)0,  // autoShift //
                        (Boolean)0,  // hasScrollBar
                        (Boolean)0  // numeric
                        );
          Char * p;
          p = InsertTextNode(ctlcode, ctltitl, fldlenx);
          FldSetTextPtr(okft1, p);
        };
      return ctlcode;
    }
```

In the above, we set bit zero of the control code (OR it with 1) if and only if we are creating a clickable control. This flag serves to differentiate between a clickable control and a text field which has been disabled (Otherwise we would still respond to a click on a disabled text field, sometimes with devastating results!)

The creation of a field is more complex, with the potential to submit a host of parameters, many of dubious value. The height check (less than 15 pixels) prevents an otherwise disastrous malfunction in PalmOS where a text box isn't tall enough. We also permit insertion of default text into the text field (if the value of `ctltitl` isn't null, it's assumed to contain this text as an [eugh] ASCIIZ string). The widget types DATEFIELD and NUMBERFIELD specify subtypes of text field, and we tweak the **ctlcode** value returned, signalling these types!

```
if (  (widgetType == FIELDNOTCTL)
    ||(widgetType == DATEFIELD)
```

```
 ||(widgetType == NUMBERFIELD)
)
{
  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_WIDGET, "Textbox:"); //
  -OPTIONAL
  if (h < 15) { h = 15; }; // [hack]

  if (widgetType == DATEFIELD)
        { ctlcode += 2; // set bit #1
        };
  if (widgetType == NUMBERFIELD)
        { ctlcode += 4; // set bit #2
        };
  // the above signal various field subtypes!
  // by default lowest 3 bits in ctlcode are all zero.

  okft1 = w_FldNewField ( (void **)frmP,
                (UInt16) ctlcode, x, y, w, h, (FontID) fnt,
                (UInt32) 63, // maxChars <== Aha. fix me!
                (Boolean) able, // editable
                (Boolean) grayUnderline,
                (Boolean)1,  // singleline
                (Boolean)0,  // dynamicSize
                (JustificationType) leftAlign,
                (Boolean)0,  // autoShift //
                (Boolean)0,  // hasScrollBar
                (Boolean)0  // numeric <===== hmm!
                );
if (ctltitl)
   { Int16 fldlen;
     fldlen = w_StrLen(ctltitl);
     if (fldlen > 0)
        {
          +OPTIONAL
          WriteConsoleAsciiz(fDEBUG_WIDGET, ctltitl); //
          -OPTIONAL
          if (! FldInsert (okft1, ctltitl, fldlen))
             { ERRmsg(ErFldTxt); // fail
               ERRSTRING(ctltitl, fldlen); // [??]
               // should we not return 0 [check me]?
             };
        };
   };
   return ctlcode;
};
```

At some stage it might be an idea to allow multiple lines, an intrinsic scroll

bar, and right alignment.[20]

Next, let's sort out list creation:

```
if (widgetType == LISTNOTCTL)
  {
    return CreateNewList(ctlcode, frmP, x, y, w, h,
                         ctltitl, fnt, aux, ilines);
  };
```

Okay, after working through the list-creation marathon (check it out), we are only left with true PalmOS controls, so their management is relatively straightforward. We convert from our code to the PalmOS one using our library function GetItemStyle. There's a wrinkle here, for the PalmOS code for a buttonctl is zero, so in fact GetItemStyle returns *one plus* the PalmOS code — we have to decrement the value to get the true code! In addition, after we've made the new control, if it's a pushbutton or checkbox, we must still select it if the value in `xtracode` is one. Finally, depending on the value in `able`, we may wish to disable the control before returning its control code.

```
  ctlstyle = GetItemStyle(SCRIPTLIBCODE, widgetType );
       if (! ctlstyle) // clumsy?
           { ERRmsg(ErBadControlStyle);
             ERRDisplay("Code",widgetType); //[???]
             return 0;
           };
   +OPTIONAL
     WriteConsoleAsciiz(fDEBUG_WIDGET, "Ctl:"); //
     WriteConsoleInteger(fDEBUG_WIDGET, ctlstyle);
   -OPTIONAL
   ctlstyle --; // get true code!
   ctp = w_CtlNewControl ( (void **)frmP,
         ctlcode,
         (enum controlStyles) ctlstyle,
         ctltitl,
         x, y, w, h,
         fnt,
         grp,
         0);   // 0 = don't resize.

   if (   (xtracode
           &&(  (ctlstyle == checkboxCtl)
             ||(ctlstyle == pushButtonCtl)
              )
           )
       || (TOGGLED
```

[20] The scroll bar might on the other hand be a really dreadful perpetuation of a *bad* idea!

```
           &&(ctlstyle == buttonCtl)
          )
      ){ w_CtlSetValue (ctp, 1); // default 0
       };
   // The 'TOGGLED' check is a hack to allow buttons to be dark!

   if (ctp && ! able)  // if 'disabled'
      { CtlSetEnabled(ctp, 0); // clear enabling
      };

   TOGGLED = 0; // reset
return ctlcode; // return code of new control!
}
```

In the above, we changed things (30-1-2006), so that any non-zero value for xtracode will result in the control being turned on. (Formerly the value had to be one). This allows us to send a value of say -1, which is in keeping with our former convoluted handling of button presses in Perl.[21]

4.7.5 List creation

Creation of a list is murder. There are three sub-sections: first we create the pop trigger, next, create the list and associate it with the pop trigger, and finally associate the list with its list-drawing function.

Let's look at the first component, the relatively simple creation of a pop trigger:

```
Int16 utility::CreateNewList(Int16 ctlcode, FormPtr * frmP,
                  Int16 x, Int16 y, Int16 w, Int16 h,
                  Char * ctltitl, FontID fnt, ActualList * aux,
                  Int16 ilines)
{
    +OPTIONAL
    WriteConsoleAsciiz(fDEBUG_WIDGET, "List:"); //
    -OPTIONAL

  Int16 ptcode;
  ptcode = GenerateUniqueItemId();
  w_CtlNewControl ( (void **)frmP,
        ptcode,
        popupTriggerCtl,
        ctltitl,
        x, y, w, h,
        fnt,
        0,
```

[21] We've now realised that this is silly and rewritten the Perl, so this wrinkle is irrelevant but should be harmless.

```
    true); // 0 = don't resize [hmm?]
```

The value in `ctltitl` becomes the current value of the poptrigger.

In the next section, where we make a list, you would think there was no earthly reason why if you already have an identical list you couldn't reuse it! Unfortunately, PalmOS is not of this world, and I can find no way of implementing such an economy. Each and every time, you have to re-create the list in the complex packed format required by PalmOS.

We submit our list as a text string contained within the variable `aux`, with list items separated by pipes (our convention). We must first request a new memory block called AUX, copy the list into the new block, and then replace the pipes with null terminators. Next we invoke a PalmOS function (3.10.1) to create the requisite array. We are left with two clumsy potential memory leaks (AUX and kk in the code below), which we must store away using our InsertListNode, so that later on when we delete the menu, we can also delete these leaks.

```
MemHandle kk;
Char * jj;
Int16 choices;

// hack:
// Int16 ci;
Int16 offs;
Int16 sl;
Char * P;
Char * D;

sl = aux->auxlen;
Char * AUX = xNew(sl+1, 0x1C); // wasteful XNW3
// [FIX ME:] WE REALLY SHOULD MAKE THIS EARLIER
// AND STORE PROTOTYPE WITHIN aux !!!!!!!!!!!
// ie augment ActualList to contain
// the following AUX prototype AND its
// length and number of items!

// now copy over every SECOND item to AUX
// making sure we terminate each with a hex zero (ASCIIZ)
offs = 1;
P = aux->auxlist;
D = AUX;
choices = 0;

offs = Advance (P, sl, '|');
while (offs > 0)
  { P  += offs;
    sl -= offs; // skip past first item!!
```

```
    offs = Advance (P, sl, '|');
    if (offs > 0)
       {
         choices ++;
         xCopy(D,P,offs);
         P += offs;
         D += offs;
         sl -= offs; // error was here 25-6-06
         *(D-1) = 0x0; // asciiz
         offs = Advance (P, sl, '|');
       };
  };
*D=0x0; // ensure asciiz!


kk = w_SysFormPointerArrayToStrings(AUX, choices);
if (!kk)
   { ERRmsg(ErFailSysPtrArray);
     return 0; // fail
   };
jj = (Char *) w_MemHandleLock(kk); // MHL3
if (!jj)
   { ERRmsg(ErFailLockPtrArray);
     return 0; // fail
   };
if (!InsertListNode (ctlcode, AUX, kk, choices, aux,
                     ptcode))
       { ERRmsg(ErListCreationFailed);
         return 0;
       };
```

Finally, we can create our new list, and associate it with the list-drawing function:

```
if (ilines > choices)
   { ilines = choices;
   };
w_LstNewList ( (void **)frmP, ctlcode,
               x, y, w, h*3,
               stdFont, ilines, ptcode);
UInt16 objcount;
FormPtr frm;
ListType * lstP;
frm = (FormPtr) * frmP;  // dereference!
objcount = w_FrmGetObjectIndex(frm, ctlcode);
lstP = (ListType *) w_FrmGetObjectPtr(frm, objcount);
if (! lstP)
   { ERRDisplay("+no lst",0); // [???]
```

```
       };
    w_LstSetDrawFunction(lstP, (ListDrawDataFuncPtr)LISTDRAWER);
    // attach our (formatted) list:
    LstSetListChoices(lstP, (Char **)jj, choices); // *system*
    return ptcode; // return id of poptrigger
}
```

In the above, it might be wise to check the value of objcount, which we don't do at present. The ERRDisplay command should be fixed up too. LstSetDrawFunction sets up the callback, and LstSetListChoices attaches our formatted list items. [FIX THIS: w_]. We return the ID of the poptrigger (`ptcode`) and *not* that of the list (`ctlcode`).

4.7.6 Finding a list item by index

While we're about it, let's address the problem of pulling out a list item. PalmOS supplies the index of the item selected, but we want to pull out the corresponding value from our original list of *pairs*! We do so by taking the index and advancing along through our original list, two by two!

We supply not only the index of the item, but also the poplist id. [This routine is in fact clumsy. It is actually unnecessary for us to store everything in the ActualList as we do. We only need store the IDs, as this is all we pull out. Would also save on space: FIX ME]

```
Char * utility::FetchListItem( Int16 popid, Int16 idx, Int16 * ilen)
{
  ListLink * LL;
  Char * tp;
  ActualList * alist;

  LL = (ListLink *) FindListByPopper(popid);
  if (! LL)
     { return 0; // or signal error?
     };
  alist = LL->alist;
  tp = alist->auxlist;
  Int16 alen;
  Int16 i=0;

  alen = alist->auxlen;

  // hey is this index 0-based? [yes]
  while (idx > 0)
    { i = Advance(tp, alen, '|');
      tp += i;
```

```
      alen -= i;
      idx --;
      i = Advance(tp, alen, '|');
      tp += i;
      alen -= i;
    };
  * ilen = -1+Advance(tp, alen, '|'); // brutal
  return tp;
}
```

4.8 Utility field functions

We have a real problem owing to the design of PalmOS. We cannot accurately identify when we 'leave' a text field by e.g. clicking on a button, because PalmOS doesn't signal the event!! Even if we click on another item, the graffiti text input remains with the text field, unless the 'other item' clicked on is itself a field.

There are two possible approaches:

1. Only 'leave' the field when we move to another field. The problem is that if we choose this approach, then the trigger to update the database may not occur before we 'accept' the menu by clicking on a button. We therefore say:

2. Whenever we click on another item, we must check whether ACTIVE-FIELD has been set, AND whether it has been *altered* (ie. its value isn't synchronised with the database). If both conditions are met, then we must fire off our own 'field exit' event, even if the focus is still retained by that event. Future changes entered in the graffiti keypad or whatever will only be actioned when we click another item, but now it's ok, as the response to the change will occur before the response to the button, or whatever!

4.8.1 Did a field change?

Here we find the current value of `ACTIVEFIELD`, and compare this to the value stored in FIELDSNAPSHOT. We return 1 if a change has occurred, otherwise zero.

```
Int16 utility::DidLastFieldChange()
{
  Char * newfld;
  Int16 newlen;
  MemHandle memh;
  Int16 ok = 0; // no change
```

```
  if (! ACTIVEFIELD) // if not defined, 'unchanged'!
     { return 0;
     };
  memh = w_FldGetTextHandle(ACTIVEFIELD);
  if (! memh)
     { return 0; // hack!
     };
  newfld = (Char *) w_MemHandleLock (memh); // MHL4
  if (! newfld)
     { return 0;
     };
  newlen = w_StrLen(newfld);
  if (newlen > MAXFIELDSNAPSHOT)
     { newlen = MAXFIELDSNAPSHOT; // hack
     };
  if (newlen != SNAPLENGTH) // if differ
     { ok = 1;
     } else
     { if (! xSame (newfld, FIELDSNAPSHOT, newlen))
          { ok = 1;
          };
     };
  if (ok) // if changed, store:
     { xCopy(FIELDSNAPSHOT, newfld, newlen);
       SNAPLENGTH = newlen;
     };
  w_MemPtrUnlock( (MemPtr) newfld);
  return ok;
}
```

Note the size of MAXFIELDSNAPSHOT, which is hard coded, so we must be cautious never to allow fields bigger than this value [FIX ME]!

4.8.2 Get text from field

This rather ugly function returns not only the size of the snapshot string (placed in the integer pointed at by the sole argument) but also an actual pointer to the field (returned as a character pointer). There is *no* ASCIIZ termination of the string.

```
Char * utility::GetFieldText(Int16 * fldlen)
{
  * fldlen = SNAPLENGTH;
  return FIELDSNAPSHOT; // no 0x0 terminal
};
```

4.8.3 Set currently active field

Given a new text field, we abandon the old one, and also copy over the text from the new field into the snapshot string (unless the 'new field' is null).

```
Int16 utility::SetActiveField (FieldType * fld, Int16 fldid)
{ ACTIVEFIELD = fld;  // simply abandon old field
  ACTIVEFIELDID = fldid;
  if (! ACTIVEFIELD) // if null field, just quit
     { SNAPLENGTH = 0;
       return 1;
     };
  Char * newfld;
  Int16 newlen;
  MemHandle memh;
  memh = w_FldGetTextHandle(ACTIVEFIELD);
  if (! memh)
     { SNAPLENGTH = 0;
       return 0; // fail.
     };
  newfld = (Char *) w_MemHandleLock (memh); // MHL5
  if (! newfld)
     { SNAPLENGTH = 0;
       return 0;
     };
  newlen = w_StrLen(newfld);
  xCopy(FIELDSNAPSHOT, newfld, newlen);
  SNAPLENGTH = newlen;
  w_MemPtrUnlock( (MemPtr) newfld);
  return 1;
}
```

4.8.4 Get ID of a field

Here we simply return the ID of the current (active) field.

```
Int16 utility::GetFieldID ()
{
  return ACTIVEFIELDID;
}
```

4.9 Utility 'database' functions

Most of our SQL functions have already been moved out to the relevant library (imaginatively called the 'SQL3 library'). Here are a few residua, which probably should also be moved out [FIX ME]! Most of the following are not SQL-specific,

and merely refer to PalmOS ‘database’ (i.e. file) management. They should be self-explanatory.

4.9.1 Open PalmOS ‘file’

Locate and open a PalmOS database.

```
DmOpenRef utility::PalmFileOpen (Char *databaseName, Int16 namlen)
{ LocalID mydbid;
  DmOpenRef mydbref;
  mydbid = u_DmFindDatabase(databaseName, namlen);
  if (! mydbid)
     { //
       ERRmsg(ErPalmDbNotFound);
       ERRSTRING(databaseName, namlen); // [??]
       return 0; //fail
     };
  mydbref = w_DmOpenDatabase (mydbid, dmModeReadWrite);
  if (! mydbref)
     { ERRmsg(ErPalmDbNotOpen);
       return 0; //fail
     };
  return mydbref;
}
```

4.9.2 Close PalmOS ‘file’

```
Int16 utility::PalmFileClose (DmOpenRef myDBref)
{ if (! w_DmCloseDatabase(myDBref))
     { ERRmsg(ErPalmDbCantClose); // unusual.
       return 0; //fail
     };
  return 1; //success.
}
```

4.10 Utility: the SELECT statement

SQLselect, the principal function, performs an SQL SELECT query. We use `cQUERY` and `dbLINK` structures to implement a fairly efficient query, but these structures are now hidden within the sql library.

Given a standard SQL SELECT statement, we write the result of the query to the STACK! We must structure the result so that translation into database format is fairly painless, but we won't have to create a new database for every SELECT statement. We resist the temptation to translate the result into ASCII text, as this *will* make conversion into a database cumbersome, and make handling of floating point numbers exceptionally clumsy.

During evaluation, we make use of an 'intermediate format' along the lines of:

```
<selection columns>
<SEPARATOR>
<tablelist>
<SEPARATOR>
<conditionals>
```

Each of the one or more `conditionals` is in the format:

```
<operator><condition tested><first variable>
  [<space><second operand>]<SEPARATOR>
```

where the square brackets indicate an optional item. `operator`s are instructions like AND, OR, and so on. `condition tested` is a code representing concepts like 'greater than', 'equal', etc. `first variable` is always the name of a column in dotted format ie. table.column Note that as things stand, the table name must always be included; later we might fancy things up a bit and permit implied table names, that are inserted automatically. `second operand` may be a constant value, or another column linking the table to another database table. It may be absent with the conditions IS NULL, IS NOT NULL.

An example of a statement is:

```
"SELECT tableA.col1,tableB.col2 FROM tableA,tableB
  WHERE tableA.col3 = 123"
```

Which is turned into:

```
tableA.col1,tableB.col2<SEPARATOR>
tableA,tableB<SEPARATOR>S=tableA.col3 123<SEPARATOR>
```

There are several rules (which we might ultimately lighten up on):

1. No spaces in comma lists

2. No double spaces (or more), tabs, or other non-space whitespace

In the following code we assume that the initial SELECT and the following space have already been clipped of the string. The `onlyone` parameter forces immediate return of just the first successful row encountered, a time-saving convenience.

```
Int16 utility::SQLselect (Char * selectstring, Int16 selen,
                          Int16 onlyone)
{
  Int16 ok;
  ok = SQL3SELECT (SQL3CODE,
                selectstring, selen, onlyone,
                STACK, STACKSTRING, janet);
  return ok;
};
```

The library routine is made more complex by the need to submit callback functions, an intermediate buffer, and the stack area.

In SQLselect, the length of the intermediate buffer is rather arbitrary. It could in many cases be shorter; if there are several floats in the form 1.0 or whatever, it could conceivably be too short [CHECK ME]!

4.11 Utility: the UPDATE statement

Given a WHERE selection criterion (as for SQLselect) update one or more columns in the selected records to given values. At present rather restrictive, as the value to be set is constant. Note the implications of the example given by Gulutzan and Pelzer page 672. Format is:

```
UPDATE tablename SET col=value[,col2=value2]
  WHERE condition;
```

Our procedure is to:

1. identify the table, open it, and count the columns
2. create a linked list of cQUERY structures to deal with the col=value clause
3. create a similar list to handle the condition (as for SQLselect)
4. Examine each row in the table, checking the condition, and then applying the update list to this row.

NB There is justification for creating a specific update which checks whether the single criterion is the key. It will then not search through the whole bloody table, but merely locate the relevant record and make the update. This will speed things CONSIDERABLY! WE WILL DO THIS. In addition, we should check whether key(s) are being rewritten as part of a 'bulk' update, and (contrary to standard SQL) *disallow* this practice!! (Only key update should be dedicated one, where we have actually selected the key within the WHERE clause, could even be more restrictive and make this the only component of the WHERE, which makes sense (as the mere presence of the condition returns only one record max? hmm, tricky). WHAT about "UPDATE table set key=key+1" ugly but powerful. Look at GandP example of this.

NB. The same applies to the select statement! in fact, the whole design philosophy of sql is erroneous in the sense that we are doing very different things if we say: *SELECT itemlist FROM tablename WHERE primarykey=value* versus the statement where we say: *SELECT itemlist FROM tablename WHERE arbitrary-condition-may-succeed-on-many-rows*

NB Likewise for UPDATE. The sole primary key condition is special and should be catered for using a special convention. We can fake this case by testing for the condition and responding appropriately.

```
Int16 utility::SQLupdate (Char * stmt, Int16 slen)
{
  Int16 ok;
  ok = SQL3UPDATE (SQL3CODE, stmt, slen, janet);
  return ok;
}
```

4.12 The INSERT statement

We have logically moved this statement to the SQL3 library too, so invoke it there:

```
Int16 utility::SQLinsert (Char * stmt, Int16 slen)
{
  Int16 ok;
  ok = SQL3INSERT (SQL3CODE, stmt, slen, janet);
  return ok;
}
```

We must supply the 'janet' callback to allow the function to search for records.

4.13 Utility: miscellaneous buffer functions

MakeEmptyRecord is like MakeFileRecord (See *Sql3Lib.tex*) but writes nothing.

4.13.1 Create empty record

```
Int16 utility::MakeEmptyRecord (DmOpenRef myDB, UInt16 idx,
                                Int16 datalen)
{ MemHandle mynewrec;
  UInt16* pIdx;
  // idx will only change if specified record is above top! check?
  pIdx = &idx; //point to index
  mynewrec = w_DmNewRecord(myDB, pIdx, datalen);
  if (! mynewrec)
     { ERRmsg(ErFailNewRecord2);
       return 0; //fail
     };
  if (! w_DmReleaseRecord(myDB, idx, true))
     { ERRmsg(ErFailReleaseRecord2);
       return 0;
     };
  return 1; // success.
}
```

4.13.2 Create buffer file

If file exists does *not* attempt to recreate buffer. Just exits!

```
Int16 utility::MakeDiskBuffer (Char * fname, Int16 fnlen,
                           Int16 reccount, Int16 recsize)
{
  if ( PalmFileFind(fname, fnlen))
     { return 1;  // 'success'. Already exists.
     }; // MAKES *NO* CHECK THAT THE STATED BUFFER(S) EXIST
        // OR ARE OF THE CORRECT SIZE!
  if (! PalmFileCreate (fname, fnlen) )
     { ERRmsg(ErMakeBufFile);
       return 0;  // fail
     };
  DmOpenRef pdb;
  pdb = PalmFileOpen(fname, fnlen);
  if (! pdb)  // trust nobody.
     { ERRmsg(ErOpenBufFile);
       return 0;
     };
  Int16 c=0;
  while (c < reccount)
    { if (! MakeEmptyRecord (pdb, c, recsize))
         { ERRmsg (ErMakeBufRecord);
            return 0;
         };
      c ++;
```

```
    };
  return PalmFileClose(pdb);
}
```

4.13.3 Find PalmOS ‘file’

```
Int16 utility::PalmFileFind (Char * databaseName, Int16 namlen)
{ LocalID mydbid;
  mydbid = u_DmFindDatabase(databaseName, namlen);
  if (mydbid) { return 1; };
  return 0; // verbose
}
```

4.13.4 Create stack

This routine (now) returns 0 on success, a nonzero number on failure!

```
#define sizeSI 16
  // size of an item on the STACK (16 bytes)
#define STACKMAX sizeSI * 2000
  // maximum size of STACK area

Int16 utility::OpenStack ()
{  // Each stack item occupies 16 bytes.
  if (! MakeDiskBuffer ("SQLSTACK", 8, 2, STACKMAX))
        // nearly 2K places on stack!
        { ERRmsg(ErMakeStack);
          return -1;
        };
  pdbSTACK = PalmFileOpen("SQLSTACK", 8);
  if (! pdbSTACK)
     { ERRmsg(ErOpenStack);
       return -2;
     };

  MemHandle hSTACK;
  hSTACK = w_DmGetRecord(pdbSTACK, 0);
  if (! hSTACK) // error. Likely is Palm lock despite reset: (Palm OS 5.4.9) (2008
  // after soft reset we get error code 527 (record busy). Busy bit still set afte
  // Check out http://www.llamagraphics.com/drupal/blogs/cewhite/2006/08/04/more-a
     { Int16 ec0 = ReconstituteRecord (pdbSTACK, 0); // v0.95
       Int16 ec1 = ReconstituteRecord (pdbSTACK, 1);
       if ( ec0 || ec1) // if reconstitution failed ...
          { PalmFileClose(pdbSTACK);  // close PDB database
            LocalID mydbid;
            mydbid = u_DmFindDatabase("SQLSTACK", 8);
            w_DmDeleteDatabase(mydbid); // delete [? check for failure]
            if (ec0) { return ec0; };
```

```
            return ec1; // fail
          };  // else..
       hSTACK = w_DmGetRecord(pdbSTACK, 0); // retry!
       if (! hSTACK)
          { ERRmsg(ScErRepairFailed);
            return -4; // simply fail
          };
     };
  STACK = (Char *) w_MemHandleLock(hSTACK); // MHL6
  hSTACK = w_DmGetRecord(pdbSTACK, 1);
  STACKSTRING = (Char *) w_MemHandleLock(hSTACK); // MHL7
  // could have more tests. naive.

  Int16 el=0;
  el = ScriptIni(SCRIPTLIBCODE, STACK, STACKSTRING);
  if (el != 1)
     { ERRmsg(ErNoScriptStack);
       return -3;
     }
  return 0;
}
```

Here's the rescue function ReconstituteRecord. It returns 0 on success. It's reason for existence is a recent PalmOS defect (see above).

```
Int16 utility::ReconstituteRecord( DmOpenRef dbref, Int16 recnum )
{
  if ( w_DmReleaseRecord(dbref, recnum, true)  // FIX: force dirty with 'true'!
     ) { return 0; // success
       };
  Int16 ecode = w_DmGetLastErr();
  return (ecode);
}
```

4.13.5 Close stack

```
Int16 utility::CloseStack()
{ if (! pdbSTACK)
     { return 1; // nothing to do;
     };
  if (! w_MemPtrUnlock(STACK))
     { ERRmsg(ErFailSTACKUnlock);
       return 0;
     };
  if (! w_DmReleaseRecord(pdbSTACK, 0, false))
     { ERRmsg(ErFailSTACKFree);
       return 0;
     };
```

```
  if (! w_MemPtrUnlock(STACKSTRING))
     { ERRmsg(ErFailSTACKUnlock2);
       return 0;
     };
  if (! w_DmReleaseRecord(pdbSTACK, 1, false))
     { ERRmsg(ErFailSTACKFree2);
       return 0;
     };
  if (! PalmFileClose(pdbSTACK))
     { ERRmsg(ErFailSTACKClose);
       return 0;
     };
return 1;                                          // ok
}
```

4.14 Utility list/node functions

We have been coerced into choosing the ‘packed block’ etc list option, because the rules seemed to change with passing pointers in PalmOS (more recent fx seem to check that a packed block is present).

We have to keep meticulous watch on our lists and their handles, so we can release them ’in the end’! (Note that the END is always when we enter a new menu, at which time all such lists *must* be completely deleted).

The catch is in the PalmOS packed blocks. It seemed silly to duplicate these if we have a column of similar items. We therefore tried to map many poptriggers and lists to one packed block in such circumstances. Unfortunately this failed.

We must nevertheless maintain a list of lists, which we implement using a linked list of nodes. Each node is a ’ListLink’ which as a unique id *supplied* by the invoker.

When somebody selects an item from a poplist, we need to put the relevant text onto the stack, and invoke the response script. We might set up a complex association between the poptrigger control and the list, BUT what we NOW do is to simply briefly retain the id of the poptrigger entered (and not process this further) so when we make our LIST selection we know which poptrigger was clicked — we can find the script associated with the *poptrigger* and apply it to the selected item (text) from the list!

4.14.1 Create list node

We create a new poplist-associated node. For further explanation see Section 4.14.3.

```
void * utility::NewListNode (Int16 id, ActualList * alist)
```

```
{ ListLink * lnk;
  // ASSUMES we have already checked for a node with a duplicate id ??
  lnk = (struct ListLink *) xNew( sizeof(struct ListLink), 0x04); // XNW4
  if (! lnk)
     { ERRmsg(ErNoListMemory);    // ? nested error
       return 0;                        // fail
     };
  xFill((Char *)lnk, sizeof(struct ListLink), 0x0);
  lnk->id = id;
  lnk->alist = alist; // insert ref to actual list
  alist->count ++;    // and bump reference count!
   // the above clears everything including pointers and ->next
  return lnk;
}
```

4.14.2 Find list node

```
void * utility::FindListNode (Int16 id)
{ ListLink * lnk;

  lnk = ROOTLINK;
  while (lnk)
    { if (lnk->id == id)
         { return lnk;
         };
      lnk = lnk->next;
    };
  return 0; // fail
};
```

Similar is the routine to find a list node using the ID of the associated poplist:

```
void * utility::FindListByPopper (Int16 id)
{ ListLink * lnk;

  lnk = ROOTLINK;
  while (lnk)
    { if (lnk->popper == id)
         { return lnk;
         };
      lnk = lnk->next;
    };
  return 0; // fail
};
```

4.14.3 Insert list node

We must keep a linked list of nodes which tell us about each poplist we have made. This requirement is not only forced upon us by PalmOS design (otherwise we get memory leaks) but is also useful. The ListLink structure is defined in *palmsql3.h* (See Section 7.2.2). InsertListNode is only invoked by FormNewControl, which in turn is invoked by the widely used InsertWidget.

```
Int16 utility::InsertListNode (Int16 id, Char * txtlist,
                        MemHandle keepme, Int16 choices,
                        ActualList * alist, Int16 popper)
{ ListLink * lnk;

  lnk = (ListLink *) FindListNode(id);
  if (lnk)
     { return 1;  // already exists [? return 0]
     };
  lnk = (ListLink *) NewListNode(id, alist);
  if (! lnk)
     { ERRDisplay("!lnode?",0); // [???]
       return 0;
     };
  lnk->txtlist = txtlist;
  lnk->keepme = keepme;
  lnk->choices = choices;
  lnk->popper = popper;
  lnk->next = ROOTLINK;   // finally, prepend lnk to list!
  ROOTLINK = lnk;
  return 1;
}
```

The contained txtlinst and keepme items are necessary for normal PalmOS function, and must be destroyed when the node is, to prevent memory leaks.

4.14.4 Delete all list nodes

```
Int16 utility::KillListNodes ()
{ ListLink * lnk;
  Char * dead;
  MemHandle deadM;
  ActualList * alist=0;

  lnk = ROOTLINK;
  while (lnk)
    {
```

```
      alist = lnk->alist;
      if (alist)
         { alist->count --;
           if (alist->count < 1)
              { Delete(alist->auxlist);
                Delete( (Char *) alist); // ? success
              };
         };

      dead = lnk->txtlist;
      // this is standard xNew-style memory, so simply:
      if (dead)
         { Delete(dead); // might check for success?
         };
      deadM = lnk->keepme;
      // this is PalmOS allocated, so need different approach:
      if (deadM)
         { if (! w_MemHandleUnlock (deadM))
              { ERRmsg(ErMemoryNoUnlock);
                return 0;
              };
           if (! w_MemHandleFree (deadM))
              { ERRmsg(ErMemoryNoFree);
                return 0;
              };
         };
      dead = (Char *) lnk;
      lnk = lnk->next; // NOW can delete old lnk (aka dead)
      Delete(dead);    // delete actual body of link
    };
  ROOTLINK = 0; // NB.
  return 1;
}
```

4.14.5 Text nodes

These have a minor function — we need them to be able to make text labels clickable! As PalmOs labels aren't, we use fields as labels, disabling text entry and not underlining them. The catch is that the field needs associated text, which we later need to free up, hence our linked list of text nodes. We have functions to insert a new node or kill all such nodes.

```
Char * utility::InsertTextNode(Int16 code, Char * p, Int16 len)
{ TextNode * tnp;
  Char * txt;

  if (len < 0)
```

```
     { return 0;
     };
  tnp = (TextNode *) xNew( sizeof(TextNode), 0x51); // XNW5
  txt = xNew(len+1, 0x52); // +1 for asciiz. // XNW6
  if (! txt)
     { return 0;
     };
  xCopy(txt, p, len);
  *(txt+len) = 0x0; // for sake of PalmOs, have ASCIIZ (ugh)

  tnp->txt  = txt;
  tnp->id   = code;
  tnp->next = TEXTROOT; // insert at front
  TEXTROOT = tnp;       //
  return txt; // return ptr to NEW text string!
}
```

The text string passed in p *will be retained* until deleted by the killing function, so do not free up this memory! Next, the killing function which is called on menu exit.

```
Int16 utility::KillAllTextNodes()
{ TextNode * tnp;
  TextNode * nxt;
  Char * p;

  tnp = TEXTROOT;
  TEXTROOT = 0;

  while (tnp) // might limit count in case of error ??
    { nxt = tnp->next;
      p = tnp->txt;
      Delete(p); // might check success?
      Delete((Char *)tnp);
      tnp = nxt;
    };
  return 1;
}
```

4.15 Utility: console functions

The console is a separate program which allows us to write messages to a large console area. The idea is to pop up the console (like a DOS box monitoring Perl under Windows) and read the messages!

To launch the console, try the PalmOS function SysAppLaunch. The console application cannot have any globals!

```
Int16 utility::LaunchConsole ()
{
  Int16 ok;
  UInt32 rslt;  // return value goes here!
  LocalID conid;

  conid =  DmFindDatabase (MAINCARD, "console");
  if (! conid)
     { ERRDisplay("F_ck!", 0); // [???]
       return 0;
     };
  ok = SysAppLaunch (MAINCARD, conid,
                     0,
                     sysAppLaunchCmdNormalLaunch, // launch code
                     0, // param block
                     & rslt);
  return ok;
}
```

5 Higher level routines

The name of this class (`sql`) is something of a misnomer, as the predominant functions here are concerned with script interpretation more than SQL. (A lot of the SQL stuff is now within the `utility` class, or has been moved to the SQL3 library). This section is large and complex.

5.1 Includes

We include the utility class file, as well as headers of various libraries.

```
#include <PalmOS.h>
#include "utility.hpp"
#include "err/ERRDEBUG.h"
#include "scripting/SCRIPTING.h"
#include "numeric/NUMERIC.h"
#include "idx/IDX.h"
#include "cache/cache.h"
```

IDX.h is the header for the indexing library, used to test this before we incorporate calls to it within the SQL library. In addition, we define several constants which more properly should be placed in a header file somewhere. Important constants are:

- FXMAX. Scripted functions are contained within a PalmOS database, but we create an index into this database to allow easy access to function definitions! FXMAX is the maximum size of the buffer area. Each function reference occupies just 16 bytes.
- FXSTACKSIZE. This defines the maximum size of the function stack, onto which we push invoked functions, allowing deep nesting of functions and some recursion. Each pushed function requires 8 bytes. The value of FXSTACKSIZE must be divisible by 8.
- MENUSTACKSIZE. Likewise, we can push menus to a depth of 16. sizeM is the size of a menu on the stack (8 bytes).
- LOCALSIZE. The size of the buffer space used for local variables within a menu.
- MINLENRUNSTMT is the minimum character length of a statement which can be executed.

- Check the following, several of which may now be redundant: BadMakeTestTable, MAXCOMMAND [FIX ME]!

```
#define FXMAX 4096
#define FXSTACKSIZE 1024
#define MENUSTACKSIZE 16
#define sizeM 8
#define LOCALSIZE 4096
#define MINLENRUNSTMT 4

#define BadMakeTestTable  +(SqlBadBase+1)
#define MAXCOMMAND 1024
```

We define certain values simply for the purposes of debugging:

```
#define DEBUG_WIDGET 1
#define DEBUG_SQL 1
```

5.2 Class definition

The class definition is long but straightforward, with our usual constructor and destructor stubs. Most of the functions and variables are private. First the public functions:

```
class sql : public utility {
public:
         sql () { };  // stub
         ~sql () { }; //
Int16    Respond(Int16 item, Int16 myevent, Int16 whatctl,
                 EventPtr e);
Int16    Kickoff();
Int16    Cleanup();
Int16    NewMenu();
Int16    ForceTimeout(Int16 i); // set/clear timeout flag
Int16    IsTimeout(); // get timeout flag
Int16    PostMenuEvent();
Int16    SleepMenu ();
Int16    EnterMenu(Char * mname, Int16 nlen);
Int16    FetchMenuDepth(); // debug only
Int16    FindWidgetType (Int32 uid);
Int32    FindWidgetUID (Int16 dbid);
Int16    KillOnId (Int32 uid);
Int16    StackDump(Int16 icount);
```

```
Int16    Confirm(Char * msg);
Int16    PushString(Char * str, Int16 slen);
Int16    WriteRestorationData();
Int16    RestorePriorState();
```

Here's the private stuff, first a number of variables. The highlights:

- DEBUGFLAGS is used in debugging. Various types of item are debugged, depending on bit flags set within this Int16.
- FUNFILE references the opened PalmOS FUN file containing our script functions;
- FxBUFFER contains references to active functions (the number of which is contained in FxCOUNT). pdbFXBUFFER is associated.
- LOCALS contains local scripting variables.
- STOPPED is a nasty hack which signals cessation of a script caused by invocation of the STOP function. We need to sit down and re-engineer the rather convoluted logic of ExeScript to permit reliable return of the value IVESTOPPED [??? fix me — cf Perl]! EXEITEM is a clumsy record of the current ExeScript item (needed so that when we exit ExeScript to obtain a widget response, we can resume).
- MENUSTACK. By keeping the unique database ID of the menu, we can readily reload it when we 'pop' the menu off the stack. The value of sizeM is currently 8. Each menustack record of a menu occupies sizeM bytes: we index into MENUSTACK using MENUINDEX at offset +0 we have the Int16 code for the menu in the database (theoretically could have int32 but we never use over int16)(hmm); at offset +2 we store the 32 bit X-parameter. The remaining 2 bytes should be zero unless the recently added 'rolling menu' option is activated.[22] CURRENTMENU is the database ID of the current menu.
- XPARAM is the current value of X, the sole parameter transferred between menus (as in the Perl program). It is (now) a 32-bit integer.[23]
- RUNLEVEL is important in knowing when to clear RERUN (clearly only after the currently running script has been returned from!)

[22] Section 5.4.8. If nonzero, then a polymorphic table has not been completely displayed. The number at offset +7 i.e. the last byte is the current offset in that menu.

[23] Formerly 16 bits, which caused endless problems!

- SQLOK records whether most recent SQL statement was successful (1) or not (0).

```
private:
  Int16    DEBUGFLAGS;
  WIDGET * ROOTWIDGET;
  FormPtr  MYFORM;
  FormPtr  ASKFORM;
  Int16    OKBUTTON;
  Int16    QUITBUTTON;
  Int16    USERTEXT;
  DmOpenRef FUNFILE;
  Char *   FxBUFFER;
  Int16    FxCOUNT;
  DmOpenRef pdbFXBUFFER;
  Char *   FxSORTED;
  Char *   FxSTACK;
  Int16    topFxSTACK;
  DmOpenRef pdbLOCALS;
  Char     * LOCALS;
  Boolean  STOPPED;
  Int32    CURRENTUSER;
  Int32    EXEITEM;
  Char *   MENUSTACK;
  Int16    MENUINDEX;
  Int32    NEWPARAM;
  Int32    XPARAM;
  Int16    CURRENTMENU;
  Char *   RERUN;
  Int16    RUNLEVEL;
  Int16    SQLOK;
  Int16    ABLING;
  Int16    LOCALROLL; // local rolldown offset in polytable
  Int16    ROLLING;
  Int16    ROLLDEPTH; // current polytable offset
  Int16    LINESLEFT; // unviewed polytable lines
  Int16    isFAILURE;
  Int16    TIMEOUTFLAG; // is timeout set?
```

The ABLING variable has been added (2/2006) to address the problem where an initialisation script either enables or disables a widget. As the initialisation script is run *before* the widget is created,[24] we cannot find and en/disable the widget. Thus we put the value in ABLING, and always check and clear this value at the time of widget creation.

ROLLDEPTH is used in rolling down a polymorphic table to the next copy of the current menu (where needed), and ROLLING is actually Boolean, 0 being

[24] This must occur, as the ini script dictates whether the widget will be created!

the default and 1 if a NewMenu command is to roll down a polymorphic table. LOCALROLL measures the *next* additional number of lines to roll down. See Section 5.4.8.

isFAILURE is an interesting (and rather ugly) global variable which we use solely to signal that a script has posted a FAIL command. We interpret this as an instruction to stop creating a particular widget!

5.2.1 A note on reloading a sequence of menus

PalmOS is not a multitasking system, a major liability. Consider the following example. You are running the pain database on a Palm PDA which is also a cellphone, and somebody rings you on the cellphone. Palm interrupts your program to take the call . . . and on re-entering you have to reload the whole program!

Fortunately, we have minimised the information stored when we are in a particular PainForm menu, and we only transfer a single numeric item (the X-variable) between menus. We can therefore easily store and re-construct our current state. We must store/restore the following values:

- The value in MENUINDEX;
- The contents of the MENUSTACK including the current values of XPARAM etc pushed to MENUSTACK (that is, sizeM * (1+MENUINDEX) bytes).
- CURRENTUSER !
- A wrinkle: Because of internal caching to optimise menu access, we must also store this caching information, which is a non-trivial task [FIX ME!!]

NB: Make sure the following are loaded (or cleared) as normal, on re-entering. We ‘clear’ (check me!) NEWPARAM, RERUN, RUNLEVEL, SQLOK, ABLING, LOCALROLL, ROLLING, ROLLDEPTH, LINESLEFT and isFAILURE. We enter appropriate values in XPARAM and CURRENTMENU.

We need to have a good look at the rolling menus, but it might be better to clear any rolling, rather than fiddling [CHECK ME, FIX ME]!

5.2.2 function list

Here’s a depressingly long list of functions:

```
Int16    MakeLocalVariables();
Int16    KillLocalVariables();
Int16    MakeTextForm (Char * title, Char * content);
Int16    KillTextForm ();
```

```
Int16    RespondToAsk(Int16 item);
WIDGET * MakeWidget (Int32 uid, Int16 dbid, Int32 v, Int16 mytype);
Int16    KillOneWidget( WIDGET * wp);
Int16    KillAllWidgets ();
Int16    FindWidgetDatabaseCode (Int32 uid);
Int32    VValue (Int32 uid);
Int16    PrependWidget (Int32 uid, Int16 dbid, Int32 V, Int16 mytype);
Int16    CreateOneWidget(Int32 widgetcode,
         Int16 x, Int16 y, Int16 w, Int16 h,
         Boolean recur, Int16 grp, Int16 able);
ActualList * MakeActualList(Int32 widgetcode);

Int16    CreateManyWidgets(Int16 thismenu,
         Int16 x, Int16 y, Int16 w, Int16 h,
         Boolean recur);
Int16    CreateOneTable(Int32 widgetcode,
         Int16 x, Int16 y, Int16 w, Int16 h,
         Char * iinitial);
Int16    CreatePolyTable(Int32 tablecode,
         Int16 x, Int16 y, Int16 w, Int16 h, Int16 grp);
Int16    MakeWholeColumn(Int16 itmid, Char * IDarray,
         Int16 idcount,
         Int16 xj, Int16 yj, Int16 iw, Int16 ih, Int16 nabled, Int16 mlines,
         Int16 grp);
Int16    GetRowHeight(Int16 lines, Int16 h);
Int16    GetNumLines(Int32 tablecode);
void *   HackWidget (Int16 id, Int16 * MYKIND);
Int16    SetPaperOrInk (Int16 id, Int16 isink);
Int16    Enabled (Int16 id, Int16 yesno);
Int32    HexColour (Char * clrstring, Int16 clen);
Int16    ExeScript(Int32 item);
Int16    PushCurrentScript (Char * scriptline, Int16 remlen);
Int16    FindDelimiter(Char * P, Int16 d);
Char *   FindDelimiter2( Int16 * thislen,  Int16 * slLEN,
         Char * scriptline);
Int16    ExeQuery(Int16 onlyone);
Int16    ExeSQL();
Int16    ExeString(Int32 xitem, Char * str, Int16 slen);
Int16    PushInteger(Int32 i);
Int16    PopFloat(double * d);
Int16    PopInteger(Int32 * IP);
Int16    ResolveX(Char * P, Int16 plen);
Int16    PopString(Char * P, Int16 plen);
Char *   PeekAndPopStringZ(Int16 e);
Int16    SetLabel (Int16 xitem);
Int16    IniFxBuffer();
Int16    CloseFxBuffer();
Int16    SayAlert();
```

```
Int16    PEEKDEBUG();
Int16    PopMenu();
Int16    GoScript(Char * cmd);
Int16    PushMenu();
Char *   FindFunction(Char * thiscommand, Int16 thislen,
         Int16 * foundlen);
Char *   DoReturn(Int16 * retlen, Boolean istop);
Int16    ShowStackItem(Char * ISRC, Int16 icount, Boolean recur);
Int16    QuickBy(Int16 code);
Int16    PeekType();
Int16    Redraw(Int16 dbid);
Int16    InsertAndGo(Int32 widgetcode, Char * cmd);
Int16    InsertAndGoV(Int32 widgetcode, Char * cmd, Int32 V);
Int16    JustGo(Char * cmd);
Int16    PeekLengthPlus();
};
```

5.3 Initialisation and clean-up

Here we first set various flags depending on our current debugging plan. In later development the code in CURRENTUSER might even be set by a security routine! We also set aside an area of memory for the MENUSTACK; a MENUINDEX of zero signals that the menu stack is empty. XPARAM at startup is zero.

The routine fails if function buffer initialisation fails (IniFxBuffer), as well as if we fail to make local variables. SQLOK starts off at zero, as there has been no recent successful SQL statement.

SetupUtility is (and must be) invoked *before* Kickoff, which sets up debugging flags. Kickoff (within sql module) has access to the handles of the various libraries which have already been loaded, for example IDXCODE.

```
Int16 sql::Kickoff()
{
/*  DEBUGFLAGS =  fDEBUG_STMT |
                  fDEBUG_SQL | fDEBUG_SQK | fDEBUG_SQJ |
                  fDEBUG_WIDGET | fDEBUG_DUMP |
                  fDEBUG_STACK | fDEBUG_AAGH;
*/
  DEBUGFLAGS = 0;  // limit debugging
  Int16 fail;

  TIMEOUTFLAG = 0; // off by default
  RERUN = 0;
  ABLING = -1;
  ROLLING = 0; // do NOT roll polytable (default)
  ROLLDEPTH = 0; // and initial offset is zero
  LOCALROLL = 0;
```

```
  LINESLEFT =0;

  CURRENTUSER = 1;
  QUITBUTTON = -1;
  OKBUTTON = -1;
  USERTEXT = -1;
  MYFORM = 0;
  ROOTWIDGET=0;
  MENUINDEX=0;
  XPARAM = 0;
  CURRENTMENU = 0;
  NEWPARAM = -1; // empty
  MENUSTACK = xNew(MENUSTACKSIZE*sizeM, 0xDC); // XNW7

  fail = IniFxBuffer();  // failure codes usually -1 to -100
  if (fail)
     { return fail;
     };

  fail = MakeLocalVariables(); // failure codes < -100
  if (fail)
     { return fail;
     };

  MirrorDebug(DEBUGFLAGS); // ?

///  // the following passes codes to IdxLib:
///  if (PassIndexBug (IDXCODE, DEBUGFLAGS, ELIBCODE, CONSOLECODE) < 0)
///     { return -3; // fail
///     }; /// [disabled at present]

  if (PassCacheBug (CACHECODE, DEBUGFLAGS, ELIBCODE, CONSOLECODE) < 0)
     { return -4; // fail (hmm)
     };

  SQLOK = 0;
  STOPPED = 0;   // not stopped [? need]
  isFAILURE = 0; // no failed widgets
  return 0;      // success
}
```

Kickoff returns 0 on success, or an error code on failure.

Cleanup closes down buffers and so forth before program exit. The local variables and menu stack are deleted, as is the function buffer, and then all menu objects rooted in ROOTWIDGET are cleaned up. Finally we erase and delete the current form.

```
Int16 sql::Cleanup()
```

```
{
  if (! KillLocalVariables())
     { return 0;
     };
  Delete(MENUSTACK);
  if (! CloseFxBuffer())
     { return 0;
     };
  KillAllWidgets();
  if (MYFORM)
     {
       Char * hdr;
       hdr = w_FrmGetTitle (MYFORM);
       Delete (hdr); // See: FormMakeDynamic
       w_FrmEraseForm(MYFORM);
       w_FrmDeleteForm(MYFORM);
     };
  return 1;
};
```

Finally, some additions (18/5/2008 v 0.95) to allow timeout if the application has been sleeping too long:

```
Int16 sql::ForceTimeout(Int16 i) // set/clear timeout flag
{
 TIMEOUTFLAG = i;
 return (0); // unused at present.
};

Int16 sql::IsTimeout() // get timeout flag
{ Int16 i = TIMEOUTFLAG;
  return(i);
};
```

5.4 Menu handling

We retain the XPARAM and CURRENTMENU values until we move to a new menu, at which point we push the current XPARAM and CURRENTMENU values, and load new values into CURRENTMENU and (where NEWPARAM isn't -1) into XPARAM. When we move back, we throw away these values, and reload them from the MENUSTACK. MENUINDEX points to the next FREE spot, unless actually being altered. We do NOT push/store CURRENTMENU and XPARAM until we go to another menu.

5.4.1 EnterMenu

EnterMenu is used by *pain5.cpp* to kickoff the menu system. Simply push the menu name to the stack, then invoke NewMenu.

```
Int16 sql::EnterMenu(Char * mname, Int16 nlen)
{
   PushString(mname, nlen);
   return NewMenu();
}
```

5.4.2 FetchMenuDepth

A debugging function which simply returns the value in MENUINDEX:

```
Int16 sql::FetchMenuDepth()
{
  return (MENUINDEX);
}
```

5.4.3 WriteRestorationData

Open the database ‘REENTRY.SQ3’ on the Palm PDA, and write all required data to it. The function RestorePriorState will read this and give the user the option of resuming. Ideally the latter function should also check the time that the exit occurred, and when re-entry occurred, and if this is more than say 5 min, refuse entry.

```
Int16 sql::WriteRestorationData()
{
  // write all data necessary for restoration of current state after exit and re-e
  // first open PalmOS database 'file', clearing old contents:

  LocalID   wrid;        // nasty palmos stuff..
  DmOpenRef restref;
  MemHandle hres;
  UInt32    recsize;
  UInt32    sysTime;
  UInt16    myTime;
  Char *    pRes;
  Int16     i;
  UInt16    rref=0;       // index of record
  Int16     ok;

  recsize = 8 + MENUSTACKSIZE*sizeM + 0; // (max!); ?later add caching stuff ipo +
    // (better to let cache library handle that side of things)
```

```
wrid = u_DmFindDatabase ("REENTRY.SQ3", 11);
if (! wrid)
   { if (! PalmFileCreate ("REENTRY.SQ3", 11))
        { return -1; // just fail ??
        };
     restref = PalmFileOpen("REENTRY.SQ3", 11);
     MakeEmptyRecord (restref, rref, recsize);
   } else
   { restref = w_DmOpenDatabase (wrid, dmModeReadWrite);
   };
hres = w_DmGetRecord(restref, rref); // must exist
pRes = (Char *) w_MemHandleLock(hres); // MHL8

// write sizeM at offset +2
i = sizeM;
w_DmWrite(pRes, 2, &i, 2 );

// here we might write the exit timestamp (but not at present) at offset +4
// TimGetSeconds gets number of seconds since 12 AM on Jan 1st, 1904.
// We first divide by 60 to get minutes.
// We aren't interested in long periods, so we then take the number modulo 14400
// (ten days) and store THIS. It fits easily into a UInt16.
sysTime = TimGetSeconds(); // [perhaps wrap TimGetSeconds]
sysTime /= 60;
myTime = (UInt16) (sysTime % 14400);
  ///   WriteConsoleAsciiz(fDEBUG_ALWAYS, "\n Debug! writing time "); //
  ///   WriteConsoleInteger(fDEBUG_ALWAYS, myTime);
w_DmWrite(pRes, 4, &myTime, 2);

// write 0000 at offset +6
// note (29/1/2008: write CURRENTUSER) ???
i = (Int16) CURRENTUSER; // note limitation to Int16 [?!]
w_DmWrite(pRes, 6, &i, 2 );

// write MENUINDEX at offset +0 // ZERO VALUE HERE will (later) mean 'cleared'
i = MENUINDEX;
w_DmWrite(pRes, 0, &i, 2 );
i *= sizeM;
// [here might balk on bad i]
// write MENUSTACK contents at offset +8 for size sizeM*MENUINDEX [???]
*((Int16 *)(MENUSTACK + 0 + i)) = CURRENTMENU;
*((Int32 *)(MENUSTACK + 2 + i)) = XPARAM;
*((Int16 *)(MENUSTACK + 6 + i)) = ROLLDEPTH;
i += sizeM;  // MENUINDEX+1 now permits capture of current menu too!
w_DmWrite(pRes, 8, MENUSTACK, i );

// close file
```

```
  MemHandleUnlock(hres);
  w_DmReleaseRecord(restref, rref, false); // not dirty, no check.
  w_DmCloseDatabase(restref);

  // last, order the cache library to keep a copy of its current state:
  ok = CACHESTORESTATE(CACHECODE);
  if (ok < 1)
     { return (ok-100);
     };

  return 1; // success.
}
```

In the above we *must* write the values of CURRENTMENU etc to the MENUSTACK, as this is otherwise not done by our program until we push the lot! We also save the area we've written, of course.

5.4.4 RestorePriorState

Performs the reverse of WriteRestorationData. We tried to keep the cache manager resident but on failing dismally, we wrote the routines needed to store the cache manager library data and then restore these linked lists, which we do here (CACHESTATERESTORE). Returns 1 on success, 0 or less on failure. A return value of zero implies that no restoration has taken place, but there was no error.

```
Int16 sql::RestorePriorState()
{
  // Does the file 'REENTRY.SQ3' exist (with a size > 0)? If not,
  // simply return 0.

  LocalID   wrid;        // nasty palmos stuff..
  DmOpenRef restref;
  MemHandle hres;
  Char *    pRes;
  Int16     i;
  UInt16    rref=0;       // index of record
  Int16     zero = 0;
  Int16     fail=0;
  Char *    pMsg;
  Int16     mlen;
  UInt32    sysTime;
  UInt16    myTime=0;
  UInt16    oldTime;

  wrid = u_DmFindDatabase ("REENTRY.SQ3", 11);
  if (! wrid)
```

```
     { return fail; // fail
     };
  restref = w_DmOpenDatabase (wrid, dmModeReadWrite);  //
  hres = w_DmGetRecord(restref, rref);
  if (! hres)
     { return fail; // fail
     };
  pRes = (Char *) w_MemHandleLock(hres); // MHL9

  // ? confirm that the user wishes to resume
  // FOR NOW, JUST RESUME! (forced)

  // at offset + 0: MENUINDEX
  MENUINDEX = * ((Int16*) (pRes+0));
  if (MENUINDEX <= 0)
     {
fin:   // spaghetti, spaghetti, ugh.
       MENUINDEX = 0;
       w_DmWrite(pRes, 0, &zero, 2 ); // clear memory
       MemHandleUnlock(hres);
       w_DmReleaseRecord(restref, rref, false); // not dirty, no check.
       w_DmCloseDatabase(restref);
       return fail; // fail
     };

  // @+4, exit timestamp (unused for now);
  sysTime = TimGetSeconds(); // [perhaps wrap TimGetSeconds]
  sysTime /= 60; // similar to when written
    ///   WriteConsoleAsciiz(fDEBUG_ALWAYS, "\n Debug: sys time "); //
    ///   WriteConsoleInteger(fDEBUG_ALWAYS, sysTime);
  myTime = (UInt16) (sysTime % 14400);
    ///   WriteConsoleAsciiz(fDEBUG_ALWAYS, "\n Debug: my time "); //
    ///   WriteConsoleInteger(fDEBUG_ALWAYS,myTime);
  oldTime = *((UInt16*) (pRes+4)); // see writing of this value!
    ///   WriteConsoleAsciiz(fDEBUG_ALWAYS, "\n Debug: old time "); //
    ///   WriteConsoleInteger(fDEBUG_ALWAYS, oldTime);
  if (oldTime > myTime)
     { myTime += 14400; // wrap! Max is thus 5 days. [hmm]
     };
  if ((myTime - oldTime) > ((MYTIMEOUTSECONDS+120)/60)) // if overtime..
     { goto fin; // should not happen as should wake up earlier!
     };

  // @+6, current user:
  // note (29/1/2008: must restore CURRENTUSER)
  i = *((Int16*) (pRes+6));
  CURRENTUSER = (Int32) i;
  // HERE CONFIRM THE USER (Honour system):
```

```
  mlen = 16;
  pMsg = xNew(mlen+2+maxStrIToALen,0); // XNW8
  xCopy (pMsg, "Is your user ID ", mlen); // len is 16.
  StrIToA (pMsg+mlen, CURRENTUSER); // [might wrap StrIToA]
  mlen += StrLen(pMsg+mlen);        // [might wrap StrLen]
  xCopy (pMsg+mlen, "?", 1);
  *(pMsg+mlen+1) = 0x0; // ASCIIZ
  fail = Confirm(pMsg);
  Delete(pMsg);
  if (! fail) { goto fin; }; // fail of zero is OK!

  // @+2, sizeM:
  fail = -1;
  i = *((Int16*) (pRes+2));
  if (i != sizeM)
     { goto fin;
     };

  // @+8 for length sizeM * MENUINDEX: menustack data:
  i *= (MENUINDEX+1);
  xCopy (MENUSTACK, (pRes+8), i);

  // NB MENUINDEX+1 ALLOWS CAPTURE OF *CURRENT* MENU !!

  w_DmWrite(pRes, 0, &zero, 2 ); // clear memory to permit normal exit+re-entry!
  MemHandleUnlock(hres);
  w_DmReleaseRecord(restref, rref, false); // not dirty, no check.
  w_DmCloseDatabase(restref);

  // [here restore caching:]
  fail = CACHESTATERESTORE(CACHECODE);
  if (fail < 1)
     { return fail;
     };

  // WE STILL NEED TO RELOAD THE TOP MENU:
  CURRENTMENU  = *((Int16 *)(MENUSTACK + 0 + sizeM*MENUINDEX));
  XPARAM       = *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX));
  ROLLDEPTH    = *((Int16 *)(MENUSTACK + 6 + sizeM*MENUINDEX));

  ROLLING = 0;     // hmm.
  PushInteger(0);  // force reload of top menu?!
  NewMenu();       // thus!
 // MENUINDEX --;  // hmm. cf NewMenu!

  return 1; // success.
}
```

We originally tried creating a simple mechanism where we left the cache li-

brary in situ, keeping the code used to reference it, and attempting to reference the library from the reloaded main program. The problem with this approach is that it simply doesn't work, as there is no association between the new program and the old cache library reference.

So we unload the library but retain the internal details (write to a PalmOS database) and then reload these details the next time around! [ugh] When we exit the cache library, we must also make sure we don't delete the associated 'p-files'. We write cache data to the 'file' CACHESTORE.SQ3 on the Palm. See *CacheLib.tex* for details!

5.4.5 PostMenuEvent

```
Int16 sql::PostMenuEvent()
{
   EventType event;
   event.eType = MyMenuEvent;
   // for now we have NO datum attached;
   //    e.g. event.data.generic.datum[0] = 0x0;
   EvtAddEventToQueue( &event );
   return IMWAITING; // ??? [CHECK ME]
}
```

5.4.6 NewMenu

This is the real menu function (Perl: see GoMenu). When moving *to* a new menu, we push the previous menu to MENUSTACK and go to the specified menu; moving back (number submitted) then we pop MENUSTACK. We synchronously push/pop X. [FIX ME: NEED BETTER DOCUMENTATION].

The routine is complex. After defining a few variables and cleaning things up, the first section is devoted to finding the menu within the database. This search involves clumsily checking the stack for an integer.

```
Int16 sql::NewMenu()
{
  SleepMenu();     // clean up
  Char * cmd;
  Int16 ok;        // various signals
  Char snature;    // type of item on stack
  Int32 backcount; // No. of menus to move back
  Int16 mlen;
  mlen = 128; // reasonable maximum length
  Char * MENUname;
  Char c; // used in testing for numeric string [eugh]
  MENUname = xNew(mlen, 0); // XNW9
```

```
snature = StackPeek(SCRIPTLIBCODE, STACK, STACKSTRING,
                    MENUname, &mlen);
c = *(MENUname); // NOW check for integer:
if (  (c == '-') || ( (c >= '0')&&(c <= '9') ) )
   { ok = QuickBy(iINTEGER); // convert string to integer!
     if (ok < 1)
        { ERRmsg(ErBadMenuInteger);
          Delete(MENUname); //
          ROLLING = 0; // safety
          return 0;
        };
     snature = 'I';
   };
Delete(MENUname);
```

If an integer is found (N/I) then we go *back* the requisite number of menus, popping them off the menu stack. After popping, the menuindex position on the menu stack is considered clear.

```
if (   (snature == 'N')
     ||(snature == 'I')
   )
   { PopInteger(&backcount); // No. of menus to pop & discard
     if (backcount > 0)  // skip if zero [? if -ve!]
        {
          ROLLING = 0;
          MENUINDEX -= (Int16) backcount; // 0=reload
          if (MENUINDEX < 0)
             { ERRmsg(ErTopMenu);
               return 0; // fail
             };
          CURRENTMENU  = *((Int16 *)(MENUSTACK + 0
                                     + sizeM*MENUINDEX));
          XPARAM       = *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX));
          ROLLDEPTH    = *((Int16 *)(MENUSTACK + 6 + sizeM*MENUINDEX));

          // debugging only:
          WriteConsoleText(fDEBUG_ALWAYS, "\nPOPMENU (", 9);
          WriteConsoleInteger(fDEBUG_ALWAYS, backcount);
          WriteConsoleText(fDEBUG_ALWAYS, ") ", 2);
          WriteConsoleInteger(fDEBUG_ALWAYS, CURRENTMENU);
          WriteConsoleText(fDEBUG_ALWAYS, " rd=", 4);
          WriteConsoleInteger(fDEBUG_ALWAYS, ROLLDEPTH);

        } else // if ZERO menu
        { if (ROLLING) // push copy of current!
             {
               *((Int16 *)(MENUSTACK + 0 + sizeM*MENUINDEX)) = CURRENTMENU;
```

```
            *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX)) = XPARAM;
            *((Int16 *)(MENUSTACK + 6 + sizeM*MENUINDEX)) = ROLLDEPTH;

          // debugging only:
          WriteConsoleText(fDEBUG_ALWAYS, "\nPUSHROLL ", 10);
          WriteConsoleInteger(fDEBUG_ALWAYS, CURRENTMENU);
          WriteConsoleText(fDEBUG_ALWAYS, " rd=", 4);
          WriteConsoleInteger(fDEBUG_ALWAYS, ROLLDEPTH);

            MENUINDEX ++;
            ROLLING = 0;
            ROLLDEPTH += LOCALROLL;
          };
      };
```

In the last portion (zero menu) above, if we've invoked NewMenu in order to roll down a polytable (are ROLLING, Section 5.4.8) we push the current menu to the menu stack, and then add the value in LOCALROLL to the ROLLDEPTH so that the next display of the polytable will be moved down that number of lines!

Otherwise, a non-numeric should be a text string. After confirming this, we push the old current menu and X values to the menu stack, and bump the menu index. We also push the current line offset (ROLLDEPTH) in the polytable which may be present and offset.

```
} else  // ensure text:
{
  ROLLING = 0;
  if (snature != 'V')
     { ERRmsg(ErBadMenuName);
       return 0;
     };
  *((Int16 *)(MENUSTACK + 0 + sizeM*MENUINDEX)) = CURRENTMENU;
  *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX)) = XPARAM;
  *((Int16 *)(MENUSTACK + 6 + sizeM*MENUINDEX)) = ROLLDEPTH;
  // debugging only:
  //     WriteConsoleText(fDEBUG_ALWAYS, "\nPUSHMENU ", 10);
  //     WriteConsoleInteger(fDEBUG_ALWAYS, CURRENTMENU);
  //     WriteConsoleText(fDEBUG_ALWAYS, " rd=", 4);
  //     WriteConsoleInteger(fDEBUG_ALWAYS, ROLLDEPTH);

  ROLLDEPTH = 0; // prevent further rolling [2008-08-04]
  MENUINDEX ++;
```

We then look up the database code of the name, using an sql query, and retain this code in the current menu variable. Now the default is for the X parameter

to remain unchanged, but if NEWPARAM is not -1 (see SetX), then X indeed changes.

```
      cmd = "&cSQ1";
      // "QUERY(SELECT ITEM.iID FROM ITEM
      //        WHERE ITEM.iName = '$[]')";
      JustGo(cmd);
      Int32 cMENU; // menu ID
      if (! PopInteger(&cMENU))
         { ERRmsg(ErMenuFound);
           return 0;
         };
           // debugging only:
           // WriteConsoleText(fDEBUG_ALWAYS, "\nMENU id ", 9);
           // WriteConsoleInteger(fDEBUG_ALWAYS, cMENU);

      CURRENTMENU = (Int16) cMENU;
      if (NEWPARAM != -1)
         { XPARAM = NEWPARAM;
           NEWPARAM = -1;
         };
   };
```

We next completely clear the whole stack (ScriptIni).[25] Note that in Perl we kept a copy of the local names with the potential to restore them and the menu if the menu-associated script failed. topFxSTACK = 0 is analogous to the Perl '@CMDSTACK = ()'.

```
ScriptIni(SCRIPTLIBCODE, STACK, STACKSTRING); // clear stack
topFxSTACK = 0;  // clean FUNCTION stack
LOCALROLL = 0;
LINESLEFT = 0;
```

We also clear LOCALROLL, for possible later alteration within a polymorphic table. Next, we get the floating point coordinates of the menu, a clumsy step involving another SQL invocation.

```
double dh;
double dw;
double dy;
double dx;
Int16 x;
Int16 y;
Int16 w;
Int16 h;
```

[25] What about if we are within a RUN statement? Need to examine this in detail!

```
cmd = "&cSQ2";
//  "QUERY(SELECT MENUITEMS.miX,MENUITEMS.miY,
//     MENUITEMS.miW,MENUITEMS.miH
//     FROM MENUITEMS WHERE MENUITEMS.miItem = $[])";
InsertAndGo(CURRENTMENU, cmd);
```

We obtain the float values from the stack, and then convert into pixel coordinates using the various Pixel functions from the relevant library. In fact, for now we simply force the values to prevent rude debugging messages from PalmOS.

```
PopFloat(&dh);   // might test each
PopFloat(&dw);   // for success
PopFloat(&dy);   // ...
PopFloat(&dx);
x = 2; // not: PixelX(SCRIPTLIBCODE,&dx,160);
y = 2; // not eg: PixelY(SCRIPTLIBCODE,&dy,160);
w = MenuScrWidth; // not: PixelW(SCRIPTLIBCODE,&dw,160);
h = MenuScrHt; // not: MenuYOffset + PixelH(SCRIPTLIBCODE,&dh,160);
```

At present we simply coerce all menus to $160x160$, submitting the smaller value of 156 to permit drawing of a 2 pixel border by PalmOS (eugh). We're not finished yet. We still have to select the menu title using more SQL; we then make the menu using FormMakeDynamic (Section 4.7.1).

```
Char * menutitle;
Int16 mtilen;
cmd = "&cSQ3";
//  "QUERY(SELECT ITEM.iText
//        FROM ITEM WHERE ITEM.iID = $[])";
InsertAndGo(CURRENTMENU, cmd);
mtilen = PeekLengthPlus();
if (!mtilen)
   { ERRmsg(ErNoMenuTitle);
     return 0;  // ?? stack
   };
mtilen += 0x10; // spare space
menutitle = xNew(mtilen, 0); // XNW10
mtilen = PopString(menutitle, mtilen); // clumsy
*(menutitle+mtilen) = 0x0;  // asciiz
MYFORM = FormMakeDynamic ("", x, y, w, h);
w_FrmSetTitle(MYFORM, menutitle);
// Delete(menutitle); // no!
if (! MYFORM)
   { ERRmsg (ErMakeDynamicForm);
     return 0;
   };
```

In the above we (sneakily) set the menutitle to the string using FrmSetTitle! This allows us to remove the title (see TITLE command) if we need to. When we delete the menu (See SleepMenu), we first delete this string!

Our next task is to run the initialisation script for the menu,[26] after obtaining it with yet another SQL query! The hard-coded SQL etc in the following should be fixed.

```
 //2. Run the initialisation script for the menu.
   cmd = "QUERY(SELECT ITEM.iInitial FROM ITEM WHERE ITEM.iID = $[])->RUN";
   InsertAndGo(CURRENTMENU, cmd);
```

Our penultimate task is to populate the menu with all of the relevant 'widgets' (controls and so forth). The cognate Perl routine is SubMenu. Important components are creation of individual widgets, incorporation of menus within menus, and different types of table creation. Grouping is a bit of a mess at present, and needs some work.

Finally, we activate the form (See section 4.7.2).

```
  CreateManyWidgets(CURRENTMENU, 0, 0, w, h, 1);
  FormActivate(MYFORM);
  return 1;
};
```

5.4.7 SleepMenu

This is the sleep of death! We've modified this routine to completely erase the background menu, rather than just letting it slumber in the background. Although this takes some time, it's almost certainly worthwhile in terms of complexity!

SetActiveField is used to clear the text field reference, thereby forestalling disaster. All variables should be cleared prior to entering the new menu, and the list nodes and widgets likewise abolished.

In PalmOS, form erasure should precede deletion.

```
Int16 sql::SleepMenu ()
{
 if (MYFORM)
    {
      Char * hdr;
      hdr = w_FrmGetTitle (MYFORM);
      Delete (hdr); // !See: FormMakeDynamic
```

[26] Check out ASK etc. Needs work! Note that in our Perl version, we first ran the script and then did the menu creation, which caused problems with e.g. TITLE usage within the ini script.

```
      w_FrmEraseForm(MYFORM);
      w_FrmDeleteForm(MYFORM);
      MYFORM = 0;
    };
  SetActiveField(0, 0);
  ClearVariables(SCRIPTLIBCODE, LOCALS);
  KillListNodes();
  KillAllWidgets();
  KillAllTextNodes(); // these too.
  return 0;
}
```

5.4.8 Rolling menus

On a tiny PDA screen (and let's face it, on most big screens too), scroll bars are a pain. We simplify things (and economise on space) using the following trickery:

1. When a polymorphic table is drawn, we count the number of lines. If there are more lines to be drawn than will fit on the screen, we record the offset of the first line *not drawn*.

2. We store this value on the MENUSTACK at byte offset +7 (the byte at +6 is zero). Otherwise, with normal completion of drawing of the table, the value at +7 is zero.

3. When the ROLLMENU instruction is encountered, we check the value of LOCALROLL. If the LOCALROLL value is zero, ROLLMENU is ignored and the next instruction is processed; otherwise the current menu is pushed to the stack and reloaded. When we push the current menu, we also push ROLLDEPTH and then increase the ROLLDEPTH value by LOCALROLL!

4. With polymorphic table creation, a nonzero value N in ROLLDEPTH signals that lines are to be skipped (not displayed) until line number N is to be drawn. Draw from there on, *and* record the last line number if there are still more undrawn.

5. We can repeat the process. The current limit is 128 lines. (Who wants to scroll through more than about 8–10 screens on a PDA looking for the correct option??)[27]

[27] There is also a constraint in that we only allow creation of MENUs sixteen deep, anyway!

5.5 Widget creation

We here use the term widget loosely to refer to any object contained within a menu, which usage is probably silly. Perhaps we should continue to abuse the term ‘control’.

5.5.1 CreateManyWidgets

We allow a menu to be included within a menu. this permits flexible addition of controls (grouped) within a menu. We use a flag (recur) to prevent deep recursion as this will screw PalmOS. The following code pulls out parameters for many widgets, and then creates them one by one.

```
Int16 sql::CreateManyWidgets(Int16 thismenu,
         Int16 x, Int16 y, Int16 w, Int16 h,
          Boolean recur)
{
  Int32 widgetcode=0;
  Int16 ok;
  double dh;
  double dw;
  double dy;
  double dx;

  QuickBy(iMARK); // must MARK stack..

  PushInteger(0); // allow while(widgetcode) to work without failure on PopInteger
  PushInteger(thismenu);
  Char * cmd;
  cmd = "&cSQ7";
  // "QMANY(SELECT MENUITEMS.miEnabled,MENUITEMS.miGroup,MENUITEMS.miX,
  // MENUITEMS.miY,MENUITEMS.miW,MENUITEMS.miH,MENUITEMS.miItem
  // FROM MENUITEMS WHERE MENUITEMS.miMenu = $[])"
  // [11-10-2007: now contains ORDER BY. See AnalgesiaDB2.tex]
  JustGo(cmd);  // OR INSERTANDGO


  ok = PopInteger(&widgetcode); // jvs 29-1-2006. This is *a* problem. fix me!
  Int32 i32;
  Int16 keeptop; // SECURITY: keep stack top before create widget!!
  Int16 newtop;  // check variable.

// A PROBLEM WITH THE FOLLOWING LINE IS THAT if an error puts
// ZERO ON STACK then WIDGET CREATION FAILS WITHOUT WARNING
// [explore me]

  while(widgetcode)  // ?? ugly. rather use QuickBy(iDEPTH) to get depth!
```

```
    {          Int16 wx;
               Int16 wy;
               Int16 ww;
               Int16 wh;
               Int16 igroup;
               Int16 able;

               PopFloat(&dh); // get *widget* screen-
               PopFloat(&dw); // coordinates
               PopFloat(&dy);
               PopFloat(&dx);
               PopInteger(&i32);
               igroup = (Int16) i32; // ugly.
               PopInteger(&i32);
               able = (Int16) i32; // likewise

               wx = x+PixelX(SCRIPTLIBCODE,&dx,w);
               wy = y+PixelY(SCRIPTLIBCODE,&dy, h);
               ww = PixelW(SCRIPTLIBCODE,&dw,w);
               wh = PixelH(SCRIPTLIBCODE,&dh,h);


               if (widgetcode != thismenu)
                  {

                    keeptop = *((Int16 *)(STACK+oTOP)); // in case
                    CreateOneWidget(widgetcode,
                      wx, wy, ww, wh,
                      recur, igroup, able); // might test ok
                    newtop =  *((Int16 *)(STACK+oTOP)); // check
                    if (newtop != keeptop)
                       { ERRmsg(ErWidgetMake);
                         WriteConsoleAsciiz(fDEBUG_AAGH, "?cw"); // create widget
                         WriteConsoleInteger(fDEBUG_AAGH, (newtop-keeptop));
                       };
                  };
               ok = PopInteger(&widgetcode);
    };
  QuickBy(iUNMARK);   // here we must UNMARK stack..
  return 1;
}
```

In the above wx, wy should be relative to the current menu (although because of our menu size coercion, this is at present irrelevant). There is a potential problem in the above if e.g. CreateOneWidget screws up our stack.[28] The ‘ if (widgetcode != thismenu)’ line is used to ignore our self-referential insertion. The stack

[28] Should probably check on fx outcome!

mark/unmark is inelegant but necessary.

5.5.2 CreateOneWidget

Given the unique ID of a widget in ITEM table (widgetcode), obtain information about it and then create the widget, inserting it into the current table. The contents of MYFORM may be changed by the called routine, as is standard for PalmOS. The function returns the ID of the new item if created, but just 1/0 for tables etc. [? fix me]; otherwise zero on failure.

First we obtain the widget details from the ITEM table, using an SQL query:

```
Int16 sql::CreateOneWidget(Int32 widgetcode,
         Int16 x, Int16 y, Int16 w, Int16 h,
         Boolean recur, Int16 grp, Int16 able)
{ Char * cmd;
  Char * iinitial;
  Int16 ok;
  Int32 xtracode = 0;

  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_WIDGET, "\x0A" "Widget code:"); //
  WriteConsoleInteger(fDEBUG_WIDGET, widgetcode);
  -OPTIONAL

  cmd = "&cSQ4";
  //  "QUERY(SELECT ITEM.iInitial,ITEM.iText,ITEM.iType
  //      FROM ITEM WHERE ITEM.iID = $[])";
  InsertAndGo (widgetcode, cmd);
```

Then extract the type of widget, and if it's a monomorphic table, discard iText items off the stack (leaving iInitial there!), extract the intialisation script, invoke the relevant routine (Section 5.6.1) and return:

```
  Int32 wtype;
  Int32 wnamelen;
  Int32 winilen;
  Char * Widgetname;
  PopInteger(&wtype);
  if (wtype == IMATABLE)  // 9 = monomorphic table
     {  QuickBy(iDISCARD); // discard iText
        iinitial = PeekAndPopStringZ(0x70); // iInitial
        if (! iinitial)
           { ERRmsg(10000); // [??? fix me]
             return 0;
           };
```

```
      ok = CreateOneTable(widgetcode,x,y,w,h, iinitial);
      Delete(iinitial); // clumsy
      return ok;
   };
```

Otherwise, get iText using the ugly routine PeekAndPopStringZ:

```
Widgetname = PeekAndPopStringZ(0x71);
if (! Widgetname)
   { ERRmsg(10000);
     return 0;
   };
```

Next, run the initialisation script (if it exists). The response to this script is inelegant, as for checkboxes and pushbuttons a numeric value (0 or non-zero) determines whether the control is on or off; for all other types, text is returns which replaces the associated text (buttons, labels, text fields, and poptriggers).

We reset the variable STOPPED at the start of this section, as scripts are also permitted to stop during widget initialisation!

```
winilen = PeekLengthPlus();
if (winilen < 3)  // too short?
   { QuickBy(iDISCARD); // junk script
   } else
   {
    Int16 keeptop;
    Int16 newtop;
    keeptop = -16 + *((Int16 *)(STACK+oTOP)); // in case
    // the -16 is to compensate for script on stack!

    isFAILURE = 0;
    JustGo("INTEGER(1)->MARK->RUN");
```

The above is a little silly as unbalanced MARK/other stack dis. might have catastrophic consequences. Far better to keep formal mark of stack, and 'manually' release it, with warning if stack imbalance.

In addition, the script which has just been run might conceivably 'enable' or 'disable' the control, but the control hasn't been created yet! We get around this by interrogating the ugly global ABLING, and then resetting it to -1![29]

The following was a stuffup as we used to use STOPPED for two separate things, one to STOP further creation of a widget, and another (later, more suspicious use) where we stop retrieval of a value for pushbuttons and checkboxes.

[29] See how JustGo (5.12.2) above Invokes ExeString with an initial parameter of zero, which when it finally trickles through to the Enabled routine (Section 5.16.2), signals that there is no associated widget yet created.

Hmm. We fix this by re-instituting the FAIL instruction, but limiting the damage it does to just one widget. FAIL forces termination of widget creation.

```
if (! isFAILURE)
   {
     if (ABLING != -1)
        { able = ABLING;
         //debug:// WriteConsoleErr(fDEBUG_AAGH, "abl",able);
          ABLING = -1;
        };
     if ( PeekDepth(SCRIPTLIBCODE, STACK) > 0) //result?
        { if ((wtype == PUSHBUTTONCTL)||(wtype == CHECKBOXCTL))
             { if (! STOPPED)
                  { PopInteger(&xtracode);
                  } else
                  { STOPPED = 0;
                  };
             } else
             { if (! STOPPED)
                  { Delete(Widgetname);
                    Widgetname  = PeekAndPopStringZ(0x72);
                  } else
                  { xCopy(Widgetname, "", 1); // ??hmm.
                    STOPPED = 0;
                  };
             };
        };
   };
     QuickBy(iUNMARK);
     newtop =  *((Int16 *)(STACK+oTOP)); // check
     if (keeptop != newtop)
        {
          ERRmsg(ErWidgetMake2);
          WriteConsoleAsciiz(fDEBUG_AAGH, "?cx"); // create widget error2
          WriteConsoleInteger(fDEBUG_AAGH, (newtop-keeptop));
         /// *((Int16 *)(STACK+oTOP))=keeptop; // force ???
        };
   if (isFAILURE)
      { isFAILURE = 0;
        Delete(Widgetname);
        return 0; // simply fail. ??
      };
   };
```

There is a little problem in the above. If the intialisation script tested SQLOK already, and then gracefully exited (but returned a default value) the second SQLOK

above will fail, and so the initialisation won't be performed! Overall, the SQLOK condition test seems attractive, but we need some way of 'normalising' things!

BTW, we have 'improved' the use of the STOPPED flag so that if we stop within an initialisation routine, we can detect this cessation and still *create* the widget but force it's value to 'uninitialised'.[30]

Next check for a menu, recursively invoke CreateManyWidgets, but reset the recursion flag to prevent infinite regress.

```
if (wtype == IMAMENU)   // type 20: included menu!
   { Delete (Widgetname);
     return CreateManyWidgets(widgetcode, x, y, w, h, 0);
   };
```

Make the widget according to the Menu codes (1–7) in Table 1, i.e. as specified in `palmsql3.h` (Section 7.2.2). The one tricky component left is the polymorphic table:

```
if (wtype ==POLYTABLE)   // 8 = polymorphic
   { Delete(Widgetname);  // [hmm?]
     return CreatePolyTable(widgetcode,x,y,w,h,grp);
   };
```

We pass the group (grp) as this signal is rather used to make text components within a polytable clickable.

Next check for a poptrigger, and if present, fix up the associated selection list using yet another SQL query! Things are made more complex because the list itself may be a script, signalled by `->` at the start of the retrieved 'list'. We mark and unmark to ensure stack integrity. Nasty coding.

```
Int16 ilines=1;  // default is 1
ActualList * alist;
alist = 0;

if (wtype == POPUPTRIGGERCTL)
   { wtype = LISTNOTCTL;
     alist = MakeActualList(widgetcode);
     wnamelen = w_StrLen(alist->auxlist);
     ilines = CountItems(NUMERICCODE, alist->auxlist, wnamelen, '|');
     ilines /= 2; // items are paired!
       // here might warn/correct if number is odd!
     if (ilines > 12) { ilines = 12; }; // limit size
   };
```

[30] The Perl implementation is similar. See that usage.

Finally, insert the widget and keep a record of it:

```
  Int32 uid;
  uid = InsertWidget (&MYFORM, (Int16) wtype,
        x, y, w, h, Widgetname, grp,  alist,
        ilines, (Int16) xtracode, able);
  if (uid)
     { PrependWidget(uid, (Int16)widgetcode, 0,
                    wtype);  // record
     };
  Delete(Widgetname);
  return uid;  // ok
}
```

5.5.3 Populating a poplist

An important function referenced by the above is one which sets up the list used within a poplist — MakeActualList.

```
ActualList * sql::MakeActualList(Int32 widgetcode)
{
  Char * cmd;
  Char * mylist;
  ActualList * alist;
  alist = 0;

  cmd = "INTEGER(1)->MARK->QUERY(SELECT ITEM.iList FROM ITEM WHERE ITEM.iID = $[])
        // [We must turn this into external SQL invocation]$

  InsertAndGo(widgetcode,cmd);
  mylist = PeekAndPopStringZ(0x73);
  if (xSame(mylist, "->", 2) )
     { PushString(mylist+2, (w_StrLen(mylist)-2));
       Delete(mylist);
       JustGo("RUN->LIST(|)");  // clumsy [check failure??]
       mylist = PeekAndPopStringZ(0x74);
     };
  JustGo("UNMARK");

       // 25-6-2006:
  if (mylist)
     { alist = (struct ActualList *) xNew (sizeof (struct ActualList), 0x50); // X
       alist->count = 0;
       alist->auxlist = mylist;
       alist->auxlen = w_StrLen(mylist);
     };
  return alist;
}
```

The caller of the above should now *never* delete mylist directly, but only when deleting the returned ActualList.

5.6 Monomorphic tables

Monomorphic tables are make up of just a single element type, e.g. a button, label or text field. Our plan of attack is:

1. Find single column (first) associated with the table in ICOLTABLE and thus get fractional width of an item and unique code of item;
2. What about number of rows?
3. Run SQL initialiser to get pairs on stack (mark stack first);
4. repetitively pull off pairs, create an item for each pair, draw it, inserting it into the menu, and *associate it with a unique value, the V-value*;
5. Repeat until end.

5.6.1 CreateOneTable

Initialise and then get column width and type of item to be duplicated:

```
Int16 sql::CreateOneTable(Int32 tablecode,
           Int16 x, Int16 y, Int16 w, Int16 h,
           Char * iini)
{ double d;
  Int16 iw;  // item width
  Int16 ih;  // item height
  Int32 dummy;
  Int32 copyitem=0;
  Int16 itype;
  Int16 lines;
  Char * cmd;
  Char * initem; // 2007-12-08

  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_STMT, "\x0A" "T:");
  -OPTIONAL

  cmd = "&cSQ5";
     // "QUERY(SELECT ICOLTABLE.irItem,ICOLTABLE.irFraction
     //        FROM ICOLTABLE
     //        WHERE ICOLTABLE.irTBL = $[]
     // AND ICOLTABLE.irOrder = 1)";
```

```
  InsertAndGo (tablecode, cmd);
  PopFloat(&d);
  iw = PixelW(SCRIPTLIBCODE,&d,w); // pixel width
  PopInteger(&copyitem); //

  cmd = "&cSQ6";
  //   "QUERY(SELECT ITEM.iType,ITEM.iInitial
  //    FROM ITEM WHERE ITEM.iID = $[])";
  // 2007-12-08: altered to pull out iInitial for item.
  InsertAndGo (copyitem, cmd);

  initem = PeekAndPopStringZ(0x79); // 2007-12-08
//  WriteConsoleText(fDEBUG_ALWAYS, "\nINI: ", 6);
//  WriteConsoleText(fDEBUG_ALWAYS, initem, w_StrLen(initem));
//  WriteConsoleText(fDEBUG_ALWAYS, "\nType: ", 7);

  PopInteger(&dummy);
  itype = (Int16) dummy;
//  WriteConsoleInteger(fDEBUG_ALWAYS, itype);
```

Then determine row height (in pixels)

```
  lines = GetNumLines(tablecode);
  ih    = GetRowHeight(lines, h);
```

We next mark the stack, run the initialisation script, and pull pairs off the stack until nothing is left. Topmost is always the displayed text string, deeper on our stack is the unique database code of the item. We will use stackpeek to get the length of the text item, set aside text for this, pull it in, and associate it (as a V value) with the created widget!!

```
  cmd = "INTEGER(0)->MARK";
  JustGo(cmd); // mark stack
  JustGo (iini); // RUN INI SCRIPT!
  //   For example:
  //   "QMANY(SELECT WARD.swrdID, WARD.swrdText
  //    FROM WARD WHERE WARD.swrdID <> 1)\";

  Char * vtext;
  Int16 vlen;
  Int32 Vid;
  Int16 xj, yj;
  xj = x;
  yj = y;
  Int32 uid;
  vlen = 128; // reasonable maximum
```

Okay, here goes . . .

```
  while (StackPeek(SCRIPTLIBCODE, STACK, STACKSTRING, 0, &vlen)
          > 0)
    { vlen += 0x10;  // a little more room
      vtext = xNew(vlen, 0); // clumsy [??] XNW12
      vlen = PopString(vtext, vlen);
      if (vlen > 0)
         {
           *(vtext+vlen) = 0x0; // asciiz
           // first, try to execute ini string:
           if (initem)
              { PushString (vtext, vlen); // push name
                JustGo("MARK(#1)");
                JustGo (initem); // RUN INI SCRIPT!
                // here we might get vtext string back!
                JustGo("UNMARK");
              };

           PopInteger(&dummy);
           Vid = dummy;
           uid = InsertWidget (&MYFORM, itype,
                 xj, yj, iw, ih, vtext, 0,  0, 1, 0, 1);
           if (uid)
              { PrependWidget(uid, (Int16)copyitem,
                Vid, itype);  // retain it
              };
         };
      Delete(vtext); // clumsy [atavism. fix me]
      // revise coordinates:
      yj += ih; // add height
      if (yj+ih > y+h+3) // if won't fit
         { yj = y;       // +3 is arbitrary!
           xj += iw;     // move 1 column right
           if (xj+iw > x+w+3) // arbitrary
              { // TOO MANY WIDGETS
              };
         };
      vlen = 128; // as above
    };
  cmd = "UNMARK";
  JustGo (cmd);

  if (initem) {Delete(initem); }; // 20071-12-08
  return 1;
}
```

In the case of a monomorphic table, the 'group' (fifth last argument of InsertWidget) is irrelevant. The final section (yj += and so on) revises the coordinates appropriately. At some stage we will need to sit down and work out fancy inter-

column spacing etc.

5.7 Polymorphic tables

These tables contain differing columns of elements. Each row is associated in some way with a unique key within the database.

Our approach is to:

1. Get the initialisation script and run it to get list of database keys, then creating an array of such ‘IDs’;
2. Get number of rows!
3. Get ’list’ of column items
4. For each *column* create an item for each row
5. Tediously run the relevant script *for each item*

Population of such a table is thus cumbersome but fairly flexible. Because we don’t constrain a row to map to a particular database row, substantial processing may be needed to populate each component of a row with data relevant to the unique key associated with that row! First, we obtain the unique IDs:

```
Int16 sql::CreatePolyTable(Int32 tablecode,
             Int16 x, Int16 y, Int16 w, Int16 h, Int16 grp)
{ Int32 dummy;
  Char * cmd;
  Int16 lines;

  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_STMT, "\x0A" "P:");
  -OPTIONAL

  y += 3; // [hack: examine and address me!]
  JustGo("INTEGER(0)->MARK");
  cmd = "&cSQ12";
    // "QUERY(SELECT ITEM.iInitial FROM ITEM
    //        WHERE ITEM.iID = $[])->RUN"
  SQL3UPTURN(SQL3CODE, 1); // reverse order!! [hack]
  InsertAndGo (tablecode, cmd);
  SQL3UPTURN(SQL3CODE, 0); // normal order!
```

We mark the stack to prevent overruns. In the above, the initialisation statement which we RUN will return IDs to process, possibly in a particular order. We here have an embarrassing problem — the Perl program will process things

in the correct order, while the C++ program will use the stack for processing, and do things in reverse order. We use the convenient internal 'hack' of setting the UPTURN flag (see *Sql3lib.tex*) rather than reversing the stack elements once retrieved.[31] The alternative would be to coerce the Perl program into reversing things, which would make *everything* the wrong way around!

Next we use DEPTH to determine the number of items on the stack. We should rewrite the following clumsy code which determines the number of items on the stack, that is, the number of rows retrieved from the database.

```
cmd = "DEPTH";
JustGo(cmd);
PopInteger(&dummy);
Int16 idcount;
idcount = (Int16) dummy;
```

We pull out each code, storing it in an array:

```
Char * IDarray=0;  // keep compiler happy
Int16 i=0;
Int16 rd = ROLLDEPTH;

if ((idcount-rd) > 0)
   { IDarray = xNew( 4*(idcount-rd), 0xE6); // Int32s thus 4; XNW13
     while (i < idcount)
       { PopInteger(&dummy);
         if (rd > 0)
            { idcount --; // one item less
              rd --;
            } else
            { *((Int32 *)(IDarray+(4*i))) = dummy;
              i ++;
            };
       };
   } else // safety precaution:
   { idcount = 0;
   };
```

See how the first item in IDarray is the 'topmost' item on the stack — but we also need to account for the line offset specified in ROLLDEPTH, so we clip this number of items off the top of the stack *before* we start putting items into IDarray.

Then obtain row height, submit more SQL to 'list' column items with their titles and fractional width:

[31] Which takes more time and effort; note that the order of elements on the STACKSTRING must also be reversed, although this is a bit of a straw man as we would almost certainly be dealing with primary keys which fit on the STACK alone!

```
Int16 ih;
lines = GetNumLines(tablecode);
ih    = GetRowHeight(lines, h);
Int16 xj, yj;
xj = x;
cmd= "&cSQ11";
// cmd = "QMANY(SELECT ICOLTABLE.irItem,ICOLTABLE.irName,
//       ICOLTABLE.irEnabled,ICOLTABLE.irFraction
//       FROM ICOLTABLE WHERE ICOLTABLE.irTBL = $[])";
InsertAndGo (tablecode, cmd);
```

Next, record number of lines shown *if* there are more rows to display than will fit in the display area.

```
if (idcount > (lines-1))
   { LOCALROLL = (lines-1);
     LINESLEFT = idcount - LOCALROLL;
   };
```

Then for each column find fractional width and insert column header widget, followed by creation of all items in the column.

```
double d;
Int16 iw;
Char * cname;
Int32 uid;
Int16 itmid;
Int16 nabled;

while (PopFloat(&d)) // while more columns
  { yj = y;
    iw = PixelW(SCRIPTLIBCODE,&d,w);
    if (! PopInteger(&dummy)) // get irEnabled
       { ERRmsg(ErBadColumnItemCode);
         return 0;
       };
    nabled = (Int16) dummy;

    cname = PeekAndPopStringZ(0x75); //irName label on top
    if (! cname)
       { ERRmsg(ErBadColumnName2);
         return 0;
       };
    if (! PopInteger(&dummy)) // irItem
       { Delete(cname);
         ERRmsg(ErBadColumnItemCode);
         return 0;
       };
```

```
        itmid = (Int16) dummy;
        uid = InsertWidget (&MYFORM, LABELNOTCTL,
                xj, yj, iw, ih, cname, 0,  0, 1, 0, 1);
        if (uid)
           { PrependWidget(uid, itmid, 0, LABELNOTCTL);
           };
        Delete(cname);

        if (idcount) // if any items
           { MakeWholeColumn(itmid, IDarray, idcount,
                 xj, yj, iw-3, ih, nabled, lines-1, grp); // [-3 is hack]
           }; // might here ELSE [message]
        xj += iw; // move to next column
      };  // END WHILE POPFLOAT
      // if idcount = 0 we might here put default text [fix me!]
  if (idcount)
     { Delete(IDarray);
     };
  cmd = "UNMARK";
  JustGo(cmd);
  return 1;
}
```

If nothing was inserted, then we should (but don't at present) insert the default text row for the whole table, (along the lines of 'This page is blank' :-) The column creation routine is examined in the next section.

5.7.1 MakeWholeColumn

First, find the type of item filling the column, and process the special case where we're dealing with a poplist:[32] The `mlines` parameter is the maximum number of lines to draw while idcount is the total number of rows fetched from the database.

```
Int16 sql::MakeWholeColumn(Int16 itmid, Char * IDarray,
             Int16 idcount,
             Int16 xj, Int16 yj, Int16 iw, Int16 ih,
             Int16 nabled, Int16 mlines,
             Int16 grp)
{ Char * cname = 0;
  Char * cmd;
  Int32 dummy;
  Int16 ok; // debugging only
  Int16 stktop; // nasty hack

  Int16 itemtype;
```

[32] Now you can see the enormous advantage of processing column by column!

```
cmd = "&cSQ6";
  // as before:
 // "QUERY(SELECT ITEM.iType,ITEM.iInitial FROM ITEM
 //          WHERE ITEM.iID = $[])";//
InsertAndGo (itmid, cmd);
QuickBy(iDISCARD); // nasty: (for now) discard iInitial
PopInteger(&dummy);
itemtype = (Int16) dummy;
dummy = 0; // fix 31-5-2006 for bad programming!
```

The last line above illustrates how bad it is to conveniently hijack a variable for a clumsy purpose. Without resetting dummy to zero, we ended up with buttons with a dark background.[33]

```
Int16 ilines = 1;
ActualList * alist;
alist = 0;
Int32 wnamelen;

if (itemtype == POPUPTRIGGERCTL)
   { itemtype = LISTNOTCTL;
     alist = MakeActualList(itmid); // hmm [does duplication occur???][check me]
     wnamelen = w_StrLen(alist->auxlist);
     ilines = CountItems(NUMERICCODE, alist->auxlist, wnamelen, '|');
     ilines /= 2; // items are paired!
       // here might warn/correct if number is odd!
     if (ilines > 12) { ilines = 12; }; // limit size
   };

Int16 i; // loop counter
Int32 V; // V value of item
i = 0;

if (idcount > mlines)
   { idcount = mlines;
   };
```

In the above we check that idcount is less than mlines. If not, we use the smaller value as we cannot display all of the lines on this screen.

Next, create a column of items. Here is where the deficiencies of PalmOS in handling copies of a poplist really come to the fore!

[33] We can also see the wisdom of exploring errors — this error provides us with the idea that we might change the background of the button thus in order to signal ‘a recently visited button’. Something to pursue at a later date! The actual error occurred related to a *second* error where we accidentally invoked CtlSetValue!

```
while (i < idcount)
  { yj += ih; // move to next line
    V = *((Int32 *)(IDarray+(4*i))); //

    stktop = -(*((Int16*)(STACK+oTOP)));  // baseline value

    cmd = "&cSQ8";
    // "QUERY(SELECT ITEM.iInitial FROM ITEM
    //        WHERE ITEM.iID = $[])->RUN";

    ok = InsertAndGoV(itmid, cmd, -(V)); // Do SQL: see VValue to explain -ve
    +OPTIONAL
    WriteConsoleAsciiz(fDEBUG_SQL, "Col=");
    WriteConsoleInteger(fDEBUG_SQL, V);
    WriteConsoleAsciiz(fDEBUG_WIDGET, ":");
    WriteConsoleInteger(fDEBUG_SQL, ok);
    -OPTIONAL

    if (! isFAILURE) // if 'creation succeeded'
       { stktop += *((Int16*)(STACK+oTOP));  // No of items
         if (stktop) // if something was fetched..
                     // CHECK for STACK ITEM
           {if ((itemtype == PUSHBUTTONCTL)||(itemtype == CHECKBOXCTL))
               { PopInteger(&dummy);
               } else
               { cname = PeekAndPopStringZ(0x76); // retrieve string value!
                 // WriteConsoleInteger(fDEBUG_WIDGET, stktop); // hmm should /8
                 // [we might even hiccough if value != 8]
                 +OPTIONAL
                 WriteConsoleText(fDEBUG_WIDGET, cname, w_StrLen(cname));
                 -OPTIONAL
               };
           } else
           { cname = 0; // hmm.
             WriteConsoleText(fDEBUG_WIDGET, " NULL", 5); // ???
           };
           Int32 uid;
           uid = InsertWidget (&MYFORM, itemtype,
                 xj, yj, iw, ih-1, cname, grp,
                 alist, ilines, (Int16)dummy, nabled);
             // pass extra parameter (state of pushbutton or checkbox)
             // as 2nd last argument of InsertWidget
             // grp is used to signal clickability of text [ugh]
           if (uid)
              { PrependWidget(uid, itmid, V, itemtype);
              };
           if (cname) {Delete(cname); };
       } else
```

```
          { isFAILURE = 0;
            +OPTIONAL
            WriteConsoleAsciiz(fDEBUG_SQL, "[fail]");
            -OPTIONAL
          };
       i ++;
     }; // end while.
  return 1;
}
```

The invocation of InsertWidget with `ih` diminished by 1 is because PalmOS seems not to take 'widget' borders into account when using dimensions supplied.

We remember to delete `cname`.

5.7.2 GetRowHeight & GetNumLines

A simple SQL query routine.

```
Int16 sql::GetRowHeight(Int16 lines, Int16 h)
{ if (lines < 1) { return 15; };  // hack
  h /= lines;
  if (h < 10) { return 10; }; // another hack
  return h; // [clumsy fix me, perhaps use float]
}
```

Formerly we had a more complex floating point routine. Here's a related routine which simply queries the database to obtain the number of lines associated with the item (usually a mono- or polymorphic table).

```
Int16 sql::GetNumLines(Int32 tablecode)
{ Char * cmd;
  Int32 dummy;
  cmd = "&cSQ9";
   // "QUERY(SELECT ITEM.iLines FROM ITEM
   //    WHERE ITEM.iID = $[])";
  InsertAndGo (tablecode, cmd);
  if (! PopInteger(&dummy)) // ilines
     { return 1; // hmm [should we write error?]
     };
  return (Int16) dummy;
}
```

5.8 Response handling

Button response handling, and so forth, as invoked in the main section (See the several invocations in Section 6.3). We pass the unique code of the widget se-

lected, as well as the nature of the event. The response differs depending on the nature of the control.

```
Int16 sql::Respond(Int16 item, Int16 myint, Int16 whatctl,
                   EventPtr e)
{ if (! SCRIPTLIBCODE)
     { return 0;
     };
  +OPTIONAL
//  WriteConsoleAsciiz(fDEBUG_STMT, "\x0A");
//  WriteConsoleErr(fDEBUG_STMT, "Clk",item); // debug+
  -OPTIONAL
///  WriteConsoleText(fDEBUG_STMT, "!", 1); // jvs 20080515
```

If we're dealing with a list, we first get the string of the item clicked on, pushing it to the stack. The item ID passed is that of the *poptrigger* — we need the trigger ID because we are responding to selection in a list, but the response script is associated with the poptrigger, not the list.

```
 if (whatctl == LISTNOTCTL)
     {  Char * showme;
        // ListType * listP;
        Int16 cnt;
        Int16 slen;
        Int16 ok;

        cnt = e->data.popSelect.selection; // which item?
        // 25-6-2006: here must fix things up
        // so that we select id from pair!
        showme = FetchListItem(item, cnt, &slen);
        PushString(showme, slen);
        ok = QuickBy(iINTEGER); // convert string to integer!
        if (ok < 1)
          { ERRmsg(ErBadListInt); // safety [to FIX PERL]
            return 0;
          };
     };
```

For checkboxes or pushbuttons, push one or zero to the stack depending on the state (activated or not) of the control.

```
  if ((whatctl == CHECKBOXCTL) || (whatctl == PUSHBUTTONCTL))
     {
       PushInteger(myint);   // a 1 or a 0
     }
```

We specifically look for the QUITBUTTON or OKBUTTON codes, as these signal click on something in ASKFORM (See MakeTextForm). If so, respond appropriately in the required, convoluted fashion.

```
if ((item == QUITBUTTON) || (item == OKBUTTON))
   { return RespondToAsk(item);
   };
```

One other smart test is whether we clicked on the header of a menu itself:

```
Int16 ans;
if (whatctl == MENUITSELF)
   { ans = CURRENTMENU;
   } else
```

Otherwise, get the database ID of the widget, and find response script.[34]

```
   { ans = FindWidgetDatabaseCode (item);
   };

if (! ans)
   { ERRmsg(ErWidgetNoDbCode);
     return 0; // fail (not found)
   };
Char * cmd;
cmd = "&cSQ10";
//  "QUERY(SELECT ITEM.iResponse FROM ITEM
//    WHERE ITEM.iID = $[])";
InsertAndGo (ans, cmd);
```

Finally execute the response script ...

```
Int16 ok;
Char * reString;
Int16 relen;

relen = 1024;  // reasonable maximum length! (modified 200805~)
if (StackPeek(SCRIPTLIBCODE, STACK,
              STACKSTRING, 0, &relen) < 0)
   { return 0;  //fail
   };
reString = xNew(relen+0x10, 0); // XNW14
  // MUST augment: eg. floats and timestamps!
relen = ResolveX(reString, relen);
if (relen < 1)
```

[34] Later on we might simply point to the actual script, not bothering with the SQL query. [fix me]

```
      { Delete(reString);
        return 0;
      };
   ok = ExeString(item, reString, relen);
   Delete(reString);
   return ok;
}
```

The ExeString line is clumsy. It would be better to retrieve a direct pointer to the database string (and record its length), and push a reference to this.

There was a significant difference between responses to a press on a push-button here and in the Perl program. In the Perl, the default was not to permit repeated push on a button already on, but here we freely allow this. This is now fixed in the Perl version.

5.9 Some utility routines

5.9.1 PushString

Push a string. With the following there is great potential for optimisation, including finding the SQL script in the file and interpreting it there, rather than laboriously copying it over into temporary memory space with the SQL query.

```
Int16 sql::PushString(Char * str, Int16 slen)
{
  return ( PushItem (SCRIPTLIBCODE, STACK,
                  STACKSTRING, str, slen, 'V', 0)
         );
}
```

5.9.2 PushInteger

Simply push an integer onto the stack.

```
Int16 sql::PushInteger(Int32 i)
{ if (! PushItem (SCRIPTLIBCODE, STACK,
                  STACKSTRING, (Char *)&i, 4, 'I', 0))
     { return 0;
     };
  return 1;
}
```

5.9.3 PopFloat

Try to pop a float off the stack. If there's no float, fail, otherwise pop it, populating a by reference float.

```
Int16 sql::PopFloat(double * DP)
{ Int16 poplen = 8;
  Int16 ok;

  ok = PeekType();
  if (ok != 'F')
     { if ((ok < 0) && (ok != -ScErPopStackEmpty))
          { ERRmsg(ErFloatPop);
          };
       return 0;
     };
 PopItem(SCRIPTLIBCODE, STACK,
         STACKSTRING, (Char*) DP, &poplen);
  return 1;
}
```

5.9.4 PopInteger

Similar to PopFloat, the wrinkle being that if the datum type is a fixed length numeric, it's converted to an integer [check this]!

```
// PopInteger: pop stack into &i, note byref invocation.

Int16 sql::PopInteger(Int32 * IP)
{ Int16 poplen = 4;
  Int16 ok;

  ok = PeekType(); // clumsy.
  if (ok != 'I')
     { if (ok == 'N') // fixed length numeric
          { ok = QuickBy(iINTEGER);
            if (ok < 0)
               { ERRmsg(ErPopIntConvertFail);
                 QuickBy(iDISCARD); // waste corresp. item
                 return 0;
               }; // else, continue to read integer!!
          } else  // if NOT 'N', just fail:
          { //if ((ok < 0) && (ok != -ScErPopStackEmpty))
            //   {
                 ERRmsg(ErIntegerPop); // hmm? check me???
                 // jvs: 29-1-2006: there is a problem here. Look at invokers
            //   };
            QuickBy(iDISCARD); // waste corresp. item
```

```
            return 0;
          };
      };
  // type IS 'I':
  if (PopItem(SCRIPTLIBCODE, STACK,
              STACKSTRING, (Char*) IP, &poplen) < 0)
      { ERRmsg(ErIntegerPop);
        QuickBy(iDISCARD); // waste corresp. item
        return 0;
      };
  return 1;
}
```

5.9.5 PopString

Pop a text string.

```
Int16 sql::PopString(Char * P, Int16 plen)
{ plen = ResolveX(P, plen);
  if (plen < 0)
     { ERRmsg(ErStringPop);
       return 0;
    };
  return plen;
}
```

5.9.6 PeekAndPopStringZ

A nasty routine to not only pop a string, but make the returned string ASCIIZ.

```
Char * sql::PeekAndPopStringZ(Int16 e)
{ Int16 ppl; // e is for memory tracing
  Char * P;
  ppl = PeekLengthPlus();
  if (!ppl)
     { ERRmsg(ErFailPopStringLen);
       return 0; // ?stack
     };
  ppl --;
  P = xNew(ppl+0x10, e);  // allow for dates etc. XNW15
  if (ppl) // if not null string
     { ppl = PopString(P, ppl+0x10);
     } else
     { QuickBy(iDISCARD);
     };
  *(P+ppl) = 0x0;  // asciiz
  return P;
}
```

5.9.7 ResolveX

Invoke the library ‘Resolve’ function to convert a stack item.

```
Int16 sql::ResolveX(Char * P, Int16 plen)
{
  Int16 ok;
   ok = Resolve(SCRIPTLIBCODE, P, plen,
                STACK, STACKSTRING);
 +OPTIONAL
 if (ok < 0)
    { WriteConsoleAsciiz(  fDEBUG_AAGH, "(Res?" ); //
      WriteConsoleInteger( fDEBUG_AAGH, plen    );
      WriteConsoleAsciiz(  fDEBUG_AAGH, ","     ); //
      WriteConsoleInteger( fDEBUG_AAGH, ok      );
      WriteConsoleAsciiz(  fDEBUG_AAGH, ")"     ); //
    };
 -OPTIONAL
 return ok;
}
```

5.9.8 FindDelimiter

This routine is similar to Advance, but specifically looks for an *unquoted* `->`, in other words, one which isn’t within quoted text or parenthesis. We achieve this by not searching once a quote has been encountered, until a second quote is met. Likewise for parenthesis.

Submitted arguments are `P` the string pointer, and `d` its length.

```
Int16 sql::FindDelimiter(Char * P, Int16 d)
{ Char c;
  Boolean quoting=0;
  Boolean pars=0;
  Int16 kept = d;
  while (d > 1) // until 2nd last char
    { c = *P++;
      d--;
      if (pars) // awaiting parenthesis closure
         { if (c == ')')
              { pars = 0;
              };
         } else
         { if (quoting) // await closing quote
              { if (c == '"')
                   { quoting = 0;
                   };
              } else
```

```
              { if (  (c == '-')
                    &&(*P == '>')
                   )
                   { return (kept + 1 - d); // after ->
                   } else
                   { if (c == '"') // opening
                        { quoting = 1;
                        } else
                        { if (c == '(') // opening
                             { pars = 1;
    };   };   };   };   };   };
  return 0; // fail
}
```

5.9.9 FindDelimiter2

This routine is similar to FindDelimiter, which it invokes. If the delimiter is present, we move to the next instruction, tracking the length, otherwise fail.

```
Char * sql::FindDelimiter2( Int16 * thislen,
             Int16 * slLEN, Char * scriptline)
{ Int16 tl;

  tl = FindDelimiter(scriptline, *slLEN);
  if (tl > 0)   // if delimiter (->) present
     { scriptline += tl; // move
       * slLEN    -= tl; // track.
       tl -= 2; // compensate for -> at end
     } else     // fail
     { tl = * slLEN;
       * slLEN = 0;
     };
  * thislen = tl;
  return scriptline;
}
```

5.10 Script execution — long!

ExeString is a wrapper for the enormous ExeScript, which we've broken into bite-sized chunks to facilitate digestion.

```
Int16 sql::ExeString(Int32 xitem, Char * str, Int16 slen)
{
  // +OPTIONAL
  // WriteConsoleAsciiz(fDEBUG_STMT, "\x0A");
  // WriteConsoleErr(fDEBUG_STMT, "x",xitem); // debug+
  // -OPTIONAL
```

```
  *((Char **)(FxSTACK+topFxSTACK)) = str;    // push script
  *((Int16 *)(FxSTACK+topFxSTACK+4)) = slen; // length too
  topFxSTACK += 8;                           //
  return ExeScript(xitem);
}
```

ExeScript begins by defining a lot (perhaps an unnecessary lot) of local variables:

```
Int16 sql::ExeScript(Int32 xitem)
{
  Int32 iresult;
  Int16 poplen;
  Int16 txtlen;
  Int16 slLEN;
  Int16  thislen=0;
  Int16 outcome = iRETURN; // to start, simply pop stack!
  Int16 skip=0;
  Int16 continu = 1;
  Char * msgbuf;
  Char * txtbuf;
  Char * thiscommand=0;
  Char * scriptline=0;
  Char c;
//   Int32 ninetynine = 99; // a clumsy hack

  Int16 helpok;  // used in caching and uncaching.
  Char * cachez;

///  WriteConsoleText(fDEBUG_STMT, "*", 1); // jvs 20080515

  STOPPED = 0;
```

The central while loops are a little tricky. The outer while terminates if `continu` is zero, the inner finds a script component and then executes it. A trick is that if DoScript signals that a numeric routine is needed, then we invoke a separate numeric library function using DoNumeric.[35]

```
  while (continu)
    { while (  (slLEN > 0)
             &&(outcome == NORMALOUTCOME)
            )
```

[35] This is clunky and should be replaced by passing the handle of the numeric library to ScriptinLib, and calling the numeric library from there!

```
{
  c = * scriptline;
  if (c == '@')
     { if (slLEN < 7)
          { ERRmsg(ErShortBytecode);
          };
       slLEN -= 7;
       scriptline += 7; // skip past
     };
```

At present if the above test for the character '@' succeeds, we simply ignore the following six characters (or fail if there are less than six).[36] Ultimately we will use `@nnnn` to signal the use of an optimised bytecode script. We continue by finding the `->` delimiter, and interpret the first instruction using DoScript:

```
  thiscommand = scriptline;
  scriptline = FindDelimiter2(&thislen, &slLEN, scriptline);
  if (! skip)
     { outcome = DoScript(SCRIPTLIBCODE, STACK,
                  STACKSTRING, thiscommand, thislen);
       if (outcome > iA_BASELINE)  // numeric
          { outcome = DoNumeric(NUMERICCODE,
                       outcome, STACK, STACKSTRING);
            if (outcome < 0)
               { ERRmsg( -(outcome) );
               };
            outcome = iNORMAL; // recover!
          };
     } else
     { skip = 0;
     };
};
continu = 0; // force exit by default
```

The value of `outcome` is crucial. Under many circumstances, the library routines cannot complete execution of the whole instruction, and need the caller to do something. This 'something' is signalled in the value of `outcome`. In the following sections we examine the consequences of returning such values.

```
Int16 lp;

switch(outcome) // clumsy handler
  {
    case iSKIP:
```

[36] Such a sequence cannot at present occur at the end of a string.

```
skip = 1; // force skip next (clumsy)
continu = 1;
break;
```

The signal iSKIP forces skipping of the next statement; iREPEAT results in repeated execution of the same statement until a STOP signal is encountered. [DRESS UP THE FOLLOWING]

```
case iREPEAT:
  // 1. move scriptline back to before the repeat fx:
  if (! slLEN)  // cf. above: is delimiter present?
     { slLEN = thislen;
       // scriptline wasn't incremented, so we can leave it *as is*!
     } else
     { slLEN += 2+thislen;             // compensate for the ->
       scriptline -= 2+thislen;        // move back in scriptline
     };
  if (! PushCurrentScript (scriptline, slLEN))
     { return 0;                          // fail
     };
  thiscommand += 8;  // length of "REPEAT(&" is 8
  thislen -= 9;      // also ignore the final right parenthesis
  scriptline = FindFunction(thiscommand, thislen, &slLEN);
  if (! scriptline)
     { return 0;
     };
  continu = 1;
break;
```

iFUNCTION assumes success, pushes the current function pointer, and then tries to invoke the relevant function.

```
case iFUNCTION:
if (! PushCurrentScript (scriptline, slLEN))
   { return 0; // fail
   };
  lp = Advance(thiscommand,thislen,'(');
  if (lp > 0) { thislen=lp-1; }; // no paren.
  thiscommand ++; // go past the '&'
  thislen --;     // track
  scriptline = FindFunction(thiscommand,
                            thislen, &slLEN);
  if (! scriptline)
     { return 0;
     };
  continu = 1;  // continue
break;
```

eFUNCTION is similar, but we need to replace the current function with the invoked one, rather than simply calling! In fact what we do is simply fail to push the current script![37]

```
          case eFUNCTION:
            lp = Advance(thiscommand,thislen,'(');
            if (lp > 0) { thislen=lp-1; }; // no paren.
            thiscommand ++; // go past the '='
            thislen --;     // track
            scriptline = FindFunction(thiscommand,
                                      thislen, &slLEN);
            if (! scriptline)
               { return 0;
               };
            continu = 1;  // continue
          break;
```

iRETURN just returns.

```
          case iRETURN:
            scriptline = DoReturn(&slLEN, 0);
            if (! scriptline)
               {
//               failure:
                 topFxSTACK = 0; // clean FUNCTION stack
                 ScriptIni(SCRIPTLIBCODE, STACK, STACKSTRING); // clear data
                 return 0;
               };
          continu = 1;
          break;
```

The above failure routine is important: if we RETURN from the topmost function in a sequence of nested functions, it ensures the stacks are clean and returns ('fails').[38]

iFAIL used to be more dramatic in its consequences, forcing clearing of function and data stacks, but now all it does (after multiple rewrites) is to allow us to stop creation of a particular widget dead in its tracks!

```
          case iFAIL:
            isFAILURE = 1; // force failure
///            WriteConsoleText(fDEBUG_STMT, "\nFAIL:", 6); // jvs 20080515
```

[37] The advantages of this convention are many, including the ability to have an if...then...else-like construct without a similar propensity to error and inefficiency, and the capacity to define a function like factorial with deep recursion but no stack overflow.

[38] Leaving this check out causes a major PDA error, but mainly related to the failure to return 0. The other stuff is redundant [check me]!

```
            while (DoReturn(&slLEN, 0))
                  { // do nothing but trim back until nil more.
///                    WriteConsoleText(fDEBUG_STMT, "+", 1);
                  };
            return (IMWAITING); // 20080515 'fail' but return as if 'acceptable'.
            // we give IMWAITING to abort deferred click on button. See PainHandle

        //    if (! scriptline)
        //       { goto failure; // ??? [check me!]
        //       };
        // NEED TO LOOK AT WHAT HAPPENS [FIX ME]
//            continu = 1; // continue
//          break;
```

iSTOP has several uses, the most important being termination of a REPEAT statement. The stop isn't propagated backwards beyond the level of the &function called by repeat. STOP is also used to signal 'don't include a submenu'.

There are two distinct uses of STOP at present:

1. To force termination of a REPEAT statement;

2. To terminate creation of a widget.

Other uses may ultimately be forthcoming. The current usage of the global (to sql) STOPPED variable is excessively ugly. It only really is kept at one to terminate widget creation, as with the REPEAT, it immediately becomes zero again. Nasty.[39] URZN also forces an iSTOPPED condition, terminating a REPEAT.

```
          case iSTOP:
            STOPPED = 1;
            scriptline = DoReturn(&slLEN, 1);
            if (! scriptline) { return 0; };
            if (xSame (scriptline, "REPEAT", 6))
               { STOPPED = 0;
                 thislen = Advance(scriptline, slLEN, ')' );
                 slLEN -= thislen;  // might be zero
                 if (slLEN) // if more stmts..
                    { scriptline += thislen+2; // ->
                      slLEN -= 2; // track
                    };
               };
            continu = 1;
          break;
```

Invoking X merely pushes the XPARAM to our data stack.

[39] Fix me!

```
case iX:
  if  (! PushInteger(XPARAM))
    { ERRmsg(ErNoXPush);
      return 0;
    };
  continu = 1;
  break;
```

Next, look at creation of a local variable name:

```
case iNAME:
outcome = MakeVariable (SCRIPTLIBCODE,
           STACK, STACKSTRING, LOCALS);
if (outcome < 0)
   { ERRmsg(ErLocalVariableName);
     return 0;
   };
continu = 1;
break;
```

We set the value of a variable by invoking SetVariable.

```
case iSET:
  outcome = SetVariable (SCRIPTLIBCODE,
      STACK, STACKSTRING, LOCALS);
if (outcome < 0)
   { ERRmsg(ErLocalVariableSet);
     ERRmsg(outcome);
     return 0;
   };
continu = 1;
break;
```

Given the variable name, we get the associated value, as instructed by iVARNAME. In the following we add 1 to `thiscommand` to move past the $ sign.

```
case iVARNAME:
outcome = GetVariable (SCRIPTLIBCODE, thiscommand+1,
   thislen-1,LOCALS, STACK, STACKSTRING);
if (outcome < 0)
   { ERRmsg(ErGetLocalVariable);
     ERRSTRING(thiscommand+1, thislen-1); // [??]
     return 0;
   };
continu = 1;
break;
```

CheckVariable in the following simply checks whether the variable exists, returning 1 or zero *on the stack*.

```
case iISNAME:
  outcome = CheckVariable (SCRIPTLIBCODE,
                           LOCALS, STACK, STACKSTRING);
  if (outcome < 0)
     { ERRmsg(10005);
       return 0;
     };
  continu = 1;
  break;
```

iKEY results in a key being generated from SQL. Rather than implementing some arbitrary (proprietary) extension of SQL, we use scripting to perform this generation.[40] The script we run is called NEWKEY.

```
case iKEY:
    if (! PushCurrentScript (scriptline, slLEN))
       { return 0;  // fail
       };
    scriptline = FindFunction("NEWKEY", 6, &slLEN);
      if (! scriptline)
         { ERRmsg(ErFailFindFx);
           return 0;
         };
      continu = 1;
      break;
```

For the sake of completeness, here's the script:

```
COPY->
QUERY(SELECT UIDS.u$[] FROM UIDS
  WHERE UIDS.uKey = 1)->
COPY->BURY->INTEGER(1)->ADD->
DOSQL(UPDATE UIDS SET UIDS.u$[] = $[]
  WHERE UIDS.uKey = 1)->
DIGUP
```

Next, we pop up a query box:

```
      case iQUERY:
        if (ExeQuery(1))
           { continu = 1;
           };
      break;
```

[40] As the script interrogates and updates an SQL table, we still need to cater for the need for atomic keys when/if we implement this in a multi-user environment!

SQL execution at present is limited to table creation, updating or insertion. We still need to add deletion, drop table, schemas and security; we have no immediate plans for PalmOS implementation of alter table, create assertion, create character set/collations, translations, triggers, creation of procedures/functions/methods, return, PSMs and UDTs, concurrency, and SQL/CLI functions; perhaps views.

```
case iDOSQL:
  if (ExeSQL())
     { continu = 1;
     };
break;
```

'ME' identifies the current user.

```
case iME:
  PushInteger(CURRENTUSER);
  continu =1;
break;
```

Associated is SETME, which sets the value in CURRENTUSER:

```
case iSETME:
  if (! PopInteger(&iresult)) // ilines
     { ERRmsg(ErBadUser);
       return 0; // fail
     };
  CURRENTUSER = iresult;
  continu = 1;
  break;
```

QMANY retrieves multiple rows onto the stack.

```
case iSQLMANY:
  if (ExeQuery(0))
     { continu = 1;
     };
break;
```

Run executes a script string, but forbids recursion.

```
case iRUN:
   if (RERUN) // no recursion
      { ERRmsg(ErRunRecursion);
        return 0;
      };
   RUNLEVEL = topFxSTACK;
   if (! PushCurrentScript (scriptline, slLEN))
```

```
      { ERRmsg(ErRunPushScript);
        return 0; // fail
      };
   slLEN = 0x10+PeekLengthPlus(); //
   RERUN = xNew(slLEN, 0xE1); // run buffer; XNW16
   slLEN = ResolveX(RERUN, slLEN);
   if (slLEN < MINLENRUNSTMT)
      { return 0; // [error msg??]
      };
   scriptline = RERUN;
   continu = 1; // simply continue!
break;
```

Should we add a few bytes to slLEN? The ResolveX actually gets the run string from the stack. See iRETURN for what happens when RUN finishes.

In the following SetX is the normal, civilized method of altering X on entering the next menu. SetZ is another stub for a function of dubious utility.

```
case iSETX:
  outcome = PopInteger(&iresult);
  if (! outcome)
     { ERRmsg(ErPopSetX);
       return 0;
     };
  NEWPARAM = iresult;
  continu = 1;
break;

case iSETZ:
  ERRmsg(10009);
break;
```

V returns the V value for the given item; it must be an integer.

```
case iV:
   iresult = VValue(xitem);
   if (! iresult)
      { ERRmsg(ErNoV);
        return 0; // fail
      };
   PushInteger(iresult);
   continu = 1;
break;
```

Popmenu is a rather clumsy but useful functionality: it pops a menu from under the current one. We do seem to need it.

```
case iPOPMENU:
  if ( !PopMenu() )
     { ERRmsg(10011);
       return 0;
     };
  continu = 1;
  break;
```

Pushmenu simply pushes the current menu.

```
case iPUSHMENU:
  if ( !PushMenu() )
     { return 0;
     };
  continu = 1;
  break;
```

Enabling or disabling of a widget:

```
case iENABLED:
   Enabled((Int16)xitem, 1); // [??? fix me]
  continu = 1;
break;

case iDISABLED:
  Enabled((Int16)xitem, 0);
  continu = 1;
break;
```

Ink should set the text colour but this function is at present rudimentary on the Palm [FIX ME]. Likewise for Paper, which should alter the background colour.

```
case iINK:
  continu = SetPaperOrInk((Int16)xitem, 1); //?
break;

case iPAPER: // PAPER : similar to ^ink
  continu = SetPaperOrInk((Int16)xitem, 0); //?
break;
```

We still must fully implement SQL COMMIT/ROLLBACK. On COMMIT, we will simply copy *all* SQL databases to backup, thereby overwriting the previous set, and setting things up for the next rollback.

```
case iCOMMIT:
  //  ERRDisplay("+Warn: COMMIT",0);
  continu = 1; // hack
```

```
        break;

        case iROLLBACK:
          // ERRDisplay("+Warn: ROLLBACK",0);
          continu = 1; // hack
        break;
```

QOK merely tests the SQLOK flag, pushing the 1/0 to the stack and thus signalling whether the last SQL statement returned any data or not. Useful!

```
        case iOKSQL:
          PushInteger(SQLOK);
          continu = 1;
          break;
```

SetLabel alters widget label text.

```
        case iLABEL:
          SetLabel((Int16)xitem); //?
          continu =1;
        break;
```

5.10.1 A test routine

We can hook various tests in here:

```
        case iTEST:
        // [deleted]

          continu =1;
        break;
```

5.10.2 Caching commands

We can turn caching on or off by saying CACHE(id) and CACHE(#0), where id is the primary key of the person (patient) for whom we wish to specifically enable caching. Only invoke this on entering a menu committed to that person, and make sure that caching is disabled on exit to more general menus. A powerful and dangerous tool.

```
      case iCACHE:
        cachez = PeekAndPopStringZ(0x7A); // nasty.
        if (cachez)
           { helpok = MAKECACHE (CACHECODE, cachez, StrLen(cachez) );
             if (helpok < 0)
```

```
            { ERRDisplay("Cache err =", helpok);
            };
        Delete(cachez);
      } else
      { ERRDisplay("BadCache", 0);
      };
    continu =1;
    break;
```

Similar is the uncaching function:

```
case iUNCACHE:
  cachez = PeekAndPopStringZ(0x7B); // nasty.
  if (cachez)
     { helpok = UNCACHE (CACHECODE, cachez, StrLen(cachez) );
       if (helpok < 0)
          { ERRDisplay("?Uncache=", helpok);
          };
       Delete(cachez);
     } else
     { ERRDisplay("UnCache?", 0);
     };
  continu =1;
  break;
```

5.10.3 MENU command

Trickery is needed in the following. If the initialisation script in NewMenu contains an iASK, then NewMenu must return IMWAITING, which in turn should be returned by iMENU below! Because we cannot multitask in PalmOS, we have to return back to the top level and wait for the response, and then resume where we left off! This is naasty.

```
case iMENU:
  // ROLLDEPTH = 0;  // THE PROBLEM WAS HERE!
  return PostMenuEvent();
```

We cleared ROLLDEPTH in the above to prevent a current ‘rolling’ status being carried over into an innocent menu, with disastrous consequences on display of polymorphic tables, but this messed up returns to rolled menus. So we transfer the zeroing of ROLLDEPTH to the relevant section of NewMenu (2008-08-04).

Here we implement ROLLMENU:

```
case iROLLMENU:
  if (LOCALROLL)
     { PushInteger(0); // push zero menu code to stack
```

```
        ROLLING = 1;
        return PostMenuEvent();
      };
continu = 1;
break;
```

If there is potential for a polytable to roll down in the next menu (some lines haven't been displayed, indicated by a nonzero value in LOCALROLL), then we invoke NewMenu by posting the relevant event to PalmOS; otherwise we do nothing! The PushInteger in the above is a little clumsy.[41]

We also allow interrogation to determine the number of lines *not yet displayed* in the polytable in this menu.

```
case iLINESLEFT:
  PushInteger(LINESLEFT);
  continu = 1;
break;
```

The following is also a clumsy hack [examine me!], at present inactive:

```
case iREFRESH:
  ERRmsg(10012);
break;
```

Display an alert box using the following:

```
case iALERT:
  if (! SayAlert())
     { return 0;
     };
  continu = 1;
break;
```

Launching the console is non-trivial. We have to clear the menu *before* we launch. After return, we must reinstate the menu. We need to work on this routine. ScriptIni completely clears the stack, we then try to re-enter the main menu.

```
case iCONSOLE:
  SleepMenu(); // clear menu
outcome = LaunchConsole();
 ERRDisplay("+Very trying",outcome); // [???]
  // reinstate ??? [fix me?!]
 topFxSTACK = 0;  // clean FUNCTION stack
 ScriptIni(SCRIPTLIBCODE, STACK, STACKSTRING);
 EnterMenu("MAIN",4);
 return 0;
```

[41] See how things pan out by looking at NewMenu.

Ask takes two items off the stack. On top is the default text to display, deeper is the message to put above the text input. Now FrmCustomResponseAlert doesn't allow us to specify default text (!) so we need a laborious workaround. We must:

1. Create and activate the input form;
2. Exit completely from the script, keeping our current locality
3. When either button (OK/Cancel) is hit, we resume where we left off!!

```
case iASK:                                  // ASK
  *((Char **)(FxSTACK+topFxSTACK)) = scriptline;
  // keep script line (above) and length:
  *((Int16 *)(FxSTACK+topFxSTACK+4)) = slLEN;
  topFxSTACK += 8;
  txtlen = 128; // get text context:
  txtbuf = xNew(txtlen, 0); // XNW17
  txtlen = ResolveX(txtbuf, 128);
  if (txtlen < 0)
     { ERRmsg(ErPopInAsk);
       Delete(txtbuf);
       return 0;
     };
  *(txtbuf+txtlen) = 0x0;  //asciiz for PalmOS
  poplen = 128; // next get title:
  msgbuf = xNew(poplen, 0); // XNW18
  poplen = ResolveX(msgbuf, 128);
  if (txtlen < 0)
     { ERRmsg(ErPopTitleAsk);
       Delete(msgbuf);
       Delete(txtbuf);
       return 0;
     };
  *(msgbuf+poplen) = 0x0;  //asciiz
  EXEITEM = xitem;  // keep record of current item!!
  MakeTextForm (msgbuf, txtbuf);
  Delete(msgbuf);
  Delete(txtbuf);
  return IMWAITING;
```

Returning IMWAITING forces exit, and later we respond to the button and re-enter.

Confirmation (using the Confirm function) returns 0 if the first (NO) button is pressed, 1 for YES.

```
case iCONFIRM:
  iresult = 0; // default
```

```
poplen = PeekLengthPlus();
if (! poplen)
   { ERRmsg(ErAlertFail); // [fix msg]
   } else
   { poplen += 0x10; // allow space for eg NULL!
     msgbuf = xNew(poplen, 0); // XNW19
     poplen = ResolveX(msgbuf, poplen);
     if (poplen < 0)
        { ERRmsg(ErFailResolve);
        } else
        { *(msgbuf+poplen) = 0x0;  //asciiz
          iresult=Confirm(msgbuf);
        };
     Delete(msgbuf);
   };
PushInteger(iresult);
continu = 1;
break;
```

Debug simply sets (or resets) the debugging flags.[42] We mirror a copy of the flags to the utility section.

```
case iDEBUG:
 if (PopInteger (&iresult))
    { DEBUGFLAGS = iresult;
      MirrorDebug(DEBUGFLAGS);
      WriteConsoleAsciiz(fDEBUG_ALWAYS, "\n" "DEBUG(");
      WriteConsoleInteger(fDEBUG_ALWAYS, iresult);
      WriteConsoleAsciiz(fDEBUG_ALWAYS, ")");
    };
 continu = 1;
break;
```

Quit moves the program to exit; if a run statement was executing, then rerun is cleared (In case quit ultimately fails based on user choice).

```
case iQUIT:
  if (RERUN)
     { Delete(RERUN);
       RERUN = 0;
     };
return DIEYOUBASTARD;
```

TITLE allows us to conveniently change the title of a menu.

[42] Do we need the ability to *read* the debugging flags. Probably yes!

```
case iTITLE:
  Char * ft;
  ft = w_FrmGetTitle(MYFORM);
  msgbuf = PeekAndPopStringZ(0x77);
  w_FrmSetTitle(MYFORM, msgbuf);
  Delete (ft);
  continu = 1;
break;
```

You would think that as we are replacing the PalmOS title in the above, we might retrieve and dispose of the old title using w_FrmGetTitle. PalmOS doesn't allow this by default, and in addition, PalmOS doesn't seem to automatically delete the new title when the menu is disposed of (FrmEraseForm) leaving a memory leak. So we sneakily force all newly created menus to take on a new title, which we can then replace/delete!

TOGGLE is very simple, setting the value in TOGGLED to 1:

```
case iTOGGLE:
  TOGGLED = 1;
  continu = 1;
break;
```

Dump dumps the stack for viewing.

```
case iDUMP:
  iresult = 10;
  StackDump(iresult);
  continu = 1;
break;
```

The return of iNORMAL implies the end of the script, and an unadorned break is fine in this case.

```
case iNORMAL:
break;
```

Penultimately, we handle iCOMPENSATED. This signals an 'error' (such as a failed number read) which isn't in fact an error — we have compensated by pushing a NULL to the stack, and the user can check this null and take appropriate action![43]

```
case iCOMPENSATED:
  continu = 1;
break;
```

[43] But we might conceivably have debugging code here.

Otherwise, the default is to fail, thus:

```
        default:
          ERRSTRING(thiscommand, thislen); // [??]
          WriteConsoleAsciiz(fDEBUG_AAGH, "\n Fx Err ");
          WriteConsoleText(fDEBUG_AAGH, thiscommand, thislen);
          WriteConsoleAsciiz(fDEBUG_AAGH,  "(" );
          WriteConsoleInteger(fDEBUG_AAGH, outcome);
          WriteConsoleAsciiz(fDEBUG_AAGH,  ")" );
          ConsoleDump(fDEBUG_STACK, 4);
          if (outcome < 0)
             { ERRmsg(ErScriptFx);
             } else
             { ERRmsg(ErScriptUnknown);
             };
          return 0;  // fail
     };   // END OF SWITCH STATEMENT
```

The ERRSTRING line above displays the errant line, which can be most useful! In the following (default not taken above, after end of switch) we check sLEN. If it's zero and there's still something on the command stack (topFxSTACK is nonzero) then we force a return, otherwise we post a normal outcome and continue.[44] By the way, if you leave out the continu=1 allocation, then a function without a return may terminate prematurely.

As we haven't yet included library error function codes in our error handling, obscure numbers will be returned. Not nice.

```
      if ((! slLEN) && topFxSTACK )
         { outcome = iRETURN;
           continu = 1;
         } else
         { outcome = NORMALOUTCOME;
         };
    }; // END OF 'continu' WHILE STATEMENT
return outcome; // [? check me ]
}
```

5.11 Other execution

Here we flesh out some of the functions invoked above.

[44] See how if we ERRDisplay("xxx",0) on the line following outcome=iRETURN, we crash PalmOS!

5.11.1 ExeQuery

ExeQuery executes an SQL SELECT statement. Ifverb+onlyone+ is set, then the first database row which meets the selection criterion is returned, and that's it. Otherwise, multiple rows are returned.

```
Int16 sql::ExeQuery(Int16 onlyone)
{ Int16 qlen;
  Char * qbuf;
  Int16 hits;
  Int16 poplen;
  Int16 ok=1;
  Int16 sq;

  SQLOK = 0; // default 'failure/warning'
  qlen = 1024; // max string length [?]
  sq = StackPeek(SCRIPTLIBCODE, STACK, STACKSTRING, 0, &qlen);
  if (  (sq < 0)
      ||(qlen < 20) // length of query bad?
     )
     { ERRmsg(ErSQLbadStmt);
         ERRmsg(10015); // [??]
       return 0;
     };
  qbuf = xNew(qlen, 0); // XNW20
  poplen = ResolveX(qbuf, qlen);
```

Ultimately SQL queries should themselves be able to resolve x-type variables, saving substantially on conversions. Here we take the easy (nasty) way out.

```
  if (poplen < 20)
     { ERRmsg(ErSQLbadStmt2);
       Delete(qbuf);
       return 0;                                  // fail
     };
  if (  xSame (qbuf, "SELECT ", 7)
      ||xSame (qbuf, "select ", 7)
     )
     { hits = SQLselect(qbuf+7, poplen-7, onlyone);
     } else
     { hits = 0;
     };
```

In the above, check for and clip off the initial SELECT, then execute the SELECT statement.

```
  if (! hits)
```

```
     { ERRmsg(ErFailSQL);
       ERRSTRING(qbuf,poplen); // [??]
       ok = 0;
     };
  if (hits > 1)
     { SQLOK = 1;
     };
  Delete(qbuf);
  return ok;
};
```

The value in `hits` is zero if an *error* occurred, one if no result was put on the stack (The default zero value of SQLOK is only changed if `hits` is *over* one).

5.11.2 ExeSQL

ExeSQL executes an SQL statement that is *not* a SELECT. Contrast this with ExeQuery. We use a larger buffer size in qlen. In the following, submitting a zero as the second last parameter of StackPeek retrieves nothing. At present a rather limited choice exists!

```
Int16 sql::ExeSQL()
{ Int16 qlen;
  Int16 poplen;
  Char * qbuf;
  Int16 ok=0;
  qlen = 2048;

  if (  (StackPeek (SCRIPTLIBCODE,
          STACK, STACKSTRING, 0, &qlen) < 1)
      ||(qlen < 20)
     )
     { ERRmsg(ErSQLbadStmtSmall);
       return 0;
     };
  qlen += 100; // just in case
  qbuf = xNew(qlen, 0); // XNW21
  poplen = ResolveX(qbuf, qlen);
  if (poplen < 20)
     { ERRmsg(ErSQLbadStmt3);
       Delete(qbuf);
       return 0;                                  // fail
     };
  if ( xSame(qbuf, "CREATE TABLE ", 13))
     {  ERRmsg(10016);
       // [TEMPORARILY REMOVED]
     } else
```

```
     { if (xSame(qbuf, "INSERT INTO ", 12))
          { ok = SQLinsert(qbuf+12, poplen-12);
          } else
          { if (xSame(qbuf, "UPDATE ", 7))
               { ok = SQLupdate(qbuf+7, poplen-7);
               } else
               { ERRmsg(ErSQLunknownStmt);
                 Delete(qbuf);
                 return 0;                        // fail
     };   };   };
  Delete(qbuf);
  return ok;
};
```

5.12 Debugging-style routines

The following routines are rather clumsy.

5.12.1 InsertAndGo

Given an integer, insert it on the stack, and then execute the SQL string provided.[45]

```
Int16 sql::InsertAndGo(Int32 widgetcode, Char * cmd)
{
///  WriteConsoleText(fDEBUG_STMT, "%", 1); // jvs 20080515
  Int16 clen;
  if (!PushInteger(widgetcode))
     { return 0;
     };
  clen = w_StrLen(cmd);
  Int16 ok;
  ok = ExeString(0, cmd, clen);
  return ok;
}
```

Very similar is InsertAndGoV. The added parameter is the V value (a database primary key value) which allows ExeString to reference this value:

```
Int16 sql::InsertAndGoV(Int32 widgetcode, Char * cmd, Int32 V)
{
  Int16 clen;
  if (!PushInteger(widgetcode))
     { return 0;
     };
```

[45] See how inserting an ERRDisplay immediately after the ExeString line crashes PalmOS intermittently!

```
///  WriteConsoleText(fDEBUG_STMT, "/", 1); // jvs 20080515
  clen = w_StrLen(cmd);
  Int16 ok;
  ok = ExeString(V, cmd, clen);
  return ok;
}
```

5.12.2 JustGo

JustGo is similar to InsertAndGo, but has no integer insertion parameter:

```
Int16 sql::JustGo(Char * cmd)
{ Int16 clen;
  clen = w_StrLen(cmd);
///  WriteConsoleText(fDEBUG_STMT, "$", 1); // jvs 20080515
  return ExeString(0, cmd, clen);
}
```

5.13 Script function handling

Scripted functions are identified by their intial ampersand, much as subroutines or functions are identified in Perl. Functions are contained *within* the PalmOS database in the FUN table. We can thus retrieve functions via an SQL query, but to speed things we do the following:

1. At startup, create a buffer area called FxBUFFER. It will be up to 32K in size although at present we only need about 2K.

2. Examine each record in FUN (from 1 onwards), and keep details in FxBUFFER thus:

Offset	*Details*
+0	a 4-byte number : locked pointer to relevant record
+4	offset of fx name
+6	length of fx name
+8	offset of fx body
+A	length of fx body
+C	index of record in PalmOS database
+E	(4 spare bytes)

Table 2: Structure of record representation in FxBUFFER

Each record is represented by 16 bytes in FxBUFFER. Records are sorted by name, we keep FxBUFFER open, and when we invoke a function, we use a binary search of FxBUFFER to locate the record.

When we invoke a function, we must keep a record of the prior function, *as well as* the current evaluation offset within that function. We create FxSTACK which devotes 4 bytes to each fx: 2 for the index of the function into FxBUFFER, and 2 for the offset.

5.13.1 IniFxBuffer

Initialisation of the function buffer. We first set aside memory for buffer and stack:

```
Int16 sql::IniFxBuffer()
{
 if (! MakeDiskBuffer ("SQLFXBUFFER", 11, 1, FXMAX))
        { ERRmsg(ErMakeStack);
          return -1;
        };
  pdbFXBUFFER = PalmFileOpen("SQLFXBUFFER", 11);
  if (! pdbFXBUFFER)
     { ERRmsg(ErFunctionBuffer);
       return -2;
     };
  FxSTACK = xNew(FXSTACKSIZE, 0xD8); // XNW22
  if (!FxSTACK)
     { ERRmsg(ErFunctionMemory);
       return -3;
     };
```

Then open FUN database.

```
  MemHandle hFX;
  Int16 fail;

  hFX = w_DmGetRecord(pdbFXBUFFER, 0);
  if (! hFX) //  v0.95
     { fail = ReconstituteRecord (pdbFXBUFFER, 0);
       if (fail)
          { return -9; // 20080510
          };
       hFX = w_DmGetRecord(pdbFXBUFFER, 0);
       if (! hFX) { return -10; }; // aagh.
     };
  FxBUFFER = (Char *) w_MemHandleLock(hFX); // MHL10

  LocalID lid; // foul constraint imposed by PalmOS
  lid = u_DmFindDatabase ("FUN", 3); // non-asciiz
```

```
if (! lid)
   { ERRmsg(ErNoFunctionDB);
     return -4;
   };
FUNFILE = w_DmOpenDatabase (lid, dmModeReadOnly);
if (! FUNFILE)
   { ERRmsg(ErOpenFunctionDB);
     return -5;
   };
```

Next, find number of records (a sluggish routine)

```
UInt32 recs; // (under 32K)
if (! w_DmDatabaseSize (lid, &recs, 0, 0) ) // 0=failed
   { ERRmsg(ErBadDbSizeCount);
     return -6; //fail
   };
FxCOUNT = (Int16) recs;
FxSORTED=xNew(FxCOUNT*2, 0xD9); // XNW23
topFxSTACK=0; // nil on stack at start
```

Unfortunately we cannot put the following in a library, as pointer stuffups then occur:

```
MemHandle thishand;
Char * thisp;
Int16 fxname;
Int16 fxnamelen;
Int16 fxbody;
Int16 fxbodylen;
Int16 Fx=0; // offset of current fx!
Int16 i=1;

while (i < FxCOUNT)
  { thishand = DmQueryRecord(FUNFILE, i); // read only
    if (! thishand)
       { ERRmsg(ErIniFxHandle);
         return -7;
       };
    thisp = (Char *)MemHandleLock(thishand); // MHL11
```

Offsets in `thisp` are key offset at 0x10, body offset at 0x12, name offset at 0x14, and top of name at 0x16. We pass the following data within FxBUFFER to IniFx, a library function in the SCRIPTING library.

```
    fxbody = *((Int16 *)(thisp+0x12));
    fxname = *((Int16 *)(thisp+0x14));
```

```
      fxbodylen = fxname - fxbody;
      fxnamelen = *((Int16 *)(thisp+0x16)) - fxname;
      w_DmWrite(FxBUFFER, Fx+0, &thisp, 4 );
      w_DmWrite(FxBUFFER, Fx+4, &fxname, 2 );
      w_DmWrite(FxBUFFER, Fx+6, &fxnamelen, 2 );
      w_DmWrite(FxBUFFER, Fx+8, &fxbody, 2 );
      w_DmWrite(FxBUFFER, Fx+0xA, &fxbodylen, 2 );
      w_DmWrite(FxBUFFER, Fx+0xC, &i, 2 );
      Fx += 16;
      i ++;
    };
  FxCOUNT --;  // ignore header record in count!
  Int16 ok;
  ok = IniFx(SCRIPTLIBCODE, FxBUFFER,
             FUNFILE, FxCOUNT, FxSORTED);
  if (ok < 0)
     { ERRmsg(ErFunctionInitialisation);
       return -8;
     };
  return 0; // success
}
```

5.13.2 CloseFxBuffer

The converse of IniFxBuffer. In the following, the while statement has `i` less than or equal to FxCOUNT as we exclude record zero, the header. The while statement allows us to unlock all locked pointers.

```
Int16 sql::CloseFxBuffer()
{
  Delete(FxSORTED);
  Delete(FxSTACK);

  Char * thisp;
  Int16 Fx=0;
  Int16 i=1;
  while (i <= FxCOUNT)
    { thisp = * ((Char **)(FxBUFFER+Fx+0));
      // Point to stored Char*, retrieve pointer
      if (! w_MemPtrUnlock(thisp))
         { ERRmsg(ErFreeFxPtr);
           return 0;
         };
      Fx+=16;
      i ++;
    };
  if (!PalmFileClose(FUNFILE))
     { ERRmsg(ErCloseFx);
```

```
    };
  if (! w_MemPtrUnlock(FxBUFFER))
    { ERRmsg(ErFreeFxBuffer);
      return 0;
    };
  if (! w_DmReleaseRecord(pdbFXBUFFER, 0, false))
    { ERRmsg(ErFreeFxBufrec);
      return 0;
    };
  if (! PalmFileClose(pdbFXBUFFER))
    { ERRmsg(ErCloseFxBuffer);
      return 0;
    };
  return 1;
}
```

5.13.3 PushCurrentScript

Here we push the current script to the function stack. We store away the pointer to the script, and the remaining length.

```
/// HMM:
Int16 sql::PushCurrentScript (Char * scriptline, Int16 remlen)
{ *((Char **)(FxSTACK+topFxSTACK)) = scriptline;
  *((Int16 *)(FxSTACK+topFxSTACK+4)) = remlen;
  topFxSTACK += 8;  // 8 bytes per fx (2 spare)
  if (topFxSTACK >= FXSTACKSIZE) // stack overflow
    { ERRmsg(ErFxStackOver);
      topFxSTACK -= 8; // correct error.
      return 0; // fail
    };
  return 1;
}
```

The stack overflow test could be moved up.

5.14 Alerts and interaction

Interactive stuff with pop-up boxes.

5.14.1 MakeTextForm

You might think that FrmCustomResponseAlert looks good, but there is one teeny problem — we cannot have default text at startup (there might be some hack, but I couldn't find it!) We thus need to create our own little form, with text box, and fill the text ...

Note that the height must be 156 while debugging because of a stupid ‘fatal alert’ which otherwise occurs!

```
Int16 sql::MakeTextForm (Char * title, Char * content)
{
  ASKFORM = FormMakeDynamic (title, 2, 2, 156, 156);
  if (! ASKFORM)
     { return 0;
     };
  OKBUTTON = InsertWidget (&ASKFORM, BUTTONCTL,
             10, 85, 50, 12, "Ok", 0,  0, 1, 0, 1);
  if (! OKBUTTON)
     { return 0;
     };
  QUITBUTTON = InsertWidget (&ASKFORM, BUTTONCTL,
             100, 85, 50, 12, "Cancel", 0, 0, 1, 0, 1);
  if (! QUITBUTTON)
     { return 0;
     };
  USERTEXT =  InsertWidget (&ASKFORM, FIELDNOTCTL,
             10, 20, 138, 12, content, 0,  0, 1, 0, 1);
  if (! USERTEXT)
     { return 0;
     };
  return FormActivate(ASKFORM);
};
```

For FormActivate see section 4.7.2.

5.14.2 KillTextForm

This also erases the component buttons.

```
Int16 sql::KillTextForm ()
{  w_FrmEraseForm(ASKFORM);
   w_FrmDeleteForm(ASKFORM);
   return 1; // ok
}
```

5.14.3 RespondToAsk

The following creates a response to the ASK form which we pop up. We get the index of the USERTEXT object, point to the object, and retrieve text.

```
Int16 sql::RespondToAsk(Int16 item)
{ Int16 obji; // to get text field
  obji = w_FrmGetObjectIndex(ASKFORM, USERTEXT);
```

```
  FieldType * okft1; // pointer to object
  okft1 = (FieldType *) w_FrmGetObjectPtr(ASKFORM, obji);
  Char * usetext;
  Int16 uselen;
  usetext = FldGetTextPtr(okft1);  // retrieve text
```

If text exists and we haven't cancelled we push the text, otherwise we force a null result.

```
  if ((usetext) && (item != QUITBUTTON))
     { uselen = w_StrLen(usetext);
     } else
     { uselen = 0; // force null
     };
  PushItem (SCRIPTLIBCODE, STACK,
    STACKSTRING, usetext, uselen, 'V', 0);
  KillTextForm(); // this works!
  QUITBUTTON = -1;
  USERTEXT = -1;
  OKBUTTON = -1;
  return( ExeScript(EXEITEM));
};
```

We don't seem to have to unlock `usetext`, but must destroy the form. We then resume where we left off, by invoking ExeScript. EXEITEM is a clumsy 'global'!

5.14.4 SayAlert

Popping up an 'alert' is relatively easy, but note that Resolve may result in a string slightly longer than the initial datum.

```
Int16 sql::SayAlert()
{ Int16 poplen;
  Char * msgbuf;
  Int16 ok = 1;

  poplen = PeekLengthPlus();
  if (! poplen)
     { ERRmsg(ErAlertFail);
       return 0;
     };
  poplen += 0x10; // allow space for eg NULL!
  msgbuf = xNew(poplen, 0); // XNW24
    // [might here check whether xNew failed]
  poplen = ResolveX(msgbuf, poplen);
  if (poplen < 0)
```

```
     { ERRmsg(ErFailResolve);
       ERRDisplay("C:",poplen); // debug
       Int16 xtype;
       xtype = PopItem(SCRIPTLIBCODE, STACK,
                   STACKSTRING, msgbuf, &poplen);
       ERRDisplay("T:", xtype);
       ERRSTRING(msgbuf, poplen); // [??]
       ok = 0;
     } else
     { *(msgbuf+poplen) = 0x0;  //asciiz
       FrmCustomAlert (MyAlert, msgbuf, "", "");
     };
  Delete(msgbuf);
  return ok;
}
```

5.14.5 Confirm

Given ASCIIZ string, return 1/0. [?check me]

```
Int16 sql::Confirm(Char * msg)
{
  if ( FrmCustomAlert (MyConfirm, msg, "", "") )
     { return 1;
     };
  return 0; // first button (NO) is zero, 2nd (YES) is 1.
}
```

5.15 Configuration and colour handling — rudimentary

This section needs extensive work.

5.15.1 HexColour

Read a hexadecimal colour, using a library function:

```
Int32 sql::HexColour (Char * clrstring, Int16 clen)
{
   if (* clrstring != '#')
      { ERRmsg(ErColour);
        ERRSTRING(clrstring, clen); // [??]
        return 0;
      };
   Int32 i;
   Int16 ok;
   ok = UsefulRead(SCRIPTLIBCODE,
```

```
          HEXADECIMALCOLOUR, (Char *)&i, 4,
          clrstring, clen);
   if (ok < 0)
      { return 0;
      };
   return i;
}
```

5.16 SetPaperOrInk

A stub.

```
Int16 sql::SetPaperOrInk (Int16 id, Int16 isink)
{ Char * cname;
  cname = PeekAndPopStringZ(0x78);
  if (cname)
     { Delete(cname);
       return 0;
     };
return 1;
}
```

5.16.1 HackWidget

Given the PalmOs ID of an object, find it in the current menu, returning a void pointer to the object. Also returns the type of object in the by reference variable MYKIND (of nature FormObjectKind).

We obtain the object index, the type and then a pointer to the actual object.

```
void * sql::HackWidget (Int16 id, Int16 * MYKIND)
{
  if ((id < 0) || (! MYKIND))
     { ERRmsg (ErObjHackID);
       return 0;  // fail
     }; // sanity check

  UInt16 obji;  // object index
  obji =  w_FrmGetObjectIndex (MYFORM, id);
  if (obji < 0)
     { ERRmsg (ErObjHackID);
       return 0;  // fail
     };
  * MYKIND = (Int16) FrmGetObjectType(MYFORM, obji); // kind
  void * myobj; // pointer to object itself.
  myobj = FrmGetObjectPtr(MYFORM, obji);
  return (myobj);
}
```

5.16.2 Enabled

Cumbersome enabling of a widget. First get object index and type and point to it using HackWidget above. Finally, fiddle with the object itself, if possible.

```
Int16 sql::Enabled (Int16 id, Int16 yesno)
{
  void * myobj;
  Int16 mykind;
  FieldAttrType flAt;

  if (id == 0) // signals ‘not yet created’
     { ABLING = yesno;
       return 1; // success, of a kind.
     }; // See CreateOneWidget/InsertWidget.

  myobj = HackWidget(id, &mykind);
  if (! myobj)
     {
       WriteConsoleErr(fDEBUG_AAGH, "id",id);
       ERRmsg(ErEnableObj);
       return 0;
     };

  switch (mykind)
    {
        case frmFieldObj:
          FldGetAttributes((FieldType *)myobj, &flAt);
          flAt.editable = 0;
          FldSetAttributes((FieldType *)myobj, &flAt);
          break;

        case frmControlObj:
          ControlType * ctp;
          ctp = (ControlType *) myobj;
          CtlSetEnabled(ctp, yesno);  // enable
          break;

        case frmListObj:
          break;

        case frmTableObj:
          break;

        case frmBitmapObj:
          break;

        case frmLineObj:
          break;
```

```
        case frmFrameObj:
          break;

        case frmRectangleObj:
          break;

        case frmLabelObj:
          break;

        case frmTitleObj:
          break;

        case frmPopupObj:
          break;

        case frmGraffitiStateObj:
          break;

        case frmGadgetObj:
          break;

        case frmScrollBarObj:
          break;

        default:
          ERRmsg(ErEnableObj);
    };
  return 1;
}
```

By the way, there's a typo in the PalmOS documentation: 'frmScrollbarObj'.

5.16.3 SetLabel

Similar to 'Enabled' above, uses HackWidget.

```
Int16 sql::SetLabel (Int16 xitem)
{
  void * myobj;
  Int16 mykind;
  Char * newlabel;
  Int16 llen=128;

  myobj = HackWidget(xitem, &mykind);
  if (! myobj)
     { return 0;
     };
```

```
newlabel = xNew(llen, 0); // XNW25
llen = PopString(newlabel, llen);
if (! llen)
   { ERRmsg(ErLabel);
     Delete (newlabel);
     return 0;
   };
*(newlabel+llen) = 0x0; // asciiz

switch (mykind)
  {   case frmFieldObj: // no label
        break;

      case frmControlObj:
        ControlType * ctp;
        ctp = (ControlType *) myobj;
        CtlSetLabel (ctp, newlabel);
        break;

      case frmLabelObj:
        break;

      case frmTitleObj:
        break;

      case frmPopupObj:
        break;

      // the following all have no label:
      case frmListObj:
        break;
      case frmTableObj:
        break;
      case frmBitmapObj:
        break;
      case frmLineObj:
        break;
      case frmFrameObj:
        break;
      case frmRectangleObj:
        break;
      case frmGraffitiStateObj:
        break;
      case frmGadgetObj:
        break;
      case frmScrollBarObj:
        break;
```

```
        default:
          ERRmsg(ErLabelObject);
          Delete(newlabel); // clumsy
          return 0;
    };
  Delete(newlabel);
  return 1;
}
```

5.17 Misc. Widget utilities

A miscellaneous bunch of widget-related utility functions follows. They handle widgets within a linked list of our creation.

5.17.1 MakeWidget

Our WIDGET type nodes contain information linking the unique (nonce) id of the widget with its database code (and thus, database properties, for example scripts). Here we create such a node. Widget types are in Table 1.

```
WIDGET * sql::MakeWidget (Int32 uid, Int16 dbid,
                          Int32 v, Int16 mytype)
{ WIDGET * wp = (struct WIDGET *) xNew (sizeof(struct WIDGET), 0xD0); // XNW26
  xFill((Char *) wp, sizeof(struct WIDGET), 0); // clean
  if (! wp)
     { ERRmsg(ErNotWidget);
       return 0;
     };
  wp->uid = uid;
  wp->dbid = dbid;
  wp->mytype = mytype;
  wp->V = v; // see monomorphic table
  // +OPTIONAL
  // WriteConsoleErr(fDEBUG_WIDGET, "!",wp->uid); // debug+
  // -OPTIONAL
  return wp;
}
```

5.17.2 PrependWidget

This function *prepends* a widget to the linked list of widgets. The list grows from ROOTWIDGET.

```
Int16 sql::PrependWidget (Int32 uid, Int16 dbid,
                          Int32 v, Int16 mytype)
{
```

```
  WIDGET * wp = MakeWidget(uid, dbid, v, mytype);
  if (!wp) { return 0; };
  wp->next = ROOTWIDGET;
  ROOTWIDGET = wp;
  return 1;
}
```

For widget types, consult Table 1.

5.17.3 KillOnId

Given the ID of a widget, KillOnId kills it (excise it from the linked list). KillOneWidget is a subsidiary function which deletes the memory allocated to the widget.

```
Int16 sql::KillOnId (Int32 uid)
{ WIDGET * wp = ROOTWIDGET;
  WIDGET * np = wp->next;
  if (wp->uid == uid)
     { ROOTWIDGET = np;
       KillOneWidget(wp);
       return 1;
     };
  while (np)
    { if (np->uid == uid)
         { wp->next = np->next;
           KillOneWidget(np);
           return 1;  // ok
         };
      wp = np;
      np = wp->next;
    };
  return 0; // fail
};

Int16 sql::KillOneWidget( WIDGET * wp)
{
 // +OPTIONAL
 // WriteConsoleErr(fDEBUG_WIDGET, "~",wp->uid); // debug+
 // -OPTIONAL
  return Delete ((Char *) wp); // [?? test success]
}
```

5.17.4 KillAllWidgets

Delete *all* widgets, clear the ROOTWIDGET.

```
Int16 sql::KillAllWidgets ()
{ WIDGET * nxt;
  WIDGET * wp = ROOTWIDGET;
  ROOTWIDGET = 0; // necessary
  while (wp)
    { nxt = wp->next;
      KillOneWidget(wp); // [?? test success]
      wp = nxt;
    };
  return 1;
};
```

5.17.5 FindWidgetDatabaseCode

Given the unique ID of the widget, find its *database* id.

```
Int16 sql::FindWidgetDatabaseCode (Int32 uid)
{ WIDGET * wp;
  wp = ROOTWIDGET;
  while (wp)
    { if (wp->uid == uid)
         { return (wp->dbid); // success
         };
      wp = wp->next;
    };
  return 0; // fail.
};
```

5.17.6 FindWidgetType

Given the unique ID of the widget, determine its *type*.

```
Int16 sql::FindWidgetType (Int32 uid)
{ WIDGET * wp;
  wp = ROOTWIDGET;
  while (wp)
    { if (wp->uid == uid)
         { return (wp->mytype); // success
         };
      wp = wp->next;
    };
  return 0; // fail
};
```

5.17.7 VValue

Given the unique ID of a widget, determine its V value (associated database value).

```
Int32 sql::VValue (Int32 uid)
{ // hack: if negative, then simply negate and return! (already established!!)
  if (uid < 0)
     { return -(uid);
     };

  WIDGET * wp;
  wp = ROOTWIDGET;
  +OPTIONAL
  WriteConsoleAsciiz(fDEBUG_WIDGET, "[V");
  -OPTIONAL

  while (wp)
    {
      if (wp->uid == uid)
         {
          +OPTIONAL
          WriteConsoleInteger(fDEBUG_WIDGET, wp->uid); // debug+
          WriteConsoleAsciiz(fDEBUG_WIDGET, "]");
          -OPTIONAL
          return (wp->V);  // success
         };
      +OPTIONAL
      WriteConsoleInteger(fDEBUG_WIDGET, wp->uid); // debug+
      WriteConsoleAsciiz(fDEBUG_WIDGET, ",");
      -OPTIONAL
      wp = wp->next;
    };

  +OPTIONAL
  WriteConsoleErr(fDEBUG_WIDGET, "?->", uid);
  -OPTIONAL
  return 0; // fail
};
```

5.17.8 FindWidgetUID

Conversely, given the *database* ID of the widget, determine its unique ID.

```
Int32 sql::FindWidgetUID (Int16 dbid)
{ WIDGET * wp;
  wp = ROOTWIDGET;
  while (wp)
    { if (wp->dbid == dbid)
         { return (wp->uid); // success
         };
      wp = wp->next;
    };
```

```
  return 0; // fail
};
```

5.18 Script variables

Local variables within scripts are manipulated using the following functions.

5.18.1 MakeLocalVariables

First, we create the local variables, setting aside a buffer, opening it, and intialising all of these variables to null.

```
Int16 sql::MakeLocalVariables()
{
  if (! MakeDiskBuffer ("SQLLOCALS", 9, 1, LOCALSIZE))
        { ERRmsg(ErMakeLocals);  // ???
          return -101;
        };
  pdbLOCALS = PalmFileOpen("SQLLOCALS", 9);
  if (! pdbLOCALS)
     { ERRmsg(ErOpenLocals);
       return -102;
     };
  MemHandle hLOCALS;
  hLOCALS = w_DmGetRecord(pdbLOCALS, 0);
  if (! hLOCALS)
     { Int16 ec0 = ReconstituteRecord (pdbLOCALS, 0); // as elsewhere v0.95
       if (ec0)
          { PalmFileClose(pdbLOCALS); // as for OpenStack
            LocalID mydbid;
            mydbid = u_DmFindDatabase("SQLLOCALS", 9);
            w_DmDeleteDatabase(mydbid); // delete
            return -103; // or perhaps (ec0-1000);
          };  // else..
       hLOCALS = w_DmGetRecord(pdbLOCALS, 0);
       if (! hLOCALS) { return -104; };
     };
  LOCALS = (Char *) w_MemHandleLock(hLOCALS); // MHL12

  Int16 ok;
  ok = IniVariables (SCRIPTLIBCODE, LOCALS, LOCALSIZE);
  if (ok < 0)
     { ERRmsg(ErFailIniLocals);
       return -104;
     };
  return 0;  // ok
}
```

We return 0 on success, or an error code under -100.

5.18.2 KillLocalVariables

The opposite of MakeLocalVariables.

```
Int16 sql::KillLocalVariables()
{ if (! pdbLOCALS)
     { return 0; // fail
     };
  if (! w_MemPtrUnlock(LOCALS))
     { ERRmsg(ErFailLocalsUnlock);
       return 0;
     };
  if (! w_DmReleaseRecord(pdbLOCALS, 0, false))
     { ERRmsg(ErFailLocalsFree);
       return 0;
     };
  if (! PalmFileClose(pdbLOCALS))
     { ERRmsg(ErFailLocalsClose);
       return 0;
     };
return 1;  // ok
}
```

5.19 Various functions

A selection of functions which should be amalgamated with other odds and ends from above!

5.19.1 PopMenu

As with the Perl program, we have revised POPMENU to take an integer off the stack, using this to clip out the specified menu (and X value) from the menu stack. A teensy problem is that we've now also introduced the ROLLDEPTH variable as a MENUSTACK parameter.

CURRENTMENU should not be affected unless we specify an index of zero, as we don't store CURRENTMENU on the menu stack until we go to a new menu. If the index is zero, we should effectively pop the top menu; practically we should always follow POPMENU(0) with a MENU(xxx).

The other issue with POPMENU(0) is ROLLDEPTH, which must also be popped in these circumstances.

We locate the position of the menu to clip out, noting that MENUINDEX points to the first *free* item on the menu stack but that this position should really

be occupied by CURRENTMENU, so we do *not* decrement idx before we retrieve the idx menu!

```
Int16 sql::PopMenu()
{ Int32 idx;

  if (! PopInteger(&idx))
     { ERRmsg(ErPopMenuStack1);
       return 0;
     };

  if ((idx > MENUINDEX) || (idx < 0))
     { ERRmsg(ErPopMenuStack2);
       return 0;
     };
  idx = MENUINDEX - idx; //

  // fix things up thus: first push current!! $jvs$
  *((Int16 *)(MENUSTACK + 0 + sizeM*MENUINDEX)) = CURRENTMENU;
  *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX)) = XPARAM;
  *((Int16 *)(MENUSTACK + 6 + sizeM*MENUINDEX)) = ROLLDEPTH;

            // debugging only: $jvs$
            // WriteConsoleText(fDEBUG_ALWAYS, "\nPPMENU ", 8);
            // WriteConsoleInteger(fDEBUG_ALWAYS, CURRENTMENU);
            // WriteConsoleText(fDEBUG_ALWAYS, " rd=", 4);
            // WriteConsoleInteger(fDEBUG_ALWAYS, ROLLDEPTH);

  Int32 mX;
  Int16 mMenu;
  mX = *((Int32 *)(MENUSTACK + 2 + sizeM*idx)); // FETCH
  mMenu = *((Int16 *)(MENUSTACK + 0 + sizeM*idx));

  while (idx < MENUINDEX)
    { xCopy(MENUSTACK + sizeM*idx,
            MENUSTACK + (sizeM*(idx+1)), sizeM);
      idx ++; // clumsy
    };

  MENUINDEX --;
  CURRENTMENU  = *((Int16 *)(MENUSTACK + 0
                   + sizeM*MENUINDEX));
  XPARAM = *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX));
  ROLLDEPTH = *((Int16 *)(MENUSTACK + 6 + sizeM*MENUINDEX));
  // only if idx was zero, will alter these ?!

            // debugging only: $jvs$
            // WriteConsoleText(fDEBUG_ALWAYS, " -> ", 4);
```

```
            // WriteConsoleInteger(fDEBUG_ALWAYS, CURRENTMENU);
            // WriteConsoleText(fDEBUG_ALWAYS, " rd=", 4);
            // WriteConsoleInteger(fDEBUG_ALWAYS, ROLLDEPTH);

  // mMenu is database id of menu!
  PushInteger(mX); // push X to the stack!
  PushInteger(mMenu); // push id of menu
  if ( GoScript ("\"SELECT ITEM.iName FROM ITEM WHERE ITEM.iID = $[]\"")
       && (ExeQuery(1))
     )
     { return 1;
     };
  return 0;
}
```

There's discordance with our Perl coding, as there we stored the menu name, here we store the database ID of the menu. We resolve this with the clumsy GoScript SQL query.

The previous code follows.

```
Int16 sql::PopMenu()
{
  if (MENUINDEX <= 0)
     { ERRmsg(ErPopMenuStack);
       return 0;
     };
  MENUINDEX --;

  Int32 mX;
  Int16 mMenu;
  mX = *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX)); // X
  mMenu = *((Int16 *)(MENUSTACK + 0 + sizeM*MENUINDEX));
    // database id of menu
  PushInteger(mX); // push X to the stack!
  PushInteger(mMenu); // push id of menu
  if ( GoScript ("\"SELECT ITEM.iName FROM ITEM \
         WHERE ITEM.iID = $[]\"")
       && (ExeQuery(1))
     )
     { return 1;
     };
  return 0;
}
```

5.19.2 PushMenu

The reverse of PopMenu, this function still needs to be *written* in Perl, to allow for general purpose menu pushes and pops! [FIX ME!] Given the menu name as a string on the stack, with X below it on the stack, we push these to the *menu stack*, as if we had come from this menu/X combination [??? what?].

```
Int16 sql::PushMenu()
{
  if (! GoScript("\"SELECT ITEM.iID FROM ITEM WHERE ITEM.iName = '$[]'\"") )
     { ERRmsg(10017);
       return 0; // fail
     };
  if (! ExeQuery(1))
     { ERRmsg(10018);
       return 0;
     };
  Int32 mX;
  Int32 mMenu;
  if (  !(PopInteger(&mMenu))
      ||!(PopInteger(&mX))
     )
     { ERRmsg(10019);
       return 0;
     };
  *((Int32 *)(MENUSTACK + 2 + sizeM*MENUINDEX)) = mX;
    // X param
  *((Int16 *)(MENUSTACK + 0 + sizeM*MENUINDEX)) = (Int16) mMenu;
    // database id of menu
  MENUINDEX ++;
  return 1;
};
```

5.19.3 GoScript

A trivial, ugly function to invoke DoScript on an ASCIIZ string.

```
Int16 sql::GoScript(Char * cmd) // aagh. asciiz.
{ if (DoScript(SCRIPTLIBCODE, STACK,
       STACKSTRING, cmd, w_StrLen(cmd)) > 0)
   { return 1;
   };
  return 0;
}
```

5.19.4 FindFunction

given a function name, locate it within the functions. Cumbersome. SearchFxBody is a library call (SCRIPTING.C).

```
Char * sql::FindFunction(Char * thiscommand,
          Int16 thislen, Int16 * foundlen)
{ Char * scriptline;
  scriptline = SearchFxBody(SCRIPTLIBCODE,
                            thiscommand, thislen,
                            foundlen, // also get length
                            FxSORTED, FxCOUNT, FxBUFFER);
  if (! scriptline)
     { ERRmsg(ErNoFunction);
       ERRSTRING(thiscommand, thislen); // [??]
       return 0;  // fail
     };
  return scriptline;
};
```

5.19.5 DoReturn

Process a RETURN statement. Also sorts out the problem of cleaning up after return from function invoking a RUN statement.

```
Char * sql::DoReturn(Int16 * retlen, Boolean istop)
{ Char * scriptline;  // clumsy, could inline for speed ??
  topFxSTACK -=8;     // go back on stack
  if (RERUN) // if a run statement was executing!
     {      if (topFxSTACK == RUNLEVEL)
               { Delete(RERUN); // see iRUN
                 RERUN = 0;
               };
     };   // ONLY clear RERUN string if back to relevant level
  if (topFxSTACK < 0)
     { topFxSTACK = 0; // correct the problem!
       return 0;       // fail
     };
  scriptline = *((Char **)(FxSTACK+topFxSTACK));
    // pop pointer
  *retlen = *((Int16 *)(FxSTACK+topFxSTACK+4));
    // and remaining length!
  return scriptline;
};
```

5.20 More debugging

Display the whole stack and stackstring. We show the stack bottom, top, maximum and number of items, followed by the stackstring (bottom, top, max) and then each item (its type, short length/actual length and offset on stackstring, and if it's type X, also show the number of items in X, the offset of the first X item, the offset of the insertion string, and the actual insertion string). X-items are compound, as yet unresolved items.

The submitted `icount` parameter specifies the maximum depth to go down to. A value of zero forces display of stack parameters only.

5.20.1 StackDump

```
Int16 sql::StackDump(Int16 icount)
{  Int16 top;
   Int16 bot;
   Int16 max;
   Int16 itms;
   top = *((Int16*)(STACK+oTOP));  // stack first..
   bot = *((Int16*)(STACK+oSTART));
   max = *((Int16*)(STACK+oMAX));
   ERRADD("Dump: ",6);
   ERRINT(bot/16);
   ERRADD(",",1);
   ERRINT(top/16);
   ERRADD(",",1);
   ERRINT(max/16);
   itms = (top-bot)/16;
   ERRADD(" i=",3);
   ERRINT(itms);

   Int16 SStop; // stackstring next
   Int16 SSbot;
   Int16 SSmax;
   SStop = *((Int16*)(STACKSTRING+oTOP));
   SSbot = *((Int16*)(STACKSTRING+oSTART));
   SSmax = *((Int16*)(STACKSTRING+oMAX));
   ERRADD("\x0A" "SS: ",5);
   ERRINT(SSbot);
   ERRADD(",",1);
   ERRINT(SStop);
   ERRADD(",",1);
   ERRINT(SSmax);

   ERRSHOW(0); // display, then individual items:
   if (itms < icount) { icount = itms; };
   while (icount > 0)
```

```
     { top -= 16;  // move down to (first) item
       ShowStackItem(STACK+top, icount, 1);
       icount --;
     };
  return 1;
}
```

5.20.2 ShowStackItem

Display a stack item. Allows X-items to be displayed, with 1-deep recursion.

```
Int16 sql::ShowStackItem(Char * ISRC, Int16 icount,
                         Boolean recur)
{ Char c;
  Int16 ilen;
  Int16 ilentrue=0;
  Int16 ioff;
  Int16 insertions=0;
  Int16 offxstring=0; // keep compiler happy
  Int16 showlen;
  Int16 ival;
  Char * S;
  Boolean more;

if (icount > 0)
  { ERRADD("Item ",5);
    ERRINT(icount);
    ERRADD("\x0A", 1);
  };

  ioff = 0;
  c = *(ISRC+15);  // get item type
  if ((c < 'A') || (c > 'X'))
     { ERRADD ("[",1);
       ERRINT(c);
       ERRADD ("]",1);
       c = '?';
     };
  ilen = 0x0F & (*(ISRC+14));
  ilentrue = ilen;
  if (ilen > 14)
     { ilentrue = *((Int16 *)(ISRC+0));
       ioff = *((Int16 *)(ISRC+2));
     };
  ERRADD(&c, 1);  // show type
  ERRADD("=", 1);

  if (ioff) // if eXtended
```

```
    { ERRADD("[",1);
      ERRINT(ilentrue);
      ERRADD(",",1);
      ERRINT(ioff);
      if (c == 'X')  // if type X:
         { insertions = *((Int16 *)(ISRC+4));
           offxstring = *((Int16 *)(ISRC+6));
           ERRADD(",",1);
           ERRINT(insertions);
           ERRADD(",",1);
           ERRINT(offxstring);
         };
      ERRADD("]",1);
    } else // otherwise simply show length
    { ERRINT(ilen);
    };

  S = ISRC;
  if (ilen > 14)
     { S = STACKSTRING+ioff;
     };
  ERRADD("\x0A",1);
  switch (c)
    { case 'V':
      case 'N':
      case 'D':
      case 'T':
      case 'S':
      showlen = ilentrue;
      more = 0;
      ERRADD(S, showlen); // display string
      if (more) { ERRADD("..",2); }; // signal stuff left out
      break;

      case 'I':
      ival = *((Int32*)(S));
      // watch out but if we store std on ARM, still ok!
      ERRINT(ival);
      break;

      default:
      ERRADD("...",3);
    };

  ERRSHOW(0);  // display final string

  if (c != 'X')
     { return 1;
```

```
    };

  if (! recur)  // one deep only
     { ERRmsg(ErDeepRecurStkShow);
       return 0;
     }; // error: X inside X!

  Char * pX;
  pX = STACKSTRING + offxstring; // first x-item
  while (insertions > 0)
    { ERRADD("->",2); // indicator at start of x-item!
      ShowStackItem(pX, insertions, 0);  // recur=0
      pX += 16;  // next x-item
      insertions --;
    };
  ERRADD("INTO:" "\x0A" ,6); // 'insertee'!
  ShowStackItem(pX, 0, 0);
  return 1; // ok
}
```

5.21 More utilities — reorganize me

5.21.1 PeekLengthPlus

Return the length of the current stack item *with one added to it*! On failure return zero.

```
Int16 sql::PeekLengthPlus()
{  Int16 mylen;
   Int16 ok;
   mylen = 2048; // max length!
   ok = StackPeek(SCRIPTLIBCODE, STACK,
                  STACKSTRING, 0, &mylen);
   if (ok < 0)
      { ERRmsg(ErBadLengthPeek);
        return 0;  //fail
      };
  return 1+mylen;
};
```

5.21.2 QuickBy

Easily perform a 'bypass' invocation, which is quicker than a standard one.

```
Int16 sql::QuickBy(Int16 code)
{ return Bypass(SCRIPTLIBCODE, code, 0,0,0, STACK, STACKSTRING);
}
```

5.21.3 PeekType

Determine type of item on stack. Ugly. We discard `plen`.

```
Int16 sql::PeekType()
{ Int16 plen = 2048;  // arbitrary
  return StackPeek(SCRIPTLIBCODE, STACK, STACKSTRING, 0, &plen);
}
```

6 Main section

This program largely conforms to standard rules for creating a PalmOS program in C++. (We cheat a little when we create our instance of the `sql` class, however). We start with the standard *PalmOS.h* header include, add a few headers of our own, and then toss in our own carefully constructed sql hierarchy (represented in *sql.hpp*), as well as headers for our library functions.

```
#include <PalmOS.h>
#include "palmsql3.h"
#include "pain5.h"
#include "sql.hpp"
#include "err/ERRDEBUG.h"
#include "scripting/SCRIPTING.h"
#include "numeric/NUMERIC.h"
#include "sql3/SQL3.h"
#include "idx/IDX.h"
#include "cache/cache.h"
#include "MathLib.h"
```

Next we have a few ugly globals, really just a box for the instance of our `sql` class, and a clumsy hack which references the most recent control used:

```
sql *mySQL3;
Int16 MOSTRECENTCONTROL;
```

6.1 The PilotMain function

The PilotMain function is pretty standard, launching our application (PainStartApplication), and then in the usual clumsy Palm/C++ fashion, polling for events with our PainEventLoop (Section 6.3). A tiny wrinkle we introduced is the `more` variable, which allows us to decide if we wish to terminate (PainStopApplication) or not.

```
UInt32 PilotMain(UInt16 cmd, void *cmdPBP,
                 UInt16 launchFlags)
{ Boolean more = 1;

  UInt16 cardNo;
  LocalID dbID;
  SysNotifyParamType *notifyP;
  /// SysAlarmTriggeredParamType * trigP;
  SysCurAppDatabase(&cardNo, &dbID);  // *sys*

  if (cmd == sysAppLaunchCmdNormalLaunch)
```

```
    { AlmSetAlarm(cardNo, dbID, NULL, 0, true);  // turn off, in case..
      SysNotifyRegister( cardNo, dbID, // trap sleep
         sysNotifySleepNotifyEvent, NULL,
         sysNotifyNormalPriority,NULL); // no user data
      if ( PainStartApplication() )
          { while (more)
              { PainEventLoop (cmd, cmdPBP, launchFlags);
                more = PainStopApplication();
              };
          };
    };

Int16 isSubCall = ( ( launchFlags & sysAppLaunchFlagSubCall ) != 0 );
if (isSubCall)
   {
       if (cmd == sysAppLaunchCmdNotify) // NB type is accessed as follows:
          { notifyP = (SysNotifyParamType *)cmdPBP;
            if (notifyP->notifyType == sysNotifySleepNotifyEvent)
               { // here set alarm to wake us up:
                 AlmSetAlarm(cardNo, dbID, NULL,
                 TimGetSeconds()+MYTIMEOUTSECONDS, true);  // *sys*
               };
          }; // return 0 anyway..

     if (cmd == sysAppLaunchCmdAlarmTriggered)
        { // time out, so force exit!
          EventType event;
          /// trigP = (SysAlarmTriggeredParamType *)cmdPBP;
          /// if (trigP->ref) // if NOT null, it's after 30s wait for true exit
             /// { event.eType = appStopEvent; // should be 'fatal' (0);
             /// } else // it's the preparatory alarm that sets up the 'Timeout!
             /// {
          event.eType = MyTimeoutEvent; //
             ///   // the problem is the locked screen which impedes continuatio
             /// }; // I doubt whether  CtlHitControl  will contribute anything.
          EvtAddEventToQueue( &event ); // enqueue!
        }; // return 0 anyway..
   } else // here if alarm, and NOT in app, reload it: sysUIAppSwitch
   { // http://www.mail-archive.com/palm-dev-forum@news.palmos.com/msg18347.html
     if (cmd == sysAppLaunchCmdAlarmTriggered)
        { // do nothing but return (?); do NOT purgeAlarm
          SndPlaySystemSound(sndAlarm); // sys
        };
     if (cmd == sysAppLaunchCmdDisplayAlarm)
        { SysUIAppSwitch (cardNo, dbID, sysAppLaunchCmdNormalLaunch, NULL);
        };
   };
return 0;
```

```
}
```

6.2 Starting and stopping

Let's look at the start and stop applications in more detail. First, kicking things off, (provided the PalmOS version isn't too primitive), we load several libraries and intialise the sole instance of our `sql` class by invoking the function PainMakeSql (Section 6.7). We can then create and enter the main menu, provided its specification exists in the database which has already been imported to the PDA. The library routines are largely self-explanatory. First we check the version and fail if not adequate.

```
static int PainStartApplication(void)
{ UInt32 romversion;
  Int16  ok;
  Int16 fail;
  UInt16 MathLibRef;

  FtrGet(sysFtrCreator, sysFtrNumROMVersion, &romversion);
  if (romversion < sysMakeROMVersion(3,5,0,sysROMStageRelease,0))
     { WinDrawChars("Bad ROM version!", 16, 20, 50);  // *sys*
       return 0;                          // fail
     };

  UInt16 scriptlibcode = LoadLibrary("SCRIPTING Library", 'ScLi');
  if (scriptlibcode)
     { SCRIPTINGOpen(scriptlibcode);
       MathLibRef = LoadLibrary(MathLibName, MathLibCreator);
       fail = MathLibOpen(MathLibRef, MathLibVersion);
       if (fail)
          { DoDebug("MathLib?",fail);
            return 0;
          } else
          { ScriptMathLib(scriptlibcode, MathLibRef);
          };
     } else
     { DoDebug("ScriptLib?",-1);
       return 0;
     };

  UInt16 elibcode = LoadLibrary ("ERRDEBUG Library", 'ErLi');
  if (elibcode)
     { fail = ERRDEBUGOpen(elibcode);
       if (fail)
          { DoDebug("ErrLib?", fail);
            return 0;
```

```
          };
     };

  UInt16 numericcode = LoadLibrary ("NUMERIC Library", 'NuLi');
  if (numericcode)
     { NUMERICOpen(numericcode);
     } else
     { DoDebug("NumLib?",numericcode);
       return 0;
     };

  UInt16 sql3code = LoadLibrary ("SQL3 Library", '4sql');
  if (sql3code)
     { SQL3Open(sql3code);
     } else
     { DoDebug("SqlLib?",sql3code);
       return 0;
     };

  UInt16 consolecode = LoadLibrary ("CONSOLE Library", 'CnLi');
  if (consolecode)
     { CONSOLEOpen(consolecode);
     } else
     { DoDebug("ConLib?",consolecode);
       return 0;
     };

///  UInt16 idxcode = LoadLibrary ("IDX Library", 'Xsql');
///  if (idxcode) // 0 = failed
///     { IDXOpen (idxcode);
///     } else
///     { DoDebug ("IdxLib?", idxcode);
///       return 0;
///     };
    UInt16 idxcode = 0;

  UInt16 cachecode = LoadLibrary ("Cache Library", 'CsQl'); //
  if (cachecode) // 0 = failed
     { CACHEOpen (cachecode);
     } else
     { DoDebug ("CacheLib?", cachecode); // nasty.
       return 0;
     }; // note need to unload these!
```

We have removed the indexing library (which provided indexing of SQL databases on the PDA) as, counter-intuitively it *doesn't* speed things up owing to the crippling slowness seen when the indexes are found and opened on the PDA! Note the unloading of all of the above libraries within KillUtility (Section 4.4.2). We make

an instance of our `sql` class and enter the MAIN menu:

```
  if (! PainMakeSql(elibcode, scriptlibcode,
            numericcode, sql3code, consolecode,
            idxcode, cachecode))
     { return 0;                          // fail
     };

  ok = mySQL3->RestorePriorState();
  if ( ok > 0)
     { return 1;
     };
  if (ok < 0)
     { DoDebug("?restore", ok);
     };

  ok = mySQL3->EnterMenu("MAIN",4);       // see sql.hpp
    // for now, ignore value in 'ok.
  return 1;                               // ok.
}
```

For details of how we create the `sql` instance, see the PainMakeSql function below (Section 6.7). Exiting the pain database application is far simpler. We merely confirm that the user wishes to leave (ultimately returning 1 to be placed in the `more` variable in PilotMain above), and close down if they do. See PainKillSql below for details (Section 6.7.1).

```
static Boolean PainStopApplication(void)
{  Int16 debugdepth;
   Int16 ok;
  UInt16 cardNo;
  LocalID dbID;
  SysCurAppDatabase(&cardNo, &dbID);  // *sys*

   ok = mySQL3->IsTimeout(); // have we timed out?
   if (ok) // if timed out...
      {
        AlmSetAlarm(cardNo, dbID, NULL, 0, true); // turn off alarm
        PainKillSql();
        return 0; // quit.
      };

   debugdepth = mySQL3->FetchMenuDepth();
   if (debugdepth <= 2)
      { if (! mySQL3->Confirm("EXIT?"))
           { return 1;  // don't
           };
```

```
        AlmSetAlarm(cardNo, dbID, NULL, 0, true); // turn off alarm
        PainKillSql();
        return 0; // signal "QUITTING"..
      }

   if (! mySQL3->Confirm("Leave for a moment?"))
      { return 1;  //
      };

   // here will write backup 'file' so can resume!
   ok = mySQL3->WriteRestorationData();
   if (ok < 1)
      {DoDebug("?Cache", ok);
      };
   AlmSetAlarm(cardNo, dbID, NULL,
      TimGetSeconds()+MYTIMEOUTSECONDS, true); // force restart!!
   PainKillSql();
   return 0; // signal "QUITTING"..
}
```

It might be appropriate to check for a recent COMMIT (when/if this functionality is present) and warn about data loss if data committment is still pending.

6.3 Event loop

Our event processor is pretty standard for PalmOS. It is only entered once PainStartApplication has been completed, which is just as well, as it ultimately uses functions from the newly created instance of `sql` (namely mySQL3). We maintain the normal sequence of EvtGetEvent, SysHandleEvent, MenuHandleEvent, our user function (PainHandleEvent), and finally FrmDispatchEvent.

```
static UInt32 PainEventLoop(UInt16 cmd, void *cmdPBP,
                            UInt16 launchFlags)
{   EventType   e;
    UInt16      err;
        // handle events:-
          do { EvtGetEvent(&e, evtWaitForever);          // *sys*
               if (!SysHandleEvent(&e))                  // *sys*
                  { if (!MenuHandleEvent(NULL, &e, &err))// *sys*
                       { if (! PainHandleEvent(&e)) // our own
                            { FrmDispatchEvent(&e);      // *sys*
                  }    }    }
             } while (e.eType != appStopEvent);
  return 0;
}
```

The do loop whirls around until an appStopEvent is posted, but we can still recover and re-enter, depending on the outcome of PainStopApplication (See section 6.1). Let's look at PainHandleEvent. There are several different types of event we must cater for, firstly the loading of a form:

```
Int16 PainHandleEvent(EventPtr e)
{   FormPtr     frm;
    Int16       formId;
    Boolean     handled = false;
    Int16       response;
    UInt16      cardNo;
    LocalID     dbID;

    if (e->eType == frmLoadEvent)
       {  formId = e->data.frmLoad.formID; // form ID number
          frm = FrmGetFormPtr(formId);     // *sys*
          if (! frm)
             { mySQL3->ERRDisplay("No form", 0000); //[???]
             };
          FrmSetActiveForm(frm);                        // *sys*
          FrmSetEventHandler(frm, PainFormHandleEvent); // *sys*
          handled = true;
        };
```

The two required functions are to set the active form, and then allocate the form an event handler (as noted below, simply a stub). Once the form is active, the OS sends events to this form. Next, we deal with the actual opening of a form, a signal that we should draw it:

```
    if (e->eType == frmOpenEvent)
       {  formId = e->data.frmLoad.formID; // get form ID
          frm = FrmGetFormPtr(formId);     // *sys*
          FrmDrawForm(frm);                // *sys*
          handled = true;
        };
```

Now let's see what happens with poptriggers. When we click on a poptrigger, the sequence of eType events which occurs is 7,8,9 and 14. Events 7 and 8 are ctlEnterEvent and ctlExitEvent, respectively.[46] We use these two events to identify the MOSTRECENTCONTROL and store its value, a nasty global which could better be stored somewhere safe within the `sql` class itself!

[46]Technically we should only ever refer to their symbolic names, although it's highly unlikely that the numeric values will be changed by PalmOS!

In the more interesting case where a popSelectEvent has actually been reported (i.e. an item was selected from a pop list, code 14), we respond with the eponymous Respond function, using the previously recorded control ID (MOSTRECENTCONTROL). We still return `false` so that the control will be redrawn! A bit of a hack, really.

```
if ((e->eType == 7) || (e->eType == 8))
     //  ctlEnterEvent,  ctlExitEvent,
   { MOSTRECENTCONTROL = e->data.ctlEnter.controlID;
     return 0;
   };

if (e->eType == popSelectEvent)  // value = 14
   { //lstid = e->data.popSelect.listID;
     mySQL3->Respond(MOSTRECENTCONTROL, 0, LISTNOTCTL, e);
     return(false); // allow control to be drawn!
  };
```

Next, we deal with other true control responses (which don't involve responses to text fields, as these aren't true controls). We're talking about buttons, checkboxes, and pushbuttons.[47] Text fields are still peripherally involved however, for we use our tricky function DidLastFieldChange to check whether there has been any recent fiddling with a text field, so that we can respond to these alterations *before* we respond to the button or whatever!

Otherwise, we carefully *ignore* fiddling with poplists (awaiting the popSelectEvent handled above!)

Next, we look for the on/off attribute of a control. This is nasty, as saying something like:

```
\begin{verbatim}
  iamon = e->data.ctlEnter.on;
```

...simply doesn't work! Our ctlSelect strategy seems to work, but is it future proof? [CHECK ME] We had to re-define the structure in *palmsql3.h*, which is worrying. (See Section 7.2.2).

The last, and most important component of this section is actually responding to the events using all of the power of our `sql` class. Again, as above, we invoke our Respond function, which deals with much of our user interaction. The brutal 'quit' response enjoins us to post an appStopEvent, which in turn allows the program to die gracefully.

```
 if (e->eType == ctlSelectEvent)   // = 9
    {
```

[47] And also poplists!

```
        if (mySQL3->DidLastFieldChange())
           { response = ActionField(e); // [20080510]
             if (response == IMWAITING)
                { return 1; //handled!
                };
           };

        Int16 myctl;
        myctl = e->data.ctlEnter.controlID;
        Int16 ctype = mySQL3->FindWidgetType(myctl);
        if ( ctype == LISTNOTCTL) // poplist
           { return 0; // don't handle, wait for 14 above!
           };

        Boolean iamon;  // on/off value?
        struct ctlSelect * cts;
        cts = (struct ctlSelect *) & e->data;
        iamon = cts->on;

        response = mySQL3->Respond(e->data.ctlEnter.controlID,
                           iamon, ctype, e);
          // (trap button click etc!!)
        if (response == DIEYOUBASTARD)   // quit?
           { e->eType = appStopEvent;    // die!
             return 0;
           };
        return 1; // handled
      };
```

What about fields? If we check for an altered field on any other control click, we can respond and fire off the necessary 'text interrupt'. But what if a text field changed? Here too we have no signal that we 'left' but we do have fldEnterEvent. So if we encounter this, we must:

1. fire off the 'text interrupt';
2. Clear ACTIVEFIELD in *utility.hpp*, so that when we 'leave' the current field, we can note the change and fire off the next 'text interrupt'.

```
if (e->eType == fldEnterEvent)
     { if (mySQL3->DidLastFieldChange())
          { response = ActionField(e); //[20080510]
            if (response == IMWAITING)
               { return 1; //handled!
               };
          };
       Int16 myfldid;
```

```
FieldType * thisfld;
myfldid = e->data.fldEnter.fieldID;
thisfld = e->data.fldEnter.pField;
mySQL3->SetActiveField(thisfld, myfldid);
  // or just pass e ?

// here we determine whether a field is editable.
// if not it MUST BE a clickable text 'label':
// thisfld is a pointer to the field, so we can
//
FieldAttrType fat;
FldGetAttributes(thisfld, &fat);   // must w_ this!
if (! fat.editable) // PalmOs unclear, but 'legal'!
   {
     if (myfldid & 1) // bit flag says 'IS clickable label'!
        { response = mySQL3->Respond(e->data.ctlEnter.controlID,
                  0, LABELNOTCTL, e); // ?
        };
   } else
   { if ((myfldid & 7)==4) // NUMBER 'PICKER'
        { SysKeyboardDialog( kbdNumbersAndPunc );
        };
     if ((myfldid & 7)==2) // IF A DATE (DATE PICKER):
        {
          Int32 secs;
          DateTimeType dtt;
          secs = TimGetSeconds(); // hmm=2026 ???
          TimSecondsToDateTime (secs, &dtt);
          // ALTERNATIVELY might obtain value from field???
         // Int16 sd_month;
         // Int16 sd_day;
         // Int16 sd_year;
         // sd_month = dtt.month;
         // sd_day = dtt.day;
         // sd_year = dtt.year;
         MemHandle txH;
         Char * ptx;
         // SelectDay (selectDayByDay,
         //   &sd_month, &sd_day, &sd_year, "Select date");
          if (SelectDay (selectDayByDay,
            &dtt.month, &dtt.day, &dtt.year, "Select date"))
             { txH = MemHandleNew(11); // YYYY-MM-DD
               ptx = (Char *) MemHandleLock(txH); // MHL13
               Write4Digits(ptx,   dtt.year);
               Write2Digits(ptx+5, dtt.month);
               Write2Digits(ptx+8, dtt.day);
               *(ptx+4) = '-';
               *(ptx+7) = '-';
```

```
                    *(ptx+10) = 0x0;
                    MemHandleUnlock(txH);
                    FldSetText (thisfld, txH, 0, 11);
                    FldDrawField (thisfld);
                  };
              };
          };

    };
```

The following few lines encapsulate a lot of trouble. If we respond to exiting a text field, and our response script contains a MENU command (for example, MENU(0) to reload the menu), then this command would delete the current button, causing a PalmOS crash. Our solution is moderately cunning — we create our own peculiar customised event called 'MyMenuEvent'. The way we implement a MENU command is to post this event to PalmOS (Section 5.4.5), and then terminate the script but leave the stack unharmed. When we subsequently encounter the event in our main loop (here!) we can enter NewMenu.

We added another wrinkle in May 2008 (v 0.95). Because we now wake up the PDA and ask to exit if the PDA has been sleeping for over 15 minutes, we need to prevent irritating repeat 'wake up calls' if the user has already woken the PDA again. Now our solution using sysNotifyLateWakeupEvent was dismally unsuccessful (It prevented the wake-up — why?) so we simply turn off the alarms when we encounter a menu event — surely nobody will stay in one menu for 15 minutes?

```
if (e->eType == MyMenuEvent)
   { SysCurAppDatabase(&cardNo, &dbID);  // *sys*
     AlmSetAlarm(cardNo, dbID, NULL, 0, true); // turn off alarm 20080518
     mySQL3->NewMenu();  //
   };

if (e->eType == MyTimeoutEvent)
   { mySQL3->ForceTimeout(1);
   ///  // if not reversed will force death..
   ///  // first, give 20s of grace.. otherwise PainStopApplication terminates
   ///  SysCurAppDatabase(&cardNo, &dbID);  // *sys*
   ///  AlmSetAlarm(cardNo, dbID, 1, TimGetSeconds()+20, true);  //try +20s  *s
   ///  // user value of 1 forces death unless countermanded!
   ///  // next confirm..
   ///  Int16 ok = mySQL3->Confirm("Timed out! Continue using PainForm?");
   ///  AlmSetAlarm(cardNo, dbID, NULL, 0, true);  // turn off alarm but..
   ///  if (ok)
   ///     { mySQL3->ForceTimeout(0);  // countermand (whew!)
   ///       // if reverse logic (! mySQL->Confirm..) THEN defaults to continue
```

```
      ///       // because Confirm is forced to 1 by PalmOS on timeout!!
      ///     } else // hasten demise..
      ///     {
        e->eType = appStopEvent;
        EvtAddEventToQueue( e ); //
      ///     };
        return (1); // handled
      };

   if (e->eType == frmTitleEnterEvent) // 29: click on menu bar at top
      {
         if (mySQL3->DidLastFieldChange()) // otherwise last change is lost!
            { response = ActionField(e); // [20080510]
              if (response == IMWAITING)
                 { return 1; //handled!
                 };
            };
         response = mySQL3->Respond(0, 0, MENUITSELF, NULL); // ??
         // get iResponse for the menu!!
      };

/* ****************************************
// Only Use this code to 'spelunk' PalmOS events:
    if(  (e->eType != MyMenuEvent)
     //  &&(e->eType != fldEnterEvent)
       &&(e->eType != ctlSelectEvent)
       &&(e->eType != popSelectEvent)
       &&(e->eType != ctlEnterEvent)
       &&(e->eType != ctlExitEvent)
       &&(e->eType != frmOpenEvent)
       &&(e->eType != frmLoadEvent)
       &&(e->eType != winEnterEvent)// common events
       &&(e->eType != winExitEvent) // found
       &&(e->eType != penUpEvent)   // using
       &&(e->eType != penDownEvent) // spelunking!
       &&(e->eType != 0) // ...

       &&(e->eType != 0) // fill in events to NOT notify here
       &&(e->eType != 0) // eg. 29, 30 (and 4) are posted on menu bar click!
       &&(e->eType != 0) // 4 occurs with click on graffiti area/MENU button/etc.
       &&(e->eType != 0) // 22 is click on Applications/Calc button
       // 15 appears to be field-related.
       )
       { DoDebug("debug EVENT",e->eType);
       }; // end spelunking area.
*/ ////////////////////////////////////////

/* The following is good for key movements:
```

```
   if (e->eType == keyDownEvent)
      { Int16 c = (Int16) e->data.keyDown.chr; // or whatever is pageUpChr or page
        // perhaps also look at TxtCharIsHardKey macro??
        DoDebug ("Key is",c); // 11 is up, 12 is down.
      };

*/

    return handled;
}
```

And that's the end of our event handler. Here's the subsidiary ActionField function already referred to. GetFieldText provides both a pointer to the text and its size. We then actually push the string onto the sql internal stack, and 'Respond'.

```
static Int16 ActionField(EventPtr e)
{ Char *      fldptr;
  Int16       fldlen;
  Int16       fldid;
  fldptr = mySQL3->GetFieldText(& fldlen); // new value!
  // might check nonzero ?!
  mySQL3->PushString(fldptr, fldlen);  //
  fldid = mySQL3->GetFieldID();
  return (mySQL3->Respond(fldid, 0, 0, e)); // null event!
}
```

A few minor functions:

```
static void Write2Digits(Char * P, Int16 i)
{ *(P) = '0' + i/10;
  *(P+1) = '0' + i%10;
};

static void Write4Digits(Char * P, Int16 i)
{ Write2Digits (P, i/100);
  Write2Digits (P+2, i%100);
}
```

6.4 Individual form event handling

Although we set up a form event handler (as recommended) at present it's simply a stub, as everything eventually trickles through to the above generic event handler, provided PainFormHandleEvent returns false!

```
static Boolean PainFormHandleEvent(EventPtr event)
{ Boolean     handled = false;
  return(handled);
}
```

6.5 A debugger function

This primitive function simply displays an error message string together with a number. We have a hard-coded form (DebugAlert) to allow display of the message.

```
static void DoDebug(const Char * msg, Int16 nmbr)
{ MemHandle memH;
  MemPtr pStr;
  memH = MemHandleNew(maxStrIToALen);            //*sys*
  if (! memH)
     { return (Aaagh(msg, "1"));
     };
  pStr = MemHandleLock(memH);                    // *sys* MHL14
  if (! pStr)                                    //
     { return (Aaagh(msg, "2"));
     };
  StrIToA((Char *)pStr, nmbr);                   // *sys*
  FrmCustomAlert(DebugAlert,msg,(Char *)pStr,"");// *sys*
  if (MemPtrUnlock (pStr) != errNone)            // *sys*
     { return;
     };
  if (MemPtrFree (pStr) != errNone)              // *sys*
     { return;
     };
 return;
}
```

Here's the 'solution' if DoDebug itself fails:

```
static void Aaagh (const Char * msg, const Char * msg2)
 {
   FrmCustomAlert(DebugAlert,msg,msg2,"Fatal");// *sys*
 }
```

6.6 Library loader

The library loader uses standard PalmOS system functions to locate and find the library, given the name and an associated identifier code. We should rationalise the latter [FIX ME].

We locate the library database, determine its type (checking for error), and then load it. All a bit clumsy, but heck, it works.

```
static UInt16 LoadLibrary (Char * libname, Int32 SILLYCODE)
{
```

```
  UInt16 libcode=0;
  LocalID lid;
  UInt32 dbtype=0;
  lid = DmFindDatabase (0, libname);
  if (! lid)
     { DoDebug ("No LIB found",0);
       return 0;
     } else
     { DmDatabaseInfo (0, lid,
                       0, 0, 0, 0,
                       0, 0, 0, 0,
                       0, &dbtype, 0);
     };
if (! dbtype)
   { return 0;
   };
Err error;
  error = SysLibFind(libname, &libcode);  // loaded?
  if (error == errNone)
     { return libcode; // already loaded
     };
  error = SysLibLoad (dbtype, SILLYCODE, &libcode);
  if (error != errNone ) // clumsy. Use if(error)
     { DoDebug("Failed to load library", error);
       return 0;
     };
  return libcode;
}
```

6.7 Creating an SQL instance, and deleting it

The following section creates an instance of the `sql` class, so can invoke all the contained functions! In C++ we should simply create a *new* instance of sql. However, in PalmOS, use of their peculiar version of 'new' has all sorts of implications.

Anyway, here's our version, which also accepts the handles of open libraries, and then passes these to mySQL3 once instantiated. We also pass necessary callback functions — note the different ways we can do this!

```
static Int16 PainMakeSql (UInt16 elibcode, UInt16 scriptlibcode,
                          UInt16 numericcode, UInt16 sql3code,
                          UInt16 consolecode,
                          UInt16 idxcode, UInt16 cachecode) // lib handles..
{ MemHandle   memH=0;
  MemPtr      memP=0;
  memH = MemHandleNew(sizeof(sql)); // *sys*
  if (! memH)
     { return 0; // fail
```

```
    };
  memP = MemHandleLock(memH); // *sys* MHL15
  if (! memP)
    { return 0;
    };  //memP pointer to locked new memory.
  mySQL3 = (sql *) memP; // easy peasy!

  Int16 fail;
  fail = (mySQL3->ErrStart(elibcode));
  if (fail) // 0=ok!
    { DoDebug("Ini?", fail);
    };
  DmComparF * janet;  // static function!
  janet = (DmComparF *) CompareRowKeys;

  fail = mySQL3->SetupUtility((ListDrawDataFuncPtr)
                       myLISTDRAWER, //
                       janet,        // callbacks
                       scriptlibcode,
                       numericcode,
                       sql3code,
                       consolecode,
                       elibcode,
                       idxcode,
                       cachecode);
  if (fail)
    { DoDebug("Util?", fail);
      fail = mySQL3->KillUtility();
      DoDebug("End", fail);
      return 0; // 2008-05-06
    };

  fail = mySQL3->Kickoff();
  if (fail)
    { DoDebug("Start?",fail);
      return 0; // 2008-05-06
    };

return 1; // success
}
```

A possible alternative which we discovered later and haven't really explored thoroughly is to use something along the lines of:

```
#include <new>
...
char *str = new (std::nothrow) char[50];
```

6.7.1 Killing the program

Finally, we have the corresponding cleanup function. Much of the dirty work is done by the mySQL3 functions Cleanup, KillUtility and ErrEnd, which remove irritating residua of the various classes.

```
static Int16 PainKillSql()
{ MemPtr killSQL;
  Int16 ok;
  Int16 fail;

  mySQL3->Cleanup();
  fail = mySQL3->KillUtility(); // 'destructor': 0=ok.
  if (fail)
      { DoDebug("?Util",fail);
      };

  ok = mySQL3->ErrEnd();
  if (ok)
     { DoDebug("?End", ok);
     };

  killSQL = (MemPtr) mySQL3;
  if ( (MemPtrUnlock (killSQL) != errNone) // *sys*
     ||(MemPtrFree (killSQL) != errNone)   // *sys*
     )
   { DoDebug("?Mem", 0);
     return 0;
   };

return 1;  // ok.
}
```

It is wise to leave ErrEnd till almost the end, as the Delete function depends on this, but we must invoke ErrEnd *before* we free up the instance of mySQL3, of course.

6.8 Static callback functions

These necessary evils (all C++ callbacks are both evil and necessary) must be defined statically here, and then passed to the dynamically created mySQL3 and its component classes. There are just two: myLISTDRAWER which dynamically draws a list in a listbox, in fact being called repeatedly to draw each blooming line, and the comparison function CompareRowKeys, used for searching through one of our own (sorted) SQL databases.

6.8.1 myLISTDRAWER callback

The PalmOS system designers have changed the rules in more recent OS versions, forcing us into their particular box. Formerly we could pass our own structured list in `**itemsText`, but now we have to play by their rules. Note that the same string is submitted each time, but with a different itemNum. The arguments conform to the PalmOS prototype.

```
static void myLISTDRAWER (Int16 itemNum,
       RectangleType *bounds, Char **itemsText)
{ Char * jj;
  jj = * (itemsText+itemNum);
  Int16 isiz;
  isiz = StrLen(jj); // *sys*(2)
  WinDrawChars(jj, isiz,
       bounds->topLeft.x, bounds->topLeft.y);
  return;
}
```

6.8.2 CompareRowKeys callback

This callback *must* return zero for equality, and -1 if the first key is less (+1 if greater) than the second key. We ignore the values in `other` and subsequent variables. All we are interested in is the 32 bit integers (big-endian format, caveat emptor) at offset +8 within the two records passed to the callback.

```
static Int16 CompareRowKeys (
                 void *rec1, // first rec
                 void *rec2, // second rec
                 Int16 other,
                 SortRecordInfoPtr rec1SortInfo,
                 SortRecordInfoPtr rec2SortInfo,
                 MemHandle appInfoH )
{  Int32 k1;
   Int32 k2;

   k1 = * (((Int32 *)(rec1)) + 2); // NB big-endian
   k2 = * (((Int32 *)(rec2)) + 2); // fix for ARM processor

// marginally faster but more confusing than:
//   k1 = mySQL3->ReadInt32((Char *)rec1+8);
//   k2 = mySQL3->ReadInt32((Char *)rec2+8);

   if (k1 < k2)  {return -1; };  // clumsy
   if (k1 > k2) { return 1; };
   return 0;
}
```

In the above, the +2 refers to moving two dwords to the right, equivalent to eight bytes, as we first cast to an Int32 pointer.

7 The Makefile, and other frills

The following files are all vital for the creation and functioning of our program.

7.1 The Makefile

Under UNIX, Linux or Cygwin, the all-important makefile determines how components are put together, including compiling and linking. We provide an introduction to makefiles elsewhere (Section 8.2.1). Here's the makefile for our project:

```
CC = m68k-palmos-gcc
CFLAGS = -Wall -pedantic -g -O2 -fno-exceptions -fno-rtti

all: pain5.prc
ls -l pain5.prc

pain5.prc: pain5.def pain5 *.bin
build-prc -o $@ pain5.def *.bin pain5

*.bin: pain5.rcp pain5.h painrcp.h
pilrc pain5.rcp

pain5: pain5.cpp pain5.h painrcp.h palmsql3.h err.hpp wraps.hpp \
       utility.hpp sql.hpp
$(CC) $(CFLAGS) -o $@ pain5.cpp

clean:
rm -f *.grc *.prc *.bin pain5
```

Make knows to fill in the definitions of CC, the compiler, and CFLAGS, the compilation options, when it encounters these names in parenthesis, preceded by a dollar sign. Reading from bottom to top (ignoring the *clean* section, which is simply used as *make clean*, to start anew), the compiler compiles the given files, pilrc is the resource compiler for our tiny *pain5.rcp* file (thus generating a lot of .bin binary files), and `build-prc` creates the actual PalmOS PRC program using the binary files, the pain5 object generated by the compiler, and the .DEF file.

Let's look at some of the files needed. In addition to the files mentioned below, take note that the trivial bitmap image file `pain.bmp` is required to create the final `pain5.prc` program file!

7.2 Header files

There are several header files:

- *pain5.h*
- *palmsql3.h* and its companion, *palmsql3A.h*
- *painrcp.h*

In addition it's important to note that *library header files* are also required. The program *will not work* without the libraries, and will not *compile* without the library header files. These are covered in detail in the next document in the series [INSERT REFERENCE HERE]. Required library header files are:

- *err/ERRDEBUG.h*
- *scripting/SCRIPTING.h*
- *numeric/NUMERIC.h*
- *sql3/SQL3.h*
- *console/CONSOLE.h*

7.2.1 pain5.h

The first, *pain5.h*, is a straightforward listing of the various functions in the main C++ file, *pain5.cpp*. All the functions are static (this property is particularly important for the callback functions).

```
#include "painrcp.h"

static int PainStartApplication(void);
static Boolean PainStopApplication(void);
static UInt32 PainEventLoop(UInt16 cmd, void *cmdPBP,
   UInt16 launchFlags);
Int16 PainHandleEvent(EventPtr e);
static Boolean PainFormHandleEvent(EventPtr event);
static void DoDebug(const Char * msg, Int16 nmbr);
static void Aaagh (const Char * msg, const Char * msg);
static void Write2Digits(Char * P, Int16 i);
static void Write4Digits(Char * P, Int16 i);
static void myLISTDRAWER (Int16 itemNum,
   RectangleType *bounds, Char **itemsText);
static Int16 CompareRowKeys ( void * r1, void * r2, Int16 o,
   SortRecordInfoPtr p1, SortRecordInfoPtr p2, MemHandle aih);
```

```
static Int16 PainKillSql();
static Int16 PainMakeSql (UInt16 elibcode, UInt16 scriptlibcode,
   UInt16 numericcode, UInt16 sql3code, UInt16 consolecode,
   UInt16 idxcode, UInt16 cachecode);
static UInt16 LoadLibrary (Char * libname, Int32 SILLYCODE);
static Int16 ActionField(EventPtr e);
```

7.2.2 palmsql3.h

The two palmsql3 files are a lot longer. They contain many constants. Here's the first, *palmsql3.h*. We start with several useful structures. The first of these is specified by PalmOS, but isn't defined in the current headers, so we re-define it![48] It is used in *pain5.cpp*.

```
#include "palmsql3A.h"

struct ctlSelect {
UInt16 controlID;
struct ControlType *pControl;
Boolean on;
UInt8 reserved1;
UInt16 value;
} ctlSelect;
```

We next define our own structure for a 'list of lists', ListLink:

```
struct ListLink  { Int16 id;
                   Char * txtlist;
                   ActualList * alist;
                   MemHandle keepme;
                   Int16 choices;
                   Int16 popper;
                   ListLink * next;
                 };
```

In the ListLink structure `id` is the ID of the list item associated with this list, `txtlist` was a pipe-delimited list but has now been transformed into an ugly PalmOS packed array of strings, `keepme` is a nasty little PalmOS index into the packed array, and `choices` is the number of items in the list. We then point to the next link using (wait for it!) `next`, unless we're at the end of the linked list, which we signal using a NULL. The strange 'popper' is the id of the associated poplist!

[48] A little worrying, really.

In order to be able to access the information in the original (pipe-delimited) list from which the ListLink structure is derived, we also create a subsidiary structure, the ActualList:

```
struct ActualList { Int16 count;
                    Char * auxlist;
                    Int16 auxlen;
                  };
```

The ActualList stores a count of the number of times it is referenced by a ListLink, as many ListLinks can refer to the same ActualList. Destruction of a ListLink forces a decrement of the value in count, and when this becomes zero, the ActualList is also destroyed!

Nasty text lists

In order to make clickable text we have to go through quite a rigmarole. We use a *field* to identify the click (making the field non-editable), and store the text in a simple linked list of 'TextNodes':

```
struct TextNode  { Int16 id;
                   Char * txt;
                   TextNode * next;
                 };
```

Next we have the WIDGET structure used to store information about GUI items:

```
struct WIDGET {
  Int16 mytype;
  Int32 uid;
  Int16 dbid;
  Int32 V;
  WIDGET * next;
              };
```

We keep a record of the type of widget (e.g. a button) in `mytype`, the unique local ID of the widget in `uid`, the actual database ID of the component (`dbid`), the V value, which is the *associated* database code now attached to the widget, and finally a `next` pointer. The V value is always unique within the database (a primary key in some table)!

In the following section we have more constants than we probably need. The first is vitally important — our unique code for a customised user event which signals the entry into a new menu (or indeed, return from a menu to a preceding menu).

```
#define MyMenuEvent (enum eventsEnum)(((UInt16)firstUserEvent)+1)
#define MyTimeoutEvent (enum eventsEnum)(((UInt16)firstUserEvent)+2)
```

The next few constants are display-related, as their names suggest. The 2-pixel values allow for the PalmOS window border, which shouldn't be included in calculations.

```
#define SCREENWIDTH 160
#define SCREENHEIGHT 160
#define MINMENUWIDTH 16
#define MINMENUHEIGHT 8
#define MINXPOS 2
#define MINYPOS 2
#define YSCREENOFF 2
```

We have a few database-related constants, the maximum number of columns in the database, and the maximum size of a header record:

```
#define MAXCOLUMNS 64
#define MAXCOLLEN  4096
```

Note that MAXCOLUMNS is also represented in Sql3Lib.tex (see usage) so we must be careful if we alter this (unwise). Next, some signals (see their usage within the program).

```
#define DIEYOUBASTARD -1234
#define NORMALOUTCOME 1
#define IMWAITING     3
#define IVESTOPPED    2
#define KICKOFFITEMID 2000
```

The value in KICKOFFITEMID should be 2000, or at least divisible by 8. Do *not* alter this value without agonizing long and hard, and knowing exactly what you are doing! Finally, we have a single error code, the sole remnant of a host of previous codes we've now gotten rid of! (This too should go) [FIX ME]

```
#define SqlBadBase                500
```

7.2.3 palmsql3A.h

This header file contains common definitions used by both the main C++ program, and certain libraries (scripting.c and numeric.c). Not all definitions are used by all players.

We define the default card (the main one, code zero), the type of database (DATA) and creator (me), and a variety of simple numeric and string constants.

```
#define MAINCARD 0
#define DBTYPE 'DATA'
#define DBCREATOR 'JovS'
#define sSEPARATOR "\x1F"
#define SEPARATOR '\x1F'
#define MYTIMEOUTSECONDS 900 // 15 min is good
/// #define MYTIMEOUTSECONDS 10 // debug only ???
// #define MAXACTUALROWS 2000
// (inactive: this value should be moved here from all other usages)


#define oSTART  0
#define oTOP    2
#define oMAX    4
#define oFLAGS  6
#define oMARKS  8
```

We also define debugging flags, as these are common to the main program and library routines which write to the console depending on which flag is set in a DEBUGFLAGS variable.

```
#define fDEBUG_ALWAYS 0
    // paradoxically, debug code of zero FORCES debugging
#define fDEBUG_STMT 1
    // tracks scripting
#define fDEBUG_SQL 2
    // tracks sql
#define fDEBUG_SQK 4
    // SQL detail (e.g. sort)
#define fDEBUG_SQJ 8
    // SQL fine detail
#define fDEBUG_WIDGET 16
    // widget creation
#define fDEBUG_DUMP 32
    // dump every stmt
#define fDEBUG_STACK 64
    // stack-dumping
#define fDEBUG_IDX 128
    // debugging of indexing!
#define fDEBUG_AAGH 256
    // major error
#define fDEBUG_CACHE 512
    // debug SQL caching.
```

The following are a bit of a hack. If the parser in SCRIPTING.c identifies a call to an arithmetic routine, it returns a code greater than iA_BASELINE, and the caller must then invoke the numeric library. In fact, we now really only need to

define iA_BASELINE in the common header, and could move the other iA values out to the two libraries (numeric and scripting).

When searching through this documentation for invocations of other iA_ functions, look for iA_BASELINE as comparisons are relative to this.

```
#define iA_BASELINE  10000
#define iA_SUB       10001
#define iA_ADD       10002
#define iA_MUL       10003
#define iA_DIV       10004
#define iA_MOD       10005
#define iA_NEG       10006
```

Next we have *our* aliases for PalmOS controls, as they appear in our SQL code. We intend to code an ordinary text field as 10, a numeric field as 11, and a date field as 12.

```
#define LABELNOTCTL     1
#define BUTTONCTL       2
#define CHECKBOXCTL     3
#define PUSHBUTTONCTL   4
#define POPUPTRIGGERCTL 6
#define FIELDNOTCTL     10
#define DATEFIELD       12
#define NUMBERFIELD     14

#define BADCTL         -1

#define LISTNOTCTL         18
// #define REPEATINGBUTTONCTL 16
// #define SELECTORTRIGGERCTL 15
// #define SLIDERCTL       7
// #define FEEDBACKSLIDERCTL  17
// the above 5 aren't represented in the Perl code.

#define POLYTABLE   8
#define IMATABLE 9
#define IMAMENU  20
#define MENUITSELF 50
```

Penultimately we have a few numeric values which describe the pixel width and height of a PalmOS screen, as well as the height of the PalmOS header on screen (MenuYOffset).

```
#define MenuScrWidth 156
#define MenuScrHt 156
#define MenuYOffset 12
#define MenuMaxHt   MenuScrHt-MenuYOffset
#define MenuLMargin 2
```

The width and height are 156 instead of 160, as we have to allow for the nasty PalmOS menu margin! In addition, when we place items on the screen, we have to introduce a further Y offset to compensate for the approx 12 pixel high header at the top of each menu!

Finally we have a bunch of definitions which are now comfortably internalised in the database (sql) library, and so could profitably be moved there: [FIX ME]

```
#define XVI 16
  // the size of a stack element. Must be 16.
#define BOTTOMSTACK 96
#define BOTTOMSTACKSTRING 16
#define MAXSS 32000
#define STACKCOUNT 2000
#define SMAX XVI * STACKCOUNT
  // maximum size of STACK area
```

7.2.4 painrcp.h

This trivial file is a header associated with the tiny resource file *pain5.rcp*, considered in the following section.

```
#define DebugAlert                  7999
#define MyAlert                     7998
#define MyConfirm                   7997
#define MyAsk                       7996
```

7.3 RCP file

The file *pain5.rcp* contains the scanty controls we create statically (as resources). Almost all controls in our program are created *dynamically* from specifications contained within the database itself. But there are these:

```
#include "painrcp.h"

VERSION "0.90"
ICON "pain.bmp"

ALERT ID DebugAlert
  WARNING
BEGIN
  TITLE "Debug alert!"
  MESSAGE "^3 '^1' (^2)"
  BUTTON "OK"
END
```

```
ALERT ID MyAlert
  WARNING
BEGIN
  TITLE "Note!"
  MESSAGE "^1"
  BUTTON "OK"
END

ALERT ID MyConfirm
BEGIN
  TITLE "Confirm.."
  MESSAGE "^1"
  BUTTONS "NO" "Yes"
END

ALERT ID MyAsk
BEGIN
  TITLE "Enter text"
  MESSAGE "^1"
  BUTTONS "OK" "Cancel"
END
```

You're looking at the trivial bitmap associated with the program, [NOTE TO SELF: (cackle) need to provide this too: fix me] and displayed on the PDA, as well as static resources for debugging, and the ALERT, CONFIRM and ASK functions.

7.4 The DEF file

Last of all we have the important single line file *pain5.def*. Here it is:

```
app { "PAIN DB" JovS }
```

This file gives the program (application) a name, and signals who created it (me again).

8 Appendix: PalmOS programming

The following section comes from some short notes I made a few years ago. It's a brief introduction to PalmOS programming, and how the current program came about.

We will be programming in C++, using current freeware tools for development. We will take the slightly longer route of *not* using MetroWerks CodeWarrior, (or other commercial software) for three reasons:

1. Our way is free;
2. The resource files used by CodeWarrior are inscrutable, so if you're trapped, you're trapped. Not so for the PilRC files we will eventually encounter, although ultimately our use of even these files will be minimal!
3. Although superficially 'easier', in the long run we find CodeWarrior even more cumbersome than just slogging the hard yards at the start!

The route we will take will appear intimidating at first, but is fairly secure. A lot of it initially involves typing in apparently meaningless sequences. These will acquire meaning as you enter them and learn. We will later on find out how to automate much of what we do, and make it painless, but expect some initial discomfort!

A word of caution. If you cannot even open up a DOS box on your Windows system,[49] you should probably not attempt the following. The following description assumes *some* familiarity with computer programming, be it however slight. You will also need dogged determination, and:

1. An understanding of the following terms: ASCII, compiler, URL, ROM, function, byte, word, tab character, and hexadecimal (hex);
2. The ability to find and download files off the Internet (although we will usually provide URLs)

Also note that although the C++ code for the PDA can be compiled using the standard compiler GCC, the *flavour* is somewhat off. A lot of standard C++ library functions, and even primitive things like *new* and *delete* are poorly represented on the Palm, or may cause problems. Memory constraints, especially the size of the heap, limit the utility of useful C++ features such as try/throw/catch, especially in larger PDA programs. Be warned, the experience is somewhat dysphoric!

[49] If you are working under Linux, you just need to do a bit of intelligent surfing, and RTFM! For the Mac, use CodeWarrior.

8.1 PDA programming — an introduction

There are several sources on the Internet which demonstrate how to create an initial Palm application using only freeware. These will change from time to time. A good Google search strategy is variants of:

```
cygwin pilrc pose C++ sdk prc
```

...but be careful! There are popular pages which recommend that you use older versions of Cygwin (like the venerable B20.1), and also have older programs for the Palm. It *is* still possible to use these, but things have changed, and using these older versions will probably result in unhappiness, now or later.

We will get and use the following:

1. Cygwin. This is a Windows-based version of UNIX. Cygwin will allow you to use:

2. PRC-tools — a freeware development environment for Palm OS. As usual, all good things come from open-source environments. Download of this into Cygwin is described later. At the same time you'll get PilRC. This "resource compiler" is used to 'make binary resource files from a resource script file'. Don't worry now about what this means!

3. A Palm OS SDK (Software development kit) You should go for version 4.0 or greater.

4. The GNU C++ compiler, GCC.

To test your programs without crashing your Pilot often, you will also need emulator software that emulates a complete Palm Pilot. This is POSE (from palmos.com). POSE requires 'skins' and a ROM image. Skins are pictures of devices, and a ROM image is the 'operating system' (not as freely available as the other components, but you can download an image from your Pilot).

As you will be working in UNIX, it's important to note sentinel differences between UNIX and DOS (and its derivative, Windows). UNIX uses slashes in path names, DOS uses backslashes. UNIX is case-sensitive, DOS isn't.[50] A good working knowledge of simple UNIX commands is desirable but not essential.

[50] Although CYGWIN is often fairly tolerant of DOS-like case insensitivity.

8.1.1 A first program

There are many little catches in the installation of the software. Here's a brief summary of what you need to do, but if all else fails, read the manual! At the time of writing, there are fairly good instructions at
http://prc-tools.sourceforge.net/install/cygwin.html. Also consider checking out: http://prc-tools.sourceforge.net/doc/prc-tools_1.html. ...but you may well find that our instructions suffice!

1. locate, download and *remember where you put*[51] the applications listed above: the setup.exe file for Cygwin, the SDK v 4.0 (or greater), and POSE (with some skins and ROM images).[52] You will also need a program that can 'unzip' — eg. WinZip, or the better JustZIPit.

2. Install Cygwin. The best way to do this is to create a download subdirectory (e.g. D:\download), and then run the Cygwin setup.exe program to download the 'standard' package. The setup package will ask you where you want to store the packages: choose a good name such as D:\download\fred. NB. You must get the GNU C++ compiler GCC with Cygwin — in the *Select Packages* menu click on the 'View' button to view and select `gcc`. Also get the important `make` package.[53]

   Once you've downloaded Cygwin, install it using the same setup.exe program.[54] Choose *Install from local Directory*, select an appropriate installation directory (e.g. D:\cygwin), and in the *Select Packages* menu use 'View' (and some clicking) to ensure that the relevant packages are selected, including `gcc`, and `make`.

3. Run the Cygwin setup program *again* but this time choose 'Download from Internet', and eventually, in the 'Choose Download Site(s)' menu, add *and then* go to the USER URL:

   `http://prc-tools.sourceforge.net/install`

   From this you should now download `prc-tools`, and `pilrc`. Select the packages as you did for GCC.

4. In DOS make a special palm directory, eg on the `D:` drive say:

[51] e.g. D:\Downloads !

[52] Click on the links!

[53] Once you've clicked on 'View', the packages will appear in the right hand column. In the 'New' column second from the left will be their status, at present 'Skip'. Click on this 'Skip' to change it to 'Install'. Packages that depend on the selected package will automatically be selected too. Don't mess with these!

[54] Remember the *fred* directory where you kept the packages?

```
md \PalmDev
```

5. Run Cygwin, and make a UNIX directory:

```
mkdir /PalmDev
```

6. From within Cygwin, symbolically link the UNIX and DOS directories! Thus:

```
mount -f "D:\PalmDev" /PalmDev
```

   You only have to do this once.[55]

7. Install the PalmOS SDK. Unzip the whole file (recreating the sub-directories) into, in our example, D:\PalmDev.

   Note that there is an associated readme file. Read it! It is possible that you may have to play around, renaming various directories. The important thing is that in your PalmDev directory (now accessible from both UNIX and DOS!) you will have a directory called `sdk-4`. This should in turn have two important subdirectories, called `lib` and `include`.[56]

8. Run Cygwin. Then change to the /PalmDev directory,[57] and then type in the following:

```
ln -s sdk-4 sdk
```

   It looks tricky, but all it does is to create a 'soft' link between a new directory (`sdk`, just made!) and the sdk-4 directory. When you use GCC, it will see this 'sdk' directory as the one it needs to use.[58]

9. Finally (whew!), still in Cygwin, type in:

[55] The corresponding command to unmount (remove) this symbolic link is *umount*. Note that if you are using a drive other than D: you'll have to alter the mount command appropriately.

[56] If you can't find the sdk-4 directory, then look at the directories you do have — you may have unzipped it into one of these, and have to move it up a level!

[57] cd /PalmDev

[58] To remove this link, simply say *rm sdk*.

```
palmdev-prep
```

What this should do is prepare links etc so that GCC (and friends) will see the files they need to see. Otherwise you might have to try and create a whole lot of such symbolic links and whatnot — a real pain with previous editions.

You should now be ready to run a simple program:

A hello world

Here's a very cursory 'Hello World!' application to start off ...

```
#include <PalmCompatibility.h>
#include <PalmOS.h>

DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{   EventType event;
    if (cmd == sysAppLaunchCmdNormalLaunch)
        { WinDrawChars("Hello world!", 12, 20, 50);
          do { EvtGetEvent(&event, evtWaitForever);
               SysHandleEvent(&event);
             } while (event.eType != appStopEvent);
        }
  return;
}
```

Later on, we will explore this program (with a detailed commentary), but for now, you may wish to simply cut and paste it using a simple text editor. Under CodeWarrior, you won't have to do any of the following, although we still recommend you read on as we believe you will derive valuable insights by doing so.

In the PalmDev directory create a subdirectory called e.g. `helloworld`, and save our hello world application as `helloworld.c`. You can do this from Windows, DOS, or Cygwin.[59]

[59] Note that if you're using an older version of the Palm SDK, the first two include lines should be replaced by: #include <Pilot.h>

Within Cygwin, perform the following:[60]

```
cd /PalmDev/helloworld
m68k-palmos-gcc -O2 helloworld.c -o helloworld
ls -l
```

All the above does is change to the relevant directory, invoke GCC on the helloworld.c file in order to compile and link it, and then `ls` shows us the directory. Note that we said -O2 and not -02 (Oh not zero) in the second line. We now have a new *object* file in our subdirectory simply called `helloworld`.

Next, we must split up our object file into applications thus:

```
m68k-palmos-obj-res helloworld
```

...and finally build a .prc resource file using PilRC. (Before you do this you may wish to `ls` again to see all the little .grc files that result from the above step)!

```
build-prc helloworld.prc "Hello world" HeWo *.helloworld.grc
```

The above, final step requires a little explanation. `*.helloworld.grc` tells us which files will be used to create the .prc file that is loaded onto the Pilot (We'll do this in a moment). The characters `HeWo` make up the unique identifier that the application has when it gets onto the Palm — if this identifier is the same as an existing one, that existing application will be replaced![61] The name of the application is clearly just "Hello world". And that's it! Well, not quite, because we still need to load the .prc file onto a Palm Pilot.

Installation

You can simply upload the hello.prc file to your Palm and see what happens, but there is a better way. Rather than crashing your Palm, first use POSE to test the application under Windows. Do so as follows:

1. In a convenient directory, create a subdirectory called `pose`.
2. Unzip the POSE emulator into this directory.

[60] You can cut each command from the document you're reading, and then paste by right clicking on the top bar of the Cygwin box, and then selecting *Edit* and *Paste*. We will soon explore a far more ergonomic way of submitting these commands!

[61] Actually, this is only partially true, as multiple Palm databases *may* have the same creator ID, as long as their *database types* differ! The database type must contain at least one upper case letter, unless it's a pre-defined Palm type. Ideally you should also register a creator ID with Palm.

3. Also obtain skins and ROM images, and keep them in appropriately named directories.

4. Run the emulator.exe file, and with a bit of fiddling you will be able to load your hello.prc file onto a fully functional emulation of the Palm of your choice! (Right click on the picture of the device to get the 'Install application' option).

How it works

Let's look at our C++ helloworld program again:

```
#include <PalmCompatibility.h>
#include <PalmOS.h>

DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{   EventType event;
    if (cmd == sysAppLaunchCmdNormalLaunch)
        { WinDrawChars("Hello world!", 12, 20, 50);
          do { EvtGetEvent(&event, evtWaitForever);
               SysHandleEvent(&event);
             } while (event.eType != appStopEvent);
        }
  return;
}
```

We include necessary header files, and then get down to the nitty gritty of this simple application — the PilotMain function that is analogous to a C++ `main` function. All this function does is intercept 'launch codes' and respond to these. Launch codes can come from several sources, but see how a 'normal launch' results in the drawing of "Hello world!" at the stated position on the screen.

That's the skeleton, but clearly there are frills. `PilotMain` receives several parameters, we repeatedly intercept *event*s that are later used in various system calls that handle and process these events, and so on. The `do` loop loops forever, or at least, until a 'stop' event is received.

As is usual in C++, we can insert comments almost anywhere, using the // type of comment (which extends to the end of the line), and the more traditional C /* this is a comment */ type of comment, which can extend across several lines.

The above application is the simplest practical one we've encountered. Next, we will look at a more complex (and more functional application). There is a massive step up to the next example, but once you've mastered it, much of the rest should be plain sailing. Be warned, the following is not for the faint-hearted!

But before we move on...

There is one important thing to note before we examine the next application. At about the time of creation of the SDK version 3.5, the Palm chaps, in their wisdom, decided to change specification of many important, basic data types. So `DWord` becomes `UInit32`, for example. A major pain in the whatsit! Now you can simply hack things by including PalmCompatibility.h as we did above, but, in order not to get our bottoms smacked later down the line, we will in future avoid the temptation to do so, and stick to the new standard.

8.2 A larger application

In this section we will demonstrate a larger, single screen application.[62] We will call our application `hello.c`.

8.2.1 The Makefile

Quite a lot of explanation is in order; CodeWarrior slaves may again wish to skip past the following section. First of all, because we will be using many little files, several of which depend on others, we will use the UNIX *make* facility to ease the typing load on our fingers (and keep things up to date). To work its magic, *make* imaginatively looks for a file called `Makefile` in the current directory. Here's the makefile, followed by an executive summary of how it works:

```
hello.prc: code0001.hello.grc tFRM0578.bin
    build-prc hello.prc "Hello world" HeWo *.hello.grc *.bin
    ls -l *.prc

tFRM0578.bin: hello.rcp hello.h
    pilrc hello.rcp

code0001.hello.grc: hello.c hello.h
    m68k-palmos-gcc -O2 hello.c -o hello
    m68k-palmos-obj-res hello
```

Many of the lines above are remarkably similar to the lines we tediously typed into Cygwin. To understand the above more completely, you need to know that:

1. a line containing a colon is a *dependency line* — to the left of the colon is the file that depends on the files to the right of the colon.

[62] In the following section, we'll try and examine most of the user interface features that we will ultimately use. In addition, a database will be opened, queried and closed, [and ultimately, infra-red beaming of data will be demonstrated, with transmission and reception].

2. Following the dependency line is/are one or more *shell lines*, which must begin with a tab character (NO, not spaces; many text editors mess this up, and the script won't run).[63] The shell lines are submitted as if they are command lines.

3. (Note that comments can be inserted into the makefile, by preceding them with a # character, and a line can be continued to the next line using a backslash at the very end of the line).

Whenever you run `make` from within the *hello* directory under Cygwin, it will update files whose dependencies are more recent than they are![64]

The fine details are that there are some new files we need to become acquainted with. The .rcp file is a text script that PilRC uses to make certain resources. tFRM0578.bin is generated by pilrc,[65] and code0001.hello.grc is generated by the associated shell lines.

[63] If you're smart and are using WinEdt, Click on Options — Preferences — Tabs — *Preserve and Display tabs* to fix this!

[64] This is a recursive process.

[65] The name may be slightly different, which illustrates the merit of being able to type in at the command line. There is no reason why you cannot type in each shell line on its own, see what happens using *ls -l*, and then alter the script!

8.2.2 The code

Our C++ code is a lot bigger, so we will break it up into readable sections, first the headers and PilotMain function, together with startup and exit code:

```
#include <PalmOS.h>
#include "hello.h"  // C++ prototypes

UInt32 PilotMain(UInt16 cmd, void *cmdPBP, UInt16 launchFlags)
{ if (cmd == sysAppLaunchCmdNormalLaunch)
       {  if (StartApplication() == 0)
             { EventLoop (cmd, cmdPBP, launchFlags);
               StopApplication();
             };
}      }; // only handle normal launch, for now.

static int StartApplication(void)
{   UInt32       romversion;
    FtrGet(sysFtrCreator, sysFtrNumROMVersion,
           &romversion);
    if (romversion < sysMakeROMVersion(3,5,0,
           sysROMStageRelease,0))
           { WinDrawChars("Bad ROM version!",
                          16, 20, 50);
             return 1; // fail
           };
   FrmGotoForm(MainForm); // first form
    return 0; // ok
}

static void StopApplication(void)
{   FrmCloseAllForms();
}
```

The StartApplication and StopApplication functions will allow us to insert startup code. For now, all we do is use the mysterious `FtrGet` function to get the ROM version, and compare it to 3.5.[66] If less, we fail, otherwise we open our main form. How do we identify this main form? We'll put you off with a gentle lie — that `MainForm` is simply the magic number of a resource defined in the included file `hello.h`.[67]

[66] We use the hideous macro sysMakeROMVersion to make the code for version 3.5, although we could just have specified 0x03503000, the relevant code. Either is ugly.

[67] In fact, the magic number is defined in the file *hellorcp.h* which is included within *hello.h*, so our lie was a very gentle one indeed!

Next, see how we've moved our do loop code out into a separate function, EventLoop:

```
static UInt32 EventLoop(UInt16 cmd, void *cmdPBP,
                        UInt16 launchFlags)
{   EventType   e;
    UInt16      err;
    do { EvtGetEvent(&e, evtWaitForever);
         if (!SysHandleEvent(&e))
            { if (!MenuHandleEvent(NULL, &e, &err))
                 { if (!myApplicationHandleEvent(&e))
                      { FrmDispatchEvent(&e);
            };   };   };
       } while (e.eType != appStopEvent); // until stop
  return 0;     // ?
}
```

Although the above is similar to our original helloworld.c, it uses a more complex hierarchy of event handling, as recommended by Palm. You get the gist — sequentially submit the event to various functions, which all do their bit, or not! Although it looks clumsy, it allows us a lot of control over events. SysHandleEvent, MenuHandleEvent and FrmDispatchEvent are all pre-defined system functions; the only function we define is myApplicationHandleEvent, which we look at next.

Here's the offending code:

```
Boolean myApplicationHandleEvent(EventPtr e)
{ FormPtr frm;
  Int16     formId;
  Boolean handled = false;

  if (e->eType == frmLoadEvent)
     {  formId = e->data.frmLoad.formID;
        frm = FrmInitForm(formId);
        FrmSetActiveForm(frm);
        switch (formId)
             {  case MainForm:
                  FrmSetEventHandler(frm,
                    MainFormHandleEvent);
                  break;
              } // "default:" would be nice!
        handled = true;
      }
  return handled;
}
```

All the above function looks for is a `frmLoadEvent`, which, if found, prompts us to set the event handler for that form (i.e. transfer control over to the form). FrmInitForm loads a form[68] — all it needs is the resource ID.

The event handler is the eponymous MainFormHandleEvent:

```
static Boolean MainFormHandleEvent(EventPtr event)
{ Boolean     handled = false;
  FormPtr     frm;
  char *      pStr;

  switch (event->eType)
   { case ctlSelectEvent:  // control button tapped
        if (event->data.ctlEnter.controlID
              == MainHelloAgainButton)
         { pStr = "Hello, Again!";
           WinDrawChars(pStr, StrLen(pStr),
                        ((160-FntCharsWidth
                           (pStr, StrLen(pStr)))/2),
                        80);
           handled = true;
         }
        break;

     case frmOpenEvent:  // initialise form
        frm = FrmGetActiveForm(); //pointer to form
        FrmDrawForm(frm);         // draw it
        handled = true;
        break;

     case menuEvent:  //menu item (our 'About')!
        MenuEraseStatus(0);  // clear display
        frm = FrmInitForm(AboutForm); // load About
        FrmDoDialog(frm);   // display, await "ok"
        FrmDeleteForm(frm); // delete About form
        handled = true;
        break;
   }
  return(handled);
}
```

The above should be fairly self-explanatory: the details may be a bit blurry, but we return true if we've successfully handled one of 3 events: initialisation, tapping the button, or clicking on 'About' in the menu. See how FrmDoDialog waits until any button is tapped (returning the resource ID of the button that was tapped).

[68] FormType *FrmInitForm (UInt16 rscID)

And that's the whole program, apart from the tiny `hello.h` header:

```
#include "hellorcp.h"

static int StartApplication(void);
static void StopApplication(void);
static UInt32 EventLoop(UInt16 cmd, void *cmdPBP,
                        UInt16 launchFlags);
Boolean myApplicationHandleEvent(EventPtr e);
static Boolean MainFormHandleEvent(EventPtr event);
```

The header file does the usual C++ thing — containing prototypes of the various functions, describing their arguments and what is returned. Next, let's look at the resources — forms and menus, and so on.

8.2.3 The resource compiler

PilRC is our resource compiler. Initially, you may wish to try pasting the various code samples into files and actually creating the application before you plod through the details, [have hrefs to various quoted code sections] but here we look at those selfsame details. Here is the hello.rcp file:

```
#include "hellorcp.h"
FORM ID MainForm AT ( 0 0 160 160 )
NOFRAME
USABLE
MENUID MainFormMenuBar
BEGIN
    TITLE "Hello"
        LABEL "Hello World!" ID MainHelloWorldLabel
              AT (49 40) FONT 2
    BUTTON "hello again" ID MainHelloAgainButton
           AT ( 46 129 68 14) USABLE FRAME FONT 0
END
VERSION "1.0"
ICON "hello.bmp"

MENU ID MainFormMenuBar
BEGIN
    PULLDOWN "Options"
    BEGIN
        MENUITEM "About"  MainOptionsAbout
    END
END

FORM ID AboutForm AT ( 2 18 156 140 )
FRAME
```

```
MODAL
USABLE
MENUID MainFormMenuBar
BEGIN
    TITLE "About"
        LABEL "Hello World" ID AboutAppNameLabel
              AT (51 22) FONT 1
        LABEL "Version 1.0" ID AboutVersionLabel
              AT (52 32) FONT 1
        BUTTON "OK" ID AboutOKButton
              AT ( 62 125 37 12) USABLE FRAME FONT 0
END
```

The above is somewhat intimidating, but all it does is specify the structure of a variety of different *resources* — FORMs and MENUs, as well as a motley collection of other things, VERSION and ICON.

Note that this script will not work with the Metrowerks CodeWarrior, only with PilRC — for the former, you will have to use their snazzy user interface to achieve a similar result. Our PilRC code also includes a header file called `hellorcp.h`, which is simply a long list of constants.[69] Here's that file:

```
//      Resource: tFRM 1100
#define AboutForm                   1100
#define AboutOKButton               1105
#define AboutAuthorLabel            1101
#define AboutAppNameLabel           1103
#define AboutVersionLabel           1104

//      Resource: tFRM 1400
#define MainForm                    1400
#define MainHelloAgainButton        1403
#define MainHelloWorldLabel         1401

//      Resource: MBAR 1000
#define MainFormMenuBar             1000

//      Resource: MENU 1010
#define MainOptionsMenu             1010
#define MainOptionsAbout            1010
```

Let's explore the details, paying particular attention to FORMS, and objects that can be contained within them. This all comes from the PilRC manual, which can be found at ardiri.com.

[69] It doesn't have to be called this, any old name with a .h suffix will do, as long as you're consistent. The constant values are also rather arbitrary, but shouldn't exceed 9999.

1. PilRC has a host of possible command-line options,[70] but normally you just say:

```
pilrc filename.prc
```

2. PilRC creates a whole lot of little .bin files that are used in our final makefile step, to make a .prc file.[71]

3. PilRC is familiar with several different types of field within a resource: identifiers (arbitrary, e.g. myJane), characters and strings (which may be appended to those on the next line by having the last character of the line as a backslash), numbers (including simple arithmetic!), and position coordinates, which may include the magic words: AUTO, CENTER, PREVLEFT.[72]

4. /* C-style */ and // C++ comments are permitted

5. You already know that `#include` directives are allowed (These can even be for java package files)!

6. The object types that can be defined are: FORM, MENU, ALERT, VERSION, STRING, STRINGTABLE, CATEGORIES, APPLICATIONICONNAME, APPLICATION, LAUNCHERCATEGORY, ICON, & BITMAP.[73] Whew! We won't concern ourselves with all of these, and will concentrate on FORMs.

7. As illustrated in our example, the format of a FORM is:

```
FORM ID <FormResourceId.n>
  AT (<Left.p> <Top.p> <Width.p> <Height.p>)
     [FRAME] [NOFRAME]
     [MODAL]
     [SAVEBEHIND] [NOSAVEBEHIND]
```

[70] Read the manual for the options, noting that there's even a little-endian option for ARM microprocessors!

[71] The naming of .bin files that are created is far from random: the four character resource type has appended to it the hexcode resource ID, and then the *.bin* suffix.

[72] and the similar PREVRIGHT, PREVTOP, PREVBOTTOM, PREVWIDTH, PREVHEIGHT, all referring to the position of the previous item.

[73] And also ICONFAMILY, SMALLICON, SMALLICONFAMILY, BITMAPGREY, BITMAPGREY16, BITMAPCOLOR, BITMAPCOLOR16, BITMAPCOLOR16K, BITMAPCOLOR24K, BITMAPCOLOR32K, BITMAPFAMILY, BITMAPFAMILYSPECIAL, BOOTSCREENFAMILY, TRAP, FONT, FONTINDEX, HEX, DATA, INTEGER, BYTELIST, WORDLIST, LONGWORDLIST, PALETTETABLE, FEATURE, GRAFFITIINPUTAREA, COUNTRYLOCALISATION, LOCALES, KEYBOARD, MIDI, HARDSOFTBUTTONDEFAULT, SYSAPPLICATIONREFERENCES, and TRANSLATION.

```
      [USABLE]
      [HELPID <HelpId.n>]
      [DEFAULTBTNID <BtnId.n>]
      [MENUID <MenuId.n>]
      [LOCALE <LocaleName.s>]
BEGIN
  <OBJECTS>
END
```

In the above, the <objects> may be a TITLE, BUTTON, PUSHBUTTON, CHECKBOX, POPUPTRIGGER, LABEL, FIELD, POPUPLIST, LIST, FORMBITMAP, GADGET, TABLE, SCROLLBAR, & SLIDER. [later whittle down the list and footnote the nasties]. Each object has its own format. TITLE is a rather special object in that it is followed by a "Text string in quotes", but many of the others have fairly similar structure:

```
<Label.s> ID <Id.n> AT ( /* coordinates */ )
    //more stuff follows here!
```

— a name, an ID number, AT followed by position coordinates in parenthesis, and then several other components. Objects that fit this general structure include buttons, triggers, checkboxes and labels. If you don't actually refer to the control from within your program (via, for example, our header file `hellorcp.h`), then you can replace the identifier with AUTOID, and PilRC will automatically generate an ID number.

Most of the other objects have no label, but do have ID and AT coordinates. They often have a host of other attributes too. For a discussion of alert, see below.

8. Note the utility of having STRING resources, and even a LIST of STRINGS.

9. One can fine-tune how the pilot views our application with LAUNCHERCATEGORY and CATEGORIES.

10. ICON is used to specify the particular bitmap associated with the program. It *must* be 32x32, 32x22, or 22x22 pixels in size. Related are SMALLICON, and ICONFAMILY (for colour, etc). BITMAP and related instructions create bitmap resources.

8.3 Debugging

The command-line debugger available with GCC is immensely powerful in debugging faulty programs! You use it together with POSE, thus:[74]

1. Use the -g switch in GCC, along the lines of:

```
m68k-palmos-gcc -g -O2 -fno-exceptions -fno-rtti
                pain5.cpp -o pain5
```

   (We've broken the line into two for convenient reading).

2. Load the COFF file the above generates (as *PAIN5*) into GDB using:

```
m68k-palmos-gdb pain5
```

3. Run POSE and load *but do not run* the program to be debugged (Here pain5.prc);

4. Within CYGWIN type in `target pilot localhost:6414` (just so)!

5. Run the POSE program, and type in `cont` in the GDB window to continue until something horrible happens (best done using a debug ROM with POSE). At this point, control will go back to GDB, and you can do smart things like ...

6. Type in the backtrace command: `bt` ...to allow you to see the call stack, that is, where the offending command crashed the Palm. Wonderful, and now you can nip off and read the GDB manual for all the other smart things you can do (or simply type in `help` at the GDB command line). Type `q` to quit.

8.4 Communications

The candid observer (at least, if that observer is me) will freely admit that on the surface, PalmOS communication between PDA and PC sucks! Each substantial program on the PDA needs its own *conduit* to plug into the Palm desktop applications, and then this conduit is used as necessary to synchronise data backwards and forwards between PC and PDA. This sounds straightforward until you realise that you now need a whole lot of Microsoft tools (C++, .NET or whatever) in order

[74] There's a good introduction on the web by Warren Young

to create the bloody conduit, if you're working under Windows as many people are. In addition, you need separate development under Linux.

Let's explore a different approach. You'd think that as the PDA connects to the serial port (or USB), it would be possible to talk directly between the two without using HotSync. You'd be right — along the following lines:[75]

1. You might (but don't yet) find the program zboxz on the Internet (Look for zboxz037.zip), unzip it, and HotSync the binary PalmOS file *zboxz.prc* onto your handheld. In the Windows binary subdirectory which appears after unzipping zboxz, you'd run the program *Winbox.exe* to turn an arbitrary text file you've created (say, *silly.txt*) into silly.txt.pdb, and also move this file to the handheld.

2. *Exit* HotSync on the PC.

3. Locate the program Teraterm (look for TeratermPro31.zip) on the Internet, and run *ttermpro.exe* on your PC.[76] Under File: New connection: click on the `Serial` radio button, choose the COM port your PDA cradle is attached to. Under Setup: choose the serial port and set the bit ('baud') rate to 19200, at least initially.[77]

4. On the PDA run ZBoxZ and select the 'box' file silly.txt which should be listed. Click on `RawSnd` and `Continue` (you can send an initial string if you wish), and the text in silly.txt should appear on the Teraterm terminal.

The above process demonstrates how we can communicate between palm and PC without having to resort to HotSync. There is potential for greater things:

- More recent version of TeraTerm are said to be able to act as an intermediate between a variety of communication channels (networking and infrared, for example) and a Perl program using networking packages [LOOK AT Net::NetLib and Net::Telnet].

- As the zboxz source is available under the GPL, we can examine this code and make our own similar code.

- PalmConnect USB software (eg. usb_win_2k.zip) is available to allow the USB connection to be seen as COM4. With USB we should theoretically

[75] with a PDA which attaches to a COM port!

[76] This is said to be a better option than the Windows-intrinsic *hypertrm.exe*; the Linux equivalent of hyperterm is said to be Minicom.

[77] Leave the data/parity/stop at 8/N/1.

then be able to proceed as above! Unfortunately this 'remapping' only seems to work with a USB/serial adapter, where the PDA doesn't actually have a USB cable. If your PDA connects via USB, then you seem to be compelled to use HotSync.

But the above is clumsy and painful! Fortunately for us, there is software available which does the task. Some of it is free: check out pilot-link (pilot-xfer, which is open-source and works on Linux) and Pilot-install (for Windows, free to private users). In addition, Plucker has a Windows facility for transferring files with minimal fuss. See the file *PerlPgm.tex* where we discuss our use of AutoIt (a good search term) and Pilot Install.

8.5 Profiling

POSE (in concert with GDB) has a whole array of profiling features. Use the Windows POSE executable with 'profile' in its name. Run this application (e.g. *Emulator_Profile.exe*), right click, and select Profile:Start. When you save profiling results, they are written to a text file (and a Metrowerks mwp file) in the same directory as the profiler executable.

C++ mangles names so the results can be a mess. Unfortunately owing to the current size of our main program, specifying the GCC flag *-mdebug-labels* results in compiler errors due to the increased size of the program with embedded labels. We can still use this flag with most libraries to good effect.

The tab-delimited .TXT file can be imported into Excel with minimal fuss. You can then easily browse through (and sort) the columns of data provided for all functions. Look particularly at the 'only msec' and 'plus kids msec' for how much time is expended in each function. The 'count' column is the number of times each function was invoked during your profiling session. Remember that you should regard all times as relative, as emulator speeds are often slower than on a good PDA, unless you have a really fast desktop!

Sorting by 'plus kids msec' gives a good idea of how much time is spent in each routine — inclusive of all called routines. You can create a tree of who is calling whom using the 'parent' and 'index' columns.

Using such profiling on our early SQL code demonstrated that routines such as that translating to intermediate format take little time — the bulk of time is expended within the SeekMany routine (See *Sql3Lib.tex*). In this routine, subroutines such as CreateTableList, MakeQueryList and FindResultColumns consume little time. DmQueryRecord and WriteAnswer are also generally not very expensive.

As expected DmFindSortPosition is expensive, about the same as TestLogic,

together consuming over 30% of total time states. TestLogic expends most of its time in the PerformJoin routine.

Oddly enough, CountRecords takes a *lot* of time, expended in DmDatabaseSize — room for optimisation![78] DmCloseDatabase (called by PalmFileClose) takes some time as well.

Relative proportions are:

Routine	*Time*
SeekMany	77
TestLogic	18
Interrupt$2F	23
PerformJoin	15
s_DmFindSortPosition	14
CountRecords	11
KillDbList	9
MemHandleSize	10
PalmFileClose	8
CreateDbLink	9
ClearJoin	6

Table 3: Relative time profile (SQL3LIB)

The numbers of course don't add up to 100 as they are relative, and some routines are called by others.

[78] Presumably PalmOS runs through and enumerates records. We might keep our own count in the header, saving about 10% of our time, at the expense of rewrites on new record creation!!

8.6 Conclusion

The above is a very simple introduction to PalmOS programming, but I hope it contains enough to get the novice Palm programmer started at minimal cost.

8.6.1 A small note on binary files

With our dogwagger program, we can pull the entire source for all of our code out of a LaTeX source document. Until recently, it wasn't possible to extract binary files, but this is sometimes necessary, so we've upgraded things to permit this functionality!

The following demonstrates decoding of a uuencoded file (*pain.bmp*)! It allows *dogwagger20.pl* to extract the binary file from the LaTeX source of this documentation.

```
begin 644 pain.bmp
M0DV^''''''''''#X''''H''''('''''"''''''!''$''''''('''''#>#@''W@X'
M''('''''"''''''''''/___P#_____________________________________
M_____________________________________________________________
M____@?''(^?F9F?GYF9GY_!F9^?^9F?@9F8GYS#$3^<____G/___YS_G_X!_
*Y_____________P''
'
end
```