Analgesia Database:
SQL3 Library for PalmOS PDA

Version 0.95

J.M. van Schalkwyk

February 27, 2009

Contents

1 The C file: SQL3.c

We here describe SQL routines for the Palm SQL database. We have moved these routines out from our main program into a library as this makes sense both in terms of space and in terms of modularity. The routines here take pretty standard SQL statements and interpret them, but still need a fair amount of refinement.

You will also need to know the structure of our data records, stored in Palm 'PDB files' made up of records. We allocate one such 'file' (database) per SQL database table — the PalmOS usage of the word 'database' is confusing.[1]

Let's look at our SQL library. First, the included headers:

```
#include <SystemMgr.h>
#include <PalmOS.h>

#define ZERO 0
#define MAXCOLUMNS 64
#define MAXACTUALROWS 2000
  // maximum number of rows in any database, at present [??]
  // (ALSO USED IN IdxLib.tex, so best have common value
  //  in higher .h file)

#include "SQL3.h"
#include "../palmsql3A.h"
#include "../err/ERRDEBUG.h"
#include "../console/CONSOLE.h"
#include "../idx/IDX.h"
#include "../cache/cache.h"

#define OPSTORE 'S'
#define OPAND   'A'
#define OPOR    'O'
#define OPNOT   'N'
#define OPNO    'F'
#define OPYES   'T'

// none of the following may be uppercase (or 0x0):
#define  ISEQUAL               '='
#define  ISGREATER             '>'
#define  ISLESS                '<'
#define  LESSEQUAL             'l'
#define  GREATEREQUAL          'g'
#define  ISNULL                '0'
#define  ISNOTNULL             '1'
```

[1] Technically, the PDB format only applies to the translated version stored on e.g. a PC, and the internal format on the PDA is reserved by Palm. This fact doesn't prevent us from structuring our records appropriately.

```
#define  NOTEQUAL              '~'
```

The CONSOLE `include` statement allows us to access routines within the CONSOLE library, a powerful addition, as we can now write debugging statements to the console for later delectation! We've recently added similar access to the error library.

1.1 Debugging

It's important that we have an adequate debugging strategy. The 'production' version of this library will not include code in between +OPTIONAL . . . -OPTIONAL markers. We use three different flags to print debugging statements to the PDA console. The most frequently used flag is fDEBUG_SQL. We reserve the flag fDEBUG_SQK for debugging of ORDER BY statements, and fDEBUG_SQJ is used for 'deep' debugging of SQL queries. For the numeric values of these flags, see the file *CProgMain.tex*.

2 Structures

We use several structures, the dbLINK which is used to implement a linked list of open PalmOS databases, and then the cQUERY structure, which links 'query nodes', flexible component parts of our SQL queries which allow us to interpret a WHERE statement.

```
struct dbLINK {
                DmOpenRef odb;  // open database
                Char * name;   // name of database
                Int16 nlen;    // length of name
                Char * head;   // header (whole of row 0 header)
                MemHandle hand;// HANDLE of header (clumsy)
                               // (MemPtr cast as Char * ???)
                Char * current;// CURRENT evaluation row !
               struct dbLINK * next; // next item (or null)
              };

struct cQUERY {
                void * dtable;  // dbLINK * dtable; // see ABOVE
                Int16 col;      // index of column referred to in tbl
                Int16 len;      // (maximum) length of item
                Char type;      // nature of operand (thus comparison)
                Int16 scale;    // scale, where relevant
                Char test;      // condition tested: compares 2 oprnds
                Char * opstring;// string of post-evaluation operatns
```

```
  Int16 olen;     // length of opstring ?!
  Char * cnstant; // constant value to cf with (2nd op)
  Int16 cnstlen;  // length of this cnstant value
struct cQUERY * next;// next item to evaluate
struct cQUERY * up;// use if 2nd operand = column! ugly.
};
```

3 A note on indexing

Profiling this library, it seems that a considerable amount of time is spent within the binary searching callback function (DmComparF *) we've called *janet*. This callback corresponds to CompareRowKeys in our main C++ program. The following discussion is rather speculative, and for the time I've *disabled* the indexing as (despite my comments below) our indexing didn't speed things up that much. The main reason for this lack of benefit seems to be the extraordinary time taken by PalmOS in locating a PalmOS database. Clearly there's some clunky code beneath the surface. It's probably best to skip over this section, especially on first reading.

On the suspicion that this functionality could be optimised substantially, I've created an indexing library (See *IdxLib.tex*). In order for functions from this library to work within the Sql3 library, we must ensure that not only are the functions accessible (passed using PassConsole, and stored in the 'global' IDXLIB), but also that an appropriate index exists for each table queried. Each such index will be stored in a PalmOS database with the same name as the relevant table, but prefixed by an 'i-', as specified in the IdxLib library.

There is a time hit for initial creating of indexes, and our vital index updater is clumsily written (NEWINDEXITEM).

I have not yet created/enforced indexing for all searches, but such an approach might speed SQL queries enormously. The current indexing is very primitive, being confined only to key-based searches for records.

Examining our original ('janet') code, top-level functions and chains of dependencies are:

1. SQL3SELECT : SeekWrap : SeekMany : TestLogic : PerformJoin : s_DmFindSortPosition

2. SQL3UPDATE : TestLogic (etc)

3. SQL3INSERT : SQLins : DataRowInsert : s_DmFindSortPosition

In addition, SeekMany makes direct reference to s_DmFindSortPosition, which in our usage always takes null values for the 3rd and 5th parameters.

You can see that references to DmFindSortPosition might be replaced by the index library call FINDROW(refnum, DatabaseName, DbNameLen, 1, &index, 4, 0), where refnum is the IDX module reference, and &index is a pointer to the double word index we are seeking.

The catch is, can we access the database from the library while we've opened it elsewhere? [CHECK THIS, FIX ME UP].

3.1 A debugging strategy

In debugging the indexing functionality we will do the following:

1. Identify early on which functions will ultimately invoke *janet*, and apply ISINDEX and MAKEINDEX appropriately, checking for errors. See CreateDbLink for details — it's appropriate to do the indexing before we open up the database tables — and FixIndex in section 10.1.

2. Then, when creating a record, similarly apply ISINDEX and MAKEINDEX. SQL3INSERT similarly uses FixIndex.

3. Next, invoke NEWINDEXITEM when creating a new record. For now, only insert a new key value. This invocation is performed within SQL3INSERT. We must check that the order is correct in the resulting index file (look in detail).

4. After each invocation of *janet*, also invoke FINDROW, checking that the two results are the same. We first do this within PerformJoin, and later only with DataRowInsert.

5. 'Finally' we can get rid of *janet*, replacing her with FINDROW. We then need to profile and look at overall performance.

6. After initial experimentation with indexes, the next logical step is to index all columns where speed is relevant, but probably not *all* columns used for searches! Functions like 'SeekMany' will probably derive particular benefit from such changes.

4 Basic functions

Simple, basic functions needed in this library.

```
Err start (UInt16 refnum, SysLibTblEntryPtr entryP)
{
  extern void *jmptable ();
  entryP->dispatchTblP = (void *) jmptable;
  entryP->globalsP = NULL;
  return 0;
  }
```

4.1 Open a library

In the following routine, we open the library — but with a difference — we make use of global variables within the library (a la Gausslib)! The principal variables are CONSOLE and DEBUGFLAGS, but, for what it's worth, we also keep a count of the number of times the library has been opened.[2]

CONSOLE allows us to write arbitrary messages to our console program, and DEBUGFLAGS controls which such debug statements are written, as in the main C++ program.

```
typedef struct {
  UInt16 CONSOLE;
  UInt16 ERRLIB;
  UInt16 DEBUGFLAGS;
  UInt16 IDXLIB;
  UInt16 CACHELIB;
  Int16 CANINDEX;
  Int16 CANCACHE;
  Int16 UPTURN;
  UInt16 refcount;
  } Sql3Lib_globals;
  // include count in expectation of multiple openings!
```

The newly added IDXLIB references the indexing library, and CACHELIB the sql caching library. CANINDEX is cleared to zero (and disables use of indexing) unless IDXLIB is nonzero *and* no indexing error has occurred. UPTURN allows us to force reversal of the sort order (ASC becomes DESC, vice versa) — it is only used when performing internal SQL statements and then retrieving data rows from the stack.

[2] We really need to look into this multiple opening option, although we don't use the facility at present. Particularly important with multiple invokers would be DEBUGFLAGS.

```
Err SQL3Open (UInt16 refnum)
 {
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  Sql3Lib_globals *gl = entryP->globalsP;
  if (!gl)
     {
       gl = entryP->globalsP = MemPtrNew (sizeof (Sql3Lib_globals));
       MemPtrSetOwner (gl, 0); // note use of sys fxs here, above
       gl->CONSOLE = 0;
       gl->ERRLIB = 0;
       gl->DEBUGFLAGS = 0;
       gl->IDXLIB = 0;
       gl->CANINDEX = 0;
       gl->CACHELIB = 0;
       gl->refcount = 0;
       gl->CANCACHE = 0; // overall governs caching (debug)
       gl->UPTURN =0; // standard
     }
  gl->refcount ++;
  return 0;
 }
```

We have to allocate memory for our globals using MemPtrNew; ideally we should wrap the system calls as usual (Here using `S_` rather than `W_`).

```
Err SQL3Close (UInt16 refnum, UInt16 *numappsP)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  Sql3Lib_globals *gl = entryP->globalsP;

  if (!gl) {
    /* We're not open! */
    return 1;
    }

  /* Clean up.  */
  *numappsP = --gl->refcount;
  if (*numappsP == 0) {
    MemChunkFree (entryP->globalsP);
    entryP->globalsP = NULL;
    }
  return 0;
}

Err nothing (UInt16 refnum) {
```

```
  return 0;
  }
```

In the above, the `nothing` routine is our standard, but SQL3Close undoes the mischief we wrought in SQL3Open, freeing the memory chunk allocated to the globals.

4.2 Set console

This routine allows us to set ‘global’ references to the console and error libraries.

```
Int16 PassConsole (UInt16 refnum, UInt16 cons, UInt16 errlib,
                   UInt16 idxlib, UInt16 cachelib)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  Sql3Lib_globals *gl = entryP->globalsP;
  if (!gl)
     {  return 1; // fail
     };
  if (gl->CONSOLE)
     { return 1; // fail if already set
     };
  gl->CONSOLE = cons; // set console
  gl->ERRLIB = errlib;

///  if (idxlib) // if indexing enabled /// [disabled at present]
///     {
///       gl->IDXLIB = idxlib;
///       gl->IDXLIB = 0;  // THIS IS DEBUGGING FOR PROFILER
///       gl->CANINDEX = 0; // 1=allow
///     };

  gl->CACHELIB = cachelib;
  gl->CANCACHE = 1; // SIMPLE FLAG:
                    //  0=OFF, 1=ON disable/enable caching
  // following is clumsy debugging hack:
  if (! gl->CANCACHE)
     { DISABLECACHE(cachelib); // disable caching
     };

  return 0;
}
```

4.3 Set debugging flags

PassConsole populates our CONSOLE global entry with the handle of the console library, allowing us to write debug messages to the console. Similarly, PassBug

does the same for DEBUGFLAGS.

```
Int16 PassBug (UInt16 refnum, UInt16 bugs)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  Sql3Lib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return 1; // fail
  };
  gl->DEBUGFLAGS = bugs;
  return 0;
}
```

4.4 Alter UPTURN flag

The UPTURN flag is used to reverse the ORDER BY sort order (when set to 1). This function should be used with caution, and only as an internal system function; UPTURN flag must be reset to zero once it has been finished with, as otherwise all ORDER BY invocations will be reversed (Danger)! The value returned is the former value of the UPTURN flag, not an indicator of success or failure.

```
Int16 SQL3UPTURN (UInt16 refnum, Int16 turn)
{
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  Int16 oldturn;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  oldturn = gl->UPTURN;
  gl->UPTURN = turn;

  return(oldturn);
}
```

Here’s the function to retrieve the value of the flag:

```
Int16 GetUpturn (UInt16 refnum)
{
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  return(gl->UPTURN);
}
```

4.5 New and Delete

```
MemHandle s_MemHandleNew (UInt32 size)
{ if (! size)
   { return 0;
   };
  return MemHandleNew (size);
}


MemPtr s_MemHandleLock (MemHandle h)
{ if (! h )
   { return 0;
   };
  return MemHandleLock (h);
}


Int16 s_MemPtrFree (MemPtr p)
{ Int16 ok;
  if (p == 0)
   { return 0;
   };
  ok = MemPtrFree (p);
  if (!ok) { return 1; };
  return 0;
}


Int16 s_MemPtrUnlock (MemPtr p)
{Int16 ok;
  if (p == 0)
   { return 0;
   };
  ok = MemPtrUnlock (p);
  if (!ok) { return 1; };
  return 0;
}
```

WriteErr writes an integer to the (low level) cyclical error buffer. It uses ErrorWrite, an error library function to do this cyclical error buffer write.

```
void WriteErr (UInt16 refnum, Int16 e)
{
  UInt16 ERRLIB;
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
```

```
  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  ERRLIB = gl->ERRLIB;
  if (! ERRLIB)
     { return; // safety 1st
     };
  ErrorWrite(ERRLIB, e);
}

Char * xNew2 (UInt16 refnum, Int16 memsize, Int16 e)
{ MemHandle memH=0; // clumsy.
  MemPtr    memP=0;

  memH = s_MemHandleNew(memsize+2); // 2 more bytes!
  if (! memH)
     { return 0;
     };
  memP = s_MemHandleLock(memH);
  if (! memP)
     { return 0;
     };
  if (e)
     { WriteErr(refnum, e);
     };
  *((Int16 *)memP) = e;
  return (((Char *) memP)+2);
}
```

Similar routines occur in ScriptingLib, and here's the Delete2 function lifted from the same library:

```
Int16 Delete2 (UInt16 refnum, MemPtr memP)
{
  Char * p;
  Int16 i;

  if (! memP)
     { return 1;
     };
  p = ((Char *)memP)-2;
  i = *((Int16 *)p);
  if (i)
     { i |= 0x8000;
       WriteErr(refnum, i);
     };
  memP = (MemPtr) (p);
  if (! s_MemPtrUnlock (memP))
```

```
     { return 0; // fail
     };
  if (! s_MemPtrFree (memP))
     { return 0;
     };
  return 1; // success
}
```

4.6 Write text to console

Now, the routine to actually write text to the console:

```
void ConTx(UInt16 refnum, Char * txt, Int16 txlen,
                      UInt16 bugflag)
{
  UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  UInt16 dbg;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  dbg = gl->DEBUGFLAGS;

  if (bugflag) // okay, might use &&
     { if (! (dbg & bugflag))
          { return;
          };
     };
  if (! CONSOLE)
     { return;  // fail
     };
  // now for some sanity checks:
  if ((txlen < 1)||(! txt))
     { ConWrite(CONSOLE, "??", 2);
       return;
     };

  // if leading ? ie error write, newline:
  if (*txt == '?')
     { ConWrite (CONSOLE, "\n", 1);
     };
  // finally, write:

   ConWrite(CONSOLE, txt, txlen);
}
```

In the above, if `bugflag` is zero, then we 'always' write to the console — setting this value to zero forces a write, regardless of the flags. This convention is to allow forced writing of short error messages to the console, and should only be used for this purpose. We define the constant ZERO to allow us to identify such statements.

```
Int16 s_StrLen (Char * txt)
{
  if (! txt) { return 0; };
  return StrLen(txt);
};
```

Here's a version which uses ASCIIZ strings:

```
void ConAsc(UInt16 refnum, Char * txt, UInt16 bugflag)
{
  Int16 txlen;
  txlen = s_StrLen(txt);
  ConTx(refnum, txt, txlen, bugflag);
}
```

Next, an error signaller:

```
void SayErr (UInt16 refnum, Int16 e)
{
  UInt16 ERRLIB;
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  ERRLIB = gl->ERRLIB;
  ErrorString (ERRLIB, 0, 0, e);
}
```

Supplying a null pointer with zero length to ErrorString forces a write of the error message to the console without returning any string. Don't confuse this with the lower-level WriteErr.

4.7 Write Integer to console

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

ConI is similar to *ConTx*.

```
void ConI (UInt16 refnum, Int32 i, UInt16 bugflag)
{ UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  UInt16 dbg;
  Int16 ilen;
  Char * CRUTCH2;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  dbg = gl->DEBUGFLAGS;

  if (bugflag) // okay, might use &&
     { if (! (dbg & bugflag))
          { return;
          };
     };

  CRUTCH2 = xNew2(refnum, maxStrIToALen+1, 0x100);
  ilen = s_StrIToA (CRUTCH2, maxStrIToALen+1, i); // sys fx.
  ConWrite (CONSOLE, CRUTCH2, ilen);
  Delete2(refnum, CRUTCH2);
}
```

4.8 Indexing stuff

The following function simply retrieves the reference for the index library, or zero if this is clear.

```
UInt16 GetIndexRef (UInt16 refnum)
{
  UInt16 idxnum; // redundant.
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  if (! gl->CANINDEX)
     { return 0;
     }; // fail if indexing disabled!

  idxnum = gl->IDXLIB; // clumsy.
  return (idxnum);
}
```

We also need a method of turning off indexing, if something has gone horribly wrong!

```
void TurnOffIndexing (UInt16 refnum)
{
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  gl->CANINDEX = 0;  // turn off
}
```

4.9 Primitive string and memory functions

Here are some primitive functions.

xCopy is straightforward (but clumsy):

```
Int16 xCopy (Char * dest, Char * xsrc, Int16 cnt)
{ if (! dest)
     { return 0;
     };
  if (! xsrc)
     { return 0;
     };
  while (cnt > 0)
    { *dest++ = *xsrc++;
      cnt --;
    };
  return 1;
}
```

The following read, write and even add to values in a 16 bit integer stored in big-endian format. Nasty but appear robust! We check for a dud pointer but it's overkill to pass a library handle to these routines so that errors can be written. They are clumsy enough already! See also NUMERIC library.

```
Int16 ReadInt16X (Char * mP)
{ return(   (*(mP+1) & 0xFF ) + ( (*(mP))<<8 )    );
}

Int16 WriteInt16X (Char * myptr, Int16 datum)
{ if (! myptr)
     { return 0;
     };
  * myptr = datum / 256;
```

```
  * (myptr+1) = datum % 256;
  return 1;
}

Int16 AddInt16X (Char * myptr, Int16 datum)
{ if (! myptr)
     { return 0;  // ERROR
     };
  datum += (*(myptr+1) & 0xFF ) + ( (*(myptr))<<8 );
  * myptr = datum / 256;
  * (myptr+1) = datum % 256;
  return 1;
}
```

xNew is similar to the function in the utility section of the main program, but we don't track blocks. We return a pointer to the start of the allocated memory segment.

```
Char * xNew ( Int16 memsize)
{ MemHandle memH=0; // could trim a few
  MemPtr memP=0;    // bytes here.
  memH = s_MemHandleNew(memsize);
  if (! memH)
     { return 0;
     };
  memP = s_MemHandleLock(memH);
  if (! memP)
     { return 0; //
     };
  return ((Char *) memP); //pointer to locked new memory.
}
```

Delete frees up a pointer allocated by xNew, with no tracking.

```
Int16 Delete (MemPtr memP)
{ if (! memP)
     { return 0;
     };

  if (! s_MemPtrUnlock (memP))
     { return 0;
     };
  if (! s_MemPtrFree (memP))
     { return 0;
     };
  return 1; // success
}
```

s_DmWrite is similar to the routine in the main section wraps.cpp, but here we have a fifth (maximum) argument to prevent us writing off the top of the stack. The sum of the second and fourth arguments must not *exceed* the fifth.

```
Int16 s_DmWrite (void *recordP, UInt32 offset, const void *srcP,
                 UInt32 bytes, UInt32 max)
{ Int16 ok;
  if (recordP == 0)
   { return 0;
   };
  if (bytes == 0)  // its acceptable to have zero length!
   { return 1;
   };
  if (srcP == 0)
   { return 0;
   };
  if (bytes + offset > max)
     { return 0;   // fail if no space
     };
  ok = DmWriteCheck(recordP, offset, bytes); // clumsy
  if (ok != errNone)
     { return 0;
     };
  ok = DmWrite (recordP, offset, srcP, bytes);
  return (! ok);
}

Int16 s_MemHandleUnlock (MemHandle h)
{ Int16 ok;
  if (h == 0)
   { return 0;
   };
  ok = MemHandleUnlock (h);
  if (!ok) { return 1; };
  return 0;
}

Int16 s_DmCloseDatabase (DmOpenRef dbP)
{ Int16 ok;
  if (dbP == 0)
   { return 0;
   };
  ok = DmCloseDatabase (dbP);
  if (!ok) { return 1; };
  return 0;
}

Int16 s_DmReleaseRecord (DmOpenRef dbP, UInt16 index,
```

```
                          Boolean dirty)
{ Int16 ok;
  if (dbP == 0)
   { return 0;
   };
  ok = DmReleaseRecord (dbP, index, dirty);
  if (!ok) { return 1; };
  return 0;
}


MemHandle s_DmGetRecord (DmOpenRef dbP, UInt16 index)
{
  return DmGetRecord (dbP, index);
}
```

DmCloseDatabase returns zero if *no* error occurred, a nonzero value if an error did occur.

```
Int16 PalmFileClose (DmOpenRef myDBref) // symmetry alone. Hmm.
{ if (! s_DmCloseDatabase(myDBref))
     { return 0; //fail
     };
  return 1; //success.
}

UInt16 s_DmFindSortPosition (DmOpenRef dbP, void *newRecord,
                             SortRecordInfoPtr newRecordInfo,
                             DmComparF *compar, Int16 other)
{
  if (dbP == 0)
   { return 0;
   };
  if (newRecord == 0)
   { return 0;
   };
  if (compar == 0)
   { return 0;
   };
  return DmFindSortPosition (dbP, newRecord, newRecordInfo,
                             compar, other);
}

MemHandle s_DmQueryRecord (DmOpenRef dbP, UInt16 index)
{
  return DmQueryRecord (dbP, index);
}
```

xFill: fill with a single character.

```
Int16 xFill (Char * p0, Int16 slen, Char c)
{   // okay, there is a faster way. Check out PalmOS dox.
  while (slen > 0)
    { * p0 = c;
      p0 ++;
      slen --;
    };
 return 1;           // success
}
```

Compare two strings, returning 0 if equal, -1 if p0 under p1, otherwise 1. Case sensitive standard ASCII collation.

Note the bloody problem with 8 bit characters from 0x80 to 0xFF inclusive: the standard (stupid) C++ is to compare as signed integers, which really screws us around. Hence the cast to [eugh] UInt8.

```
Int16 xCompare (Char * p0, Char * p1, Int16 slen)
{ while (slen > 0)
    { if (*p0 != *p1)
         { if (((UInt8)*p0) > ((UInt8)*p1)) { return 1; };
           return -1;
         };
      slen --;
      p0 ++;
      p1 ++;
    };
  return 0;                  // identical strings.
}
```

xSame compares two strings of given length.

```
Int16 xSame (Char * s0, Int16 d0, Char * s1, Int16 d1)
{ if (d0 != d1)
     { return 0;  // can't be the same if different length
     };
while (  (d0 > 0)
       &&(* s0++ == * s1++ )
      )
    { d0 --;
    };
  return (d0 == 0);    // if d0 == 0, the same
}
```

Scan accepts two strings, the first being a string to search within, and the second the string being sought within the first. It must be optimised (Boyer-Moore?) as it is currently *very* slow. It returns the offset of the first byte *after* the first occurrence of the sought string within the target!

```
Int16 Scan (Char * inme, Int16 inlen, Char * sought, Int16 slen)
{
  Int16 lgth = 1+inlen-slen; // max number of characters to search
  while (lgth)
    { if (*inme++ == *sought)
         { if (xSame (inme, slen-1, sought+1, slen-1))
              { return slen + (1+inlen-slen) - lgth;    //
              };
         };
      lgth --;
    };
  return 0;
}
```

Returning 1+ offset allows us to check first place and return a meaningful result.

```
Int16 xSame2(Char *s0, Char *s1, Int16 d1)
{ return xSame (s0, d1, s1, d1);
};
```

4.9.1 Read an integer

The following routine is very primitive, and will only read positive integers, failing (with -1) on negative ones, at present.

```
Int32 Ascii2I32 (Char * P, Int16 ilen)
{ Int32 i;
  Char c;

  i = 0;
  while (ilen > 0)
    { c = *P++;
      ilen --;
      if ((c < '0') || (c > '9')) // test me with char = e.g. 0xF0
         { return -1;             // fail
         };
      c -= '0';
      i = (i*10) + c;
    };
  return i;
}
```

4.9.2 Thirty two bit integer read

ReadInt32 has not one but two limitations.

1. It won't work on an ARM processor, for which a different routine must be written, as we *always* store our long integers in big-endian format;
2. The offset read from must be on an integral boundary divisible by 4. If not, the program will croak.

Use with caution; if in doubt, use SafeReadInt32, which has neither of the above limitations.

```
Int32 ReadInt32 (Char * myptr)  // hi byte is stored first
{
 return * ((Int32 *) myptr);   // need to FIX IF ARM PROCESSOR (little endian)
}
```

Here's the 'safer version':

```
Int32 SafeReadInt32 (Char * myptr)  // hi byte is stored first
{
  return   (((UInt32)*myptr)<<24) +
           (((UInt32)((UInt8)*(myptr+1)))<<16) +
           (((UInt16)*(myptr+2))<<8) +
           ((UInt8)*(myptr+3));
}
```

Eugh. Try leaving out the leftmost (UInt8) cast and see what happens!

WriteInt32 only writes to 4-byte integral boundaries (see usage).

```
Int16 WriteInt32 (Char * myptr, Int32 datum) // big endian
{
 Int32 * p2;
 if (! myptr)
     { return 0; // attempt to write to null pointer aargh.
     };
p2 = (Int32 *) myptr; // likewise NOT for ARM processor
* p2 = datum;
return 1;
}
```

4.9.3 Advance

As in main program usage.

```
Int16 Advance (Char * myptr, Int16 limit, WChar target)
{ WChar ch;                   // NOTE: limit is _signed_
  Int16 howfar = 0;

  while (limit > 0)
```

```
    { ch = *myptr; // NNNOOOO. SIMPLIFY THIS! [fix me]
      howfar ++;
      if (ch == target)
         { return howfar;
         };
      myptr ++;
      limit --;
    };
  return 0; // fail.
}

Char UPPERCASE (Char c)
{ if (  (c >= 'a')
      &&(c <= 'z')
      ) { return (c-0x20);
        };
  return c;
}
```

4.9.4 Insert

Given a string **stringi** to insert at the start of string **dstring**, copy the string into dstring, after moving the remaining characters in dstring up. We must also provide **dmax** to ensure that there is no buffer overflow, as well as the actual lengths of the two strings.

```
Int16 InsertString (Char * dstring, Int16 dlen, Int16 dmax,
                    Char * stringi, Int16 leni)
{ Int16 i;
  Char * sP;
  Char * dP;

  if (dlen < 0)
     { return 0; // fail
     };
  if ( (dmax - dlen) < leni )
     { return 0; // fail
     };

  sP = dstring+dlen-1;
  dP = sP + leni;
  i = dlen;
  while (i > 0)
    { *dP-- = *sP--;
      i--;
    };
  i = leni;
```

```
  sP = stringi+leni-1;
  while (i > 0)
    { *dP-- = *sP--;
      i --;
    };
  return leni; // characters inserted
}
```

4.9.5 Other stuff

LUPSAME is similar to xSame, but forces left string s0 to UPPERCASE!

```
Int16 LUPSAME (Char * s0, Int16 d0, Char * s1, Int16 d1)
{ if (d0 != d1)
     { return 0; // can't be the same if different length
     };
while (  (d0 > 0)
       &&( UPPERCASE(*s0++) == * s1++ )
      )
    { d0 --;
    };
  return (d0 == 0); // if d0 == 0, the same
}
```

The following is copied from NUMERIC.c.

```
Int16 CountItems (Char * myptr, Int16 max, Char c)
{ Int16 cnt = 0;
  while (max > 0)
    { if (* myptr++ == c)
         { cnt ++;
         };
      max --;
    };
  return cnt;
}
```

BetterCompare returns -1 if L less than R, 0 if equal, otherwise 1. Fixed for characters over 0x7F — see usage of (UInt8).

```
Int16 BetterCompare(Char * L, Int16 llen, Char * R, Int16 rlen)
{
  if (llen > rlen)
     { while (  (rlen > 0)
              &&( * L++ == * R++ )
             )
         { rlen --;
```

```
        };
      if (! rlen)         // end of line, so L *must* be greater
         { return 1;      // signal L > R
         };
      if (((UInt8)*(L-1)) > ((UInt8)*(R-1)))
         { return 1;      // L > R
         };
      return -1;          // L < R
    };
  rlen -= llen;           // know that rlen is >= llen.
  while (  (llen > 0)
         &&( *L++ == * R++)
        )
    { llen --;
    };
  if (! llen)             // end of line
     { if (! rlen)        // if same length
          { return 0;     // identical
                    // [although FUNCTION NAME COMPARISON = NEVER]
          };
       return -1;         // L < R
     };
  if ( ((UInt8)*(L-1)) > ((UInt8)*(R-1)) )
     { return 1;          // L > R
     };
  return -1;              // L < R
}
```

CompareStackItems: iA, iB are integer offsets into STACK, return -1 if A under B, 0 if equal, otherwise 1. Hmm. What about floats? (ok).

```
Int16 CompareStackItems (Char * STACK, Char * STACKSTRING,
                         Int16 iA, Int16 iB)
{ // NB if dud compare return value < -1 (!?)

  Char tA;
  Int16 lA;
  Int16 lB; // lengths
  Char * pA;
  Char * pB;

  tA = *(STACK + iA + 15);
  if (tA != *(STACK + iB + 15))
     { return -SqErBadTypeCompare;
     };
  // do not need to switch on type, as
  // all types are stored appropriately even float.
  // (assuming integers are big-endian)
```

```
  lA = 0x0F & (*(STACK + iA + 14));
  lB = 0x0F & (*(STACK + iB + 14));

  if (lA > 14)
     { lA = *((Int16 *)(STACK+0+iA));
       pA = STACKSTRING + *((Int16 *)(STACK+2+iA));
     } else
     { pA = STACK+iA;
     };
  if (lB > 14)
     { lB = *((Int16 *)(STACK+0+iB));
       pB = STACKSTRING + *((Int16 *)(STACK+2+iB));
     } else
     { pB = STACK+iB;
     };
  return BetterCompare(pA, lA, pB, lB);
}
```

FindStackPosition: given a buffer (sort area) of i two-byte indices into data items on STACK, get ith item off stack and find its position within the buffer, returning this position. Insertion is to the *left* of the specified position (e.g. zero) means 'put me in at the start and shift everything else up to the right'. This routine is NOT now order preserving for equal items, but could be made so by identifying equal items and then moving right until non-equal item found, inserting above the last equal item (assuming caller still also moves from left to right during its traverse of the stack.

```
Int16 FindStackPosition(Char * STACK, Int16 bottom, Char * sortarea,
                        Int16 itm, Char * STACKSTRING)
{ // we might also submit top, so we can ensure
  // that no value in sortarea is ridiculous[?]

//  return 0; // we will try simple debug (reversal)

  Int16 refitem = bottom + itm*XVI; // item to compare others with
  Int16 i; // we move i around looking for correct position
  Int16 c;
  Int16 thisitem;
  Int16 left=0;
  Int16 right = itm-1; // indices into sortarea,
                       // top item is currently at itm-1

  while (left <= right)  // NB [check me] cf can we shorten FindPosition!
    { i = (left+right) / 2;
      thisitem = bottom + XVI * (*((Int16*)(sortarea+2*i)));
      c = CompareStackItems (STACK, STACKSTRING, thisitem, refitem);
```

```
      if (c < -1)
         { return c;  // fail if error (cannot compare dissimilar items)
         };
      if (c < 0) // if current item is below reference item,
         { left = i+1;                // move right
         } else
         { if (c > 0)                 // otherwise, if above,
              { right = i-1;          // move left
              } else                  // must be EQUAL
              { return i;             // will be inserted TO LEFT
              };
         };
    };
  return left; // found position
}
```

4.10 Stack handling

PushItem pushes to stack.

```
Int16 PushItem (Char * STACK, Char * STACKSTRING, Char * itm,
                Int16 ilen, Char itype, Int16 scale)
{
  Int16 top;
  Int16 max;
  Int16 strtop;
  Int16 strmax;
  Char plen;
  Char sc;
  Char * null8 = "\x0" "\x0" "\x0" "\x0" "\x0" "\x0" "\x0" "\x0";

  if (! STACK)
      { return -SqErPushNoStack; // fail if no stack
      };

  top = *((Int16 *)(STACK+oTOP));
  max = *((Int16 *)(STACK+oMAX));
  if ((max - top) < XVI)
     { return -SqErPushStackFull; // no space for new item on stack
     };
  if (  !(s_DmWrite(STACK, top, null8, 8, SMAX)) // slow but safe
      ||!(s_DmWrite(STACK, top+8, null8, 8, SMAX))
     )
     { return -SqErPushFailed; // stack write failed
     };

  s_DmWrite(STACK, top+15, &itype, 1, SMAX);
  sc = (Char) scale;
```

```
  s_DmWrite(STACK, top+13, &sc, 1, SMAX);
```

In the following, we push NULL onto the stack, despite error, as this is the best possible error correction!

```
  if (ilen < 0)
     {
       top += XVI;
       s_DmWrite(STACK, oTOP, &top, 2, SMAX);
       return -SqErPushBadLength;  // silly length
     };
  if (! itm)
     {
       top += XVI;
       s_DmWrite(STACK, oTOP, &top, 2, SMAX);
       return -SqErPushNothing; // no item
     };

  if (ilen <= 14)
     { s_DmWrite(STACK, top, itm, ilen, SMAX);
       plen = (Char) ilen;
     } else
     { plen = (Char) 15;
       strtop = *((Int16 *)(STACKSTRING+oTOP));
       strmax = *((Int16 *)(STACKSTRING+oMAX));
       if ((strmax - strtop) < ilen)
          { return -SqErLongPushFailed;
            // no string space for long stack string
          };
       s_DmWrite(STACK, top, &ilen, 2, SMAX);
       s_DmWrite(STACK, top+2, &strtop, 2, SMAX);
       if (! s_DmWrite(STACKSTRING, strtop, itm, ilen, MAXSS) )
          { return -SqErPushLong;
            // stack write of long string failed
          };
       strtop += ilen;
       s_DmWrite(STACKSTRING, oTOP, &strtop, 2, MAXSS);
     };

  s_DmWrite(STACK, top+14, &plen, 1, SMAX);

  top += XVI;
  s_DmWrite(STACK, oTOP, &top, 2, SMAX);
  return 1;  // success
}
```

InsertItem: A common function used by SORT and SortFx. Given A an array of 2-byte integers, insert itm at word offset posn. Size of array is alen +items+

which is 2*alen bytes. In other words alen, posn are both WORD offsets. Oh for a rep movsw! might check that posn <= alen, and fail if not!

```
Int16 InsertItem( Char * A, Int16 alen, Int16 posn, Int16 itm)
{  Int16 i;
   Char * P;

  if (posn > alen)
      { return -SqErInsortFailed;
      };
   if (posn < 0)
      { return -SqErInsortFailed; //Can't insert at -ve posn
      };
   P = A + alen*2;  // 2 bytes per item
   i = alen-posn;
   while (i > 0)   // can improve on:
     { *((Int16*)(P)) = *((Int16*)(P-2)); // ugly. [fix me]
        P-=2;
        i --;
     };
   *((Int16 *)P) = itm;
   return 1;                                    // success
}
```

Standard setting of mark on stack.

```
Int16 SetMark(Char * STACK)
{
  Int16 bottom;
  Int16 top;
  Int16 i;
  Int16 marked;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -SqErMarkEmpty; // insufficient stack
     };

  marked = *((Int16 *)(STACK+oMARKS));
           // get index of where to store current mark
  if (marked >= 32) // excessive: max marks is 32
     { return -SqErMarkMax;
     };
  s_DmWrite(STACK, 16 + 2*marked, &bottom, 2, SMAX);
            // store current mark (bottom)
  marked ++;
  s_DmWrite(STACK, oMARKS, &marked, 2, SMAX);
```

```
            // store mark count

  top -= XVI;
  s_DmWrite(STACK, oTOP, &top, 2, SMAX); // pop just 1 item
  if (* (STACK+top+15) != 'I')
     { return -SqErMarkArg;
     };
  i = (Int16) (*((Int32 *)(STACK+top))); // get int32 as int16
  if (i > STACKCOUNT)          // -> (won't work)
     { return -SqErMarkEmpty;  // fail: insufficient stack
     };
  i *= XVI;
  if (top - i < bottom)
     { return -SqErMarkEmpty; // clumsy
     };
  bottom = top - i;
  s_DmWrite(STACK, oSTART, &bottom, 2, SMAX); // set new bottom
  return 1;
}
```

Unmarking does the converse, resetting current top to marked position.

```
// 17-5-2005: [FIX ME! --- need to clear STACKSTRING AS WELL]

Int16 ClearMark(Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 marked;
  Int16 top;
  Int16 SStop;
  Char * B;
  Char * T;

  marked = *((Int16 *)(STACK+oMARKS)); // get index of mark
  if (marked < 1)                      // nada
     { return -SqErMarkMin;
     };

  // here find out if stackstring referenced; if so, clear it!
  T = STACK + *((Int16 *)(STACK+oTOP));
  top = *((Int16 *)(STACK+oSTART)); // top down to old bottom!
  B = STACK + top;

  while (B < T)
    { if ( (0x0F & *(B+14)) > 14)
         { SStop = *((Int16 *)(B+2));
           // get reference to start of lowest stackstring string
           s_DmWrite(STACKSTRING, oSTART, &SStop, 2, MAXSS);
```

```
          // update SS
          B = T; // force exit
        };
     B += XVI;
   };
  // sort out top..
  s_DmWrite(STACK, oTOP, &top, 2, SMAX);
          // remove whole chunk from stack

  marked --; // go back to stored 'bottom' value
  s_DmWrite(STACK, oMARKS, &marked, 2, SMAX); // write diminished count
  bottom = *((Int16 *)(STACK+16 + 2*marked));
  s_DmWrite(STACK, oSTART, &bottom, 2, SMAX); // reset bottom

  return 1;
}
```

UnmarkOnly does *not* take the stack top back.

```
Int16 UnmarkOnly(Char * STACK)
{ Int16 bottom;
  Int16 marked;
  marked = *((Int16 *)(STACK+oMARKS));
    // get index of where to store current mark
  if (marked < 1) // nada
     { return -SqErMarkMin;
     };
  marked --; // go back to stored 'bottom' value
  s_DmWrite(STACK, oMARKS, &marked, 2, SMAX);
  bottom = *((Int16 *)(STACK+16 + 2*marked));
  s_DmWrite(STACK, oSTART, &bottom, 2, SMAX);
  return 1;  // similar to ClearMark above
}
```

5 Maximum of a list

Submit items on marked stack, returning the biggest item on the stack and a return value of 1, or, if no items found, return zero. Error code is a value under zero.

```
Int16 DoMax (Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 here;

  Int16 ok;
  Int16 maxitem;
  Int16 newSStop=0;
  Int16 newSSsource;
  Int16 newSSlen;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  if (top == bottom)  // special case: nothing
     { return 0;
     };
  top -= XVI;
  if (top == bottom)  // special case: only one item ('max')
     { return 1;      // success
     };
  if (top < bottom)  // stack stuffed up
     { return -SqErNoMax;
     };
  maxitem = top;
  here = bottom;

  while (here < top) // while still more than one item on stack..
    { ok = CompareStackItems (STACK, STACKSTRING, here, maxitem);
           // a little clumsy
      if (ok < -1)
         { return ok;   // fail
         };
      if (ok > 0)  // iff here item is greater, it is maxitem!
         { maxitem = here;
         };
      // check to find new stackstring bottom:
      //   (ie. first reference to stackstring)
      if (! newSStop)
         { if ((0x0F & (*(STACK+here+0x14))) > 14)
              { newSStop = *((Int16 *)(STACK+2+here));
              };
         };
```

```
      here += XVI;
    };

  // move max item to bottom, update top:
  if (bottom != maxitem)
     { s_DmWrite(STACK, bottom, STACK+maxitem, XVI, SMAX);
     };
  top = bottom + XVI;
  s_DmWrite(STACK, oTOP, &top, 2, SMAX);
    // new top is just one item on stack

  // fix up stackstring:
  if (newSStop)       // if no bottom, stackstring is ok, else...
     { if ((0x0F & (*STACK+bottom+14)) > 14) // current itm long
          { newSSsource = *((Int16 *)(STACK+2+bottom));
            newSSlen = *((Int16 *)(STACK+0+bottom));
            if (newSSsource != newSStop)
               { s_DmWrite(STACKSTRING, newSStop,
                  STACKSTRING+newSSsource, newSSlen, MAXSS);
               }; // ? assuming too much on part of PalmOS DmWrite?
            newSStop += newSSlen; // add length of item just copied
          }; // else new top will simply be newSStop!
       s_DmWrite(STACKSTRING, oTOP, &newSStop, 2, MAXSS);
     };
  return 1; //ok
}
```

6 Minimum

Similar to DoMax above. (We might even share code).

```
Int16 DoMin (Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 here;

  Int16 ok;
  Int16 minitem;
  Int16 newSStop=0;
  Int16 newSSsource;
  Int16 newSSlen;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  if (top == bottom)  // special case: nothing
     { return 0;
     };
  top -= XVI;
  if (top == bottom)  // special case: 1 item ('max')
     { return 1;      // success
     };
  if (top < bottom)  // nothing to min?
     { return -SqErNoMin;
     };
  minitem = top;
  here = bottom;

  while (here < top) // while still >1 item on stack..
    { ok = CompareStackItems (STACK, STACKSTRING, here, minitem);
           // a little clumsy
      if (ok < -1)
         { return ok;   // fail
         };
      if (ok < 0)
      // ONLY DISSIMILARITY BETWEEN THIS FX AND MAX! [fix]
         { minitem = here;
         };
      // check to find new stackstring bottom: (ie. 1st ref to stackstring)
      if (! newSStop)
         { if ((0x0F & (*(STACK+here+0x14))) > 14)
              { newSStop = *((Int16 *)(STACK+2+here));
              };
         };
```

```
      here += XVI;
    };

  // move min item to bottom, update top:
  if (bottom != minitem)
     { s_DmWrite(STACK, bottom, STACK+minitem, XVI, SMAX);
     };
  top = bottom + XVI;
  s_DmWrite(STACK, oTOP, &top, 2, SMAX);
    // new top is just one item on stack

  // fix up stackstring:
  if (newSStop)  // no bottom, stackstring is ok, otherwise...
     { if ((0x0F & (*STACK+bottom+14)) > 14) // current is long
          { newSSsource = *((Int16 *)(STACK+2+bottom));
            newSSlen = *((Int16 *)(STACK+0+bottom));
            if (newSSsource != newSStop)
               { s_DmWrite(STACKSTRING, newSStop,
                     STACKSTRING+newSSsource, newSSlen, MAXSS);
               }; // as above (see DoMax)?
            newSStop += newSSlen; // +length of item just copied
          }; // else new top will simply be newSStop!
       s_DmWrite(STACKSTRING, oTOP, &newSStop, 2, MAXSS);
     };
  return 1; //ok
}
```

7 Distinct

DISTINCT is tricky and slow. For now, we only permit 1 column (1 item per row), but later we must have more. So we take topmost stack item (integer) and if it's not 1, return error. [fix me: later develop so more than 1 can be taken off]

If duplicate items exist, then excise the first duplicate found, and so on. is slow because: a. $O(n^2)$ – compare first with each of rest, and so on; b. Excision is cumbersome. Speed a little by flagging excised items, and then copying all non-excised items down, item by item. We mark excised item by setting type to zero!

```
Int16 DoDistinct (Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 current;
  Int16 nextitem;
  Int16 ok;
  Char c = 0x0; // for clearing items, see usage
  Int16 deadSS;
  Int16 SSlen;
  Int16 SSsrc;
  Int16 itemsinarow;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top == bottom)
     // special case: nothing except the column count specifier
     { return 0;
     };
  if (top < bottom)  // nothing to do
     { return -SqErNoDistinct;
     };
  if (*(STACK+top+15) != 'I')
     { return -SqErBadDistinct;  //
     };
  itemsinarow = *((Int32 *)(STACK+top)); // get number of items
  if (itemsinarow != 1)
     { return -SqErBadDistinct;  // temporary
     };
  top -= XVI; // down to topmost item
```

We might also address the special case where there's only one item [check me]

7.1 Identify duplicates and flag them as deleted

```
current = bottom;
while (current < top)  // each item, cf with ALL subsequent items
    // if identical, then flag subsequent item(s) as deleted
  { if ( *(STACK+current+15) ) // if not already cleared
      { nextitem = current + XVI;
        while (nextitem <= top)
          { if (*(STACK+nextitem+15)) // NOT ALREADY CLEARED?
               { ok = CompareStackItems (STACK, STACKSTRING,
                                         current, nextitem);
                 if (ok < -1)
                    { return ok; // fail
                    };
                 if (! ok) // actually, if EQUAL!
                    { s_DmWrite(STACK, nextitem+15, &c, 1, SMAX);
                      // clear duplicate item!
                    };
               };
            nextitem += XVI;
          };
      };
    current += XVI;
  };
```

7.2 Clean out all deleted items

Tricky!

```
current = bottom;
deadSS = 0;
while (current <= top)             // NB top is AT last item, not above
  { while (  (bottom <= top)       // MOVE bottom up to first DELETED item
           &&( *(STACK+bottom+15) )// type is NOT zero
          )
      { bottom += XVI;
      }; // now either bottom > top OR bottom at a null value
    if (current < bottom) // bottom now invalid, we want current above,ok
       { current = bottom;           // go to that deleted item
       };
    while (  (current <= top)        // MOVE current up to first VALID item
           &&(! (*(STACK+current+15)))// continue if type IS zero
          )
      {
// 1. check whether deleted item had SS extension (unless already found)!
        if (! deadSS)
           { if ( (0x0f & *(STACK+current+14)) > 14)
                { deadSS = *((Int16 *)(STACK+2+current));
```

```
                    // get 1st dead string on stackstring
                  };
               };
            // 2. move to next item, so that we can test it
            current += XVI; // find first still-valid item!
          };
        // NOW either current above top OR current at first valid item
        if (current <= top)
           { s_DmWrite(STACK, bottom, STACK+current, XVI, SMAX);
                      // copy over valid item
             if ( (0x0F & *(STACK+current+14)) > 14) //is this item long?
                { SSlen = *((Int16 *)(STACK+0+current));
                  SSsrc = *((Int16 *)(STACK+2+current));
                  if (deadSS)
                     { s_DmWrite(STACKSTRING, deadSS,
                          STACKSTRING+SSsrc, SSlen, MAXSS);
                       deadSS += SSlen;
                     }; // else no dead long itm, so stackstring unchanged!
                };
             current += XVI;
             bottom += XVI;
           };
    };

  // finally, jack up top of stack:
  s_DmWrite(STACK, oTOP, &bottom, 2, SMAX);
  if(deadSS)
    { s_DmWrite(STACKSTRING, oTOP, &deadSS, 2, MAXSS);
    };
 return 1; //ok
}
```

8 Sorting

Here follow a few routines to implement a heapsort. The catch with heapsort is the recursion, with the chance that a PalmOS version running under small memory conditions will run out of memory and overflow the stack. This catastrophe would seem less likely on modern PalmOS PDAs.

8.1 Preliminary heapsort routines

```
Int16 heapify(UInt16 refnum, Int16 heapsize, UInt16 * arr, UInt16 i,
              Char * STACK, Char * STACKSTRING)
{ UInt16 great;
  Int16 ok;
  UInt16 temp;
  UInt16 lft = 1+(2*i); // LEFT
  UInt16 r   = 2+(2*i); // RIGHT

   +OPTIONAL
   ConAsc(refnum, "\n l=", fDEBUG_SQK); //
   ConI  (refnum, lft, fDEBUG_SQK);
   ConAsc(refnum, ",r=", fDEBUG_SQK); //
   ConI  (refnum, r, fDEBUG_SQK);
   -OPTIONAL

  if (lft < heapsize)
     { ok = CompareStackItems (STACK, STACKSTRING, arr[lft], arr[i]);
       if (ok > 0)
          { great = lft;
          } else
          { if (ok < -1)
               {return -1; // fail
               }
            great = i;
          };
     } else
     { great = i;
     };
  if (r < heapsize)
       { ok = CompareStackItems (STACK, STACKSTRING, arr[r], arr[great]);
         if (ok > 0)
            { great = r;
            } else
            { if (ok < -1)
                 { return -2; // fail
                 };
            };
       };
```

```
  if (great != i)
     {

   +OPTIONAL
   ConAsc(refnum, "@", fDEBUG_SQK);
   ConI  (refnum, i, fDEBUG_SQK);
   ConAsc(refnum, ":", fDEBUG_SQK);
   ConI  (refnum, arr[i], fDEBUG_SQK);
   ConAsc(refnum, "<->@", fDEBUG_SQK);
   ConI  (refnum, great, fDEBUG_SQK);
   ConAsc(refnum, ":", fDEBUG_SQK);
   ConI  (refnum, arr[great], fDEBUG_SQK);
   -OPTIONAL

temp = arr[i];
       arr[i] = arr[great];
       arr[great] = temp;

       ok = heapify(refnum, heapsize,
         arr, great, STACK, STACKSTRING); // NOTE RECURSION
       return ok; // clumsy.
     };
  return 1; // success
}

Int16 BuildMaxHeap(UInt16 refnum, Int16 heapsize, UInt16 * arr,
                   Char * STACK, Char * STACKSTRING)
{ Int16 i;
   +OPTIONAL
   ConAsc(refnum, "\n Build(size:", fDEBUG_SQK); //
   ConI  (refnum, heapsize, fDEBUG_SQK);
   ConAsc(refnum, ")", fDEBUG_SQK); //
   -OPTIONAL

  for (i = (heapsize - 1) / 2; i >= 0; i--)
  { if ( heapify(refnum, heapsize,
                 arr, i, STACK, STACKSTRING) < 0)
       { return -1; // fail
       };
    // debug:
    //     ConTx (refnum, "]", 1, 0);
  };
  return 1; // ok.
}
```

8.1.1 SubHeapSort

Given an array **arr** which indexes the stack, using UInt pointers to actual offsets within the **STACK**, as well as extensions of these items in **STACKSTRING**, perform the actual heapsort. The items in **arr** are rearranged. Comparisons use CompareStackItems (STACK, STACKSTRING, thisitem, refitem); which returns -1 if item indexed by thisitem ¡ item indexed by refitem, 0 if equal, 1 if greater.

```
Int16 SubHeapSort(UInt16 refnum, Int16 heapsize, UInt16 * arr,
                  Char * STACK, Char * STACKSTRING)
{ Int16 temp;
  Int16 i;
  if (BuildMaxHeap(refnum, heapsize, arr, STACK, STACKSTRING) < 0)
     { ConTx (refnum, "?BLD", 4, 0);
       return -1; // fail
     };

   +OPTIONAL
   ConAsc(refnum, "\n  {", fDEBUG_SQK); //
   -OPTIONAL

  for (i = heapsize; i > 0; i--)
  {
    +OPTIONAL
    ConI  (refnum, i, fDEBUG_SQK);
    ConAsc(refnum, ":", fDEBUG_SQK); //
    -OPTIONAL
   temp = arr[0];
    arr[0] = arr[heapsize - 1];
    arr[heapsize - 1] = temp;
    heapsize = heapsize - 1;
    if (heapify(refnum, heapsize,
                arr, 0, STACK, STACKSTRING) < 0)
       { return -2; // fail
       };
   +OPTIONAL
   ConAsc(refnum, "}", fDEBUG_SQK); //
   -OPTIONAL
  };
  return 1; // ok.
}
```

This routine returns 1 = sorted; error code is signalled by value under zero.

8.1.2 HeapSort

Finally, the actual HeapSort routine. Submit the top of the stack (as a byte offset from the bottom of the stack, divisible by XVI), the byte offset of the first item to be sorted (**stkpos**), the width of each sort item (**itemwidth**, where 1 means just one XVI-byte item per row, 2 means two of these, and so forth), and **ioffset**, the offset within each row of the item to sort, where 0 is the first row item (XVI bytes long), 1 is the second (at offset XVI) and so forth. The Boolean variable order is zero *unless* the sort order is descending.

We have a particular problem with sorting of items longer than 14 bytes, because these are represented by extensions stored in the STACKSTRING. We identify such items by their length flag (a value over 14, conventionally 15, is stored at offset +14 within the representation of the item on the STACK). We then must re-order these items on the STACKSTRING as well as sorting them! (Compare this with the MySwop routine in the file *ScriptingLib.tex*).

Heapsort supports the SQL convention which allows sorting on a column not specified at the start of the SELECT statement. This means that this column must be *deleted* from the answers, so that even if there is only one item to be sorted, it must be processed in this fashion! The relevant flag is **vanishcolumn**, which, if set, mandates removal of the relevant column from all data returned.

```
Int16 HeapSort(UInt16 refnum, UInt16 stktop, UInt16 stkpos,
               UInt16 itemwidth, UInt16 ioffset, Int16 stackstringtop,
               Char * STACK, Char * STACKSTRING, Boolean order,
               Boolean vanishcolumn)
{
  UInt16 * arr;
  Int16 i;
  Int16 j;
  Int16 ok;
  Int16 heapsize;
  Char * stackArea;
  Char * stackstringArea;
  Int16 sslower;
  Int16 offs;
  Char * sP;
  Int16 xtend;
  Int16 xoffnew;
  Int16 xlen;
  Char * Vdest;
  Int16 newstacksize;

   +OPTIONAL
   ConAsc(refnum, "(stack:", fDEBUG_SQK); //
   ConI  (refnum, stkpos, fDEBUG_SQK);
```

```
 ConAsc (refnum, ",top:", fDEBUG_SQK);
 ConI  (refnum, stktop, fDEBUG_SQK);
 ConAsc (refnum, ",width:", fDEBUG_SQK);
 ConI  (refnum, itemwidth, fDEBUG_SQK);
 ConAsc (refnum, ",offset:", fDEBUG_SQK);
 ConI  (refnum, ioffset, fDEBUG_SQK);
 ConAsc (refnum, ",stkstr top:", fDEBUG_SQK);
 ConI  (refnum, stackstringtop, fDEBUG_SQK);
 ConAsc (refnum, ",order:", fDEBUG_SQK);
 ConI  (refnum, order, fDEBUG_SQK);
 ConAsc (refnum, ")", fDEBUG_SQK);
 -OPTIONAL

// sanity check for ioffset:
if (ioffset >= itemwidth)
   { return -10;
   };

// ensure length is divisible by itemwidth:
if ((stktop-stkpos) % itemwidth) // if remainder:
   { return -20; // fail
   };
heapsize = ((stktop-stkpos)/itemwidth)/XVI ;

// if nothing to sort:
if (heapsize <= 0)
   { return 0; // 'sorted'
   };

arr = (UInt16 *) xNew(2*heapsize); // 2 bytes per UInt
if (! arr)
   { return -30;
   };

    i = 0;

 +OPTIONAL
 ConAsc(refnum, "\n(", fDEBUG_SQK); //
 -OPTIONAL
while (i < heapsize)
  {
    arr[i] = stkpos + i*XVI*itemwidth + ioffset*XVI;
 +OPTIONAL
 ConI  (refnum, arr[i], fDEBUG_SQK);
 ConAsc(refnum, ",{", fDEBUG_SQK);
 ConTx (refnum, STACK+arr[i], 16, fDEBUG_SQK);
 ConAsc(refnum, "}", fDEBUG_SQK);
 -OPTIONAL
```

```
    // arr[i] is OFFSET in bytes relative to STACK start!
    i ++; // clumsy
  };
 +OPTIONAL
 ConAsc(refnum, ")", fDEBUG_SQK); //
 -OPTIONAL

ok = SubHeapSort(refnum, heapsize, arr, STACK, STACKSTRING);
```

The variable heapsize is the number of rows to sort, NOT items or bytes! We next re-order items on the stack, based on the values in **arr**.

```
// DEBUG:
  //     ConTx (refnum, "\nMov:", 5, 0);
  //     ConI (refnum, ok, 0);
  //     ConTx (refnum, ":", 1, 0);

if (ok > 0) // unless failed
   {
     // 1. request buffer area of same size as stack
     stackArea = xNew(stktop-stkpos); // IS > 0
     // might check for success.

     // 2a. determine stackstring size. We must
     // search through each item up to top, until encounter
     // first extended string, and then record this offset
     // as bottom.

     sslower = stackstringtop; // default
     i = 0;
     sP = STACK + stkpos;
     while (  (i < heapsize*itemwidth)
            &&(sslower >= stackstringtop)
           )
       {
         if ( *(sP+14) > 14 )
            { sslower = * ((Int16 *)(sP+2)); // get offset
            };
         sP += XVI;
         i ++;
       };
    // NOW if sslower == stackstringtop, NO extensions!

     // 2b. request stackstring buffer
     stackstringArea = xNew(2+stackstringtop-sslower);
     // if length is zero, we request just 2 bytes
     // to prevent an error!
```

```
    // 3. copy each item over to new buffer area
    i = 0;
    xoffnew = 0; // 0 offset at start

    while (i < heapsize)
      {
+OPTIONAL
ConAsc(refnum, "\n  ", fDEBUG_SQK);
ConI (refnum, i,fDEBUG_SQK);
ConAsc(refnum, ":", fDEBUG_SQK);
-OPTIONAL
        // 4. Get source offset, copy over items:
        offs = arr[i] - ioffset*XVI;
        if (order) // if reverse order:
           { offs = arr[heapsize-i-1] - ioffset*XVI;
           };
        Vdest = stackArea + i*XVI*(itemwidth-vanishcolumn);
        // one item less if vanish sort item!
        j = 0;
        while (j < itemwidth)
          { if(  (! vanishcolumn )
               ||(j != ioffset)
              ) // unless must vanish the item:
               {
+OPTIONAL
ConAsc(refnum, "{", fDEBUG_SQK);
ConTx (refnum, STACK+offs, 16, fDEBUG_SQK);
-OPTIONAL
                 xCopy(Vdest, STACK+offs, XVI);
                 if ( * (Vdest + 14) > 14 ) // if extension present:
                    { xtend = * ( (Int16*) (Vdest + 2) );
                      xlen =  * ( (Int16*) (Vdest) );
                      xCopy (stackstringArea+xoffnew,
                             STACKSTRING + xtend, xlen);
+OPTIONAL
ConAsc(refnum, "|", fDEBUG_SQK);
ConTx (refnum, stackstringArea+xoffnew, xlen, fDEBUG_SQK);
-OPTIONAL
                      * ((Int16 *) (Vdest + 2)) = sslower+xoffnew;

                      // DEBUG:
                      //   ConTx (refnum, "{", 1, 0);
                      //   ConTx (refnum, stackstringArea+xoffnew, xlen, 0);
                      //   ConTx (refnum, "}", 1, 0);
                      // end debug.
                      xoffnew += xlen;
                    };
```

```
                  Vdest += XVI;
                };
             offs += XVI;
  +OPTIONAL
  ConAsc(refnum, "}", fDEBUG_SQK);
  -OPTIONAL
             j ++;
           };
         i ++;
       };

      // 5. overwrite stack area with new values
      newstacksize = heapsize * XVI * (itemwidth-vanishcolumn);
      s_DmWrite(STACK, stkpos, stackArea,
                newstacksize, SMAX); // might check success
      // HERE MUST ALSO REWRITE stack size
      stktop = stkpos + newstacksize;
      s_DmWrite(STACK, oTOP, &stktop, 2, SMAX);

      // 6. overwrite stackstring with new values
      s_DmWrite(STACKSTRING, sslower, stackstringArea,
                 xoffnew, MAXSS); // might check success
      // HERE MUST ALSO REWRITE stackstring size!
      // (as part might have vanished!)
      stackstringtop = sslower+xoffnew;
      s_DmWrite(STACKSTRING, oTOP, &stackstringtop, 2, MAXSS);

      // 7. clean up
      Delete(stackArea);
      Delete(stackstringArea);
    };
  Delete(arr);
  return ok;
}
```

A return value of under 0 signals failure.

8.2 Sort invocation

Given items on the stack, sort them. We need to know how many items there are in a row — we might for example want to sort 3 items per row, as well as whether each item to be sorted is sorted in ascending or descending order. We also need to specify the order of the sort.

We will sort based on up to four items, and each of these four sorts will be either up or down. If we encounter something along the lines of

```
SELECT GAMMA, ALPHA, DELTA, BETA FROM ...
```

```
... ORDER BY BETA ASC, ALPHA DESC, GAMMA ASC
```

...we need a method for encoding the sort. The variable **itemsinarow** is clearly four, but how do we specify the rest? We might pass a linked list, but at present we pass four bytes in the 32 bit integer **sortorder**. The *lowest* order byte specifies the number of first column to sort, the next byte up the number of the second, and so on. If any one of these *bytes* has a value of over 63, then one of two flags might be set (The lower 6 bits still identify the column). If bit #6 is set (mask is 0x40) then the column is a vanishing column; if bit #7 is set, then the sort must be performed in reverse (descending) order from the normal ascending sort.

Note that this implies a maximum of 63 columns, i.e. itemsinarow cannot be over 63. It is also evident that the first column is column 1 (not zero) and a zero value means that we don't try to identify any (more) columns to sort on.

Vanishing columns are interesting. In SQL it is permissible to sort on a column not specified as one of the columns to SELECT. We select such columns, but flag these as vanishing, and after we've sorted, we remove these columns!

At present we only sort on *one* column; the remaining three columns to 'sub-sort' on are for future development. We also only vanish one column.

```
Int16 DoSort (UInt16 refnum, Char * STACK, Char * STACKSTRING,
              Int16 itemsinarow, Int32 sortorder )
{
  Int16 bottom;
  Int16 top;
  Int16 items;
  Int16 ok;
  Int16 sstop;       // top of stackstring
  Int16 ioffset;
  Int16 vanishcolumn = 0; // do not vanish sort column, by default.
  Boolean order=0;   // 0 means ascending, 1 = descending.

   +OPTIONAL
   ConAsc(refnum, "(sorting..[", fDEBUG_SQK); //
   -OPTIONAL

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  items = (top - bottom)/XVI;
  if (items < 0)  // insufficient arguments
     { return -SqErNoSort;
     };
  if (items == 0)
     { return 0; // nothing to sort!
     };
```

8.2.1 Perform sort

We use our new heapsort function. The variable **top** points to just above the topmost item, and bottom is the first item on the stack.

```
 ioffset =  ( (UInt8) (sortorder & 0x0FF) );
 if (ioffset & 0x80)
    { order = 1;  // descend
      ioffset &= 0x7F; // remove flag
    };
 if (ioffset & 0x40)
    { vanishcolumn = 1;
      ioffset &= 0x3F; // remove flag
    };

   +OPTIONAL
   ConI(refnum, vanishcolumn, fDEBUG_SQK); //1=vanish!
   ConAsc(refnum, "]", fDEBUG_SQK); //
   -OPTIONAL

  sstop = *((Int16 *)(STACKSTRING+oTOP)); // get top of stack
  //  ssmax = *((Int16 *)(STACKSTRING+oMAX)); // and max
  ok = HeapSort(refnum, top, bottom,
                itemsinarow, ioffset-1, sstop,
                STACK, STACKSTRING, order,
                (Boolean) vanishcolumn);
  // NOTE that we -- ioffset, so first item is ZERO !?!
  if (ok < 0)
     { ConTx(refnum, "\nErr(", 5, 0);
       ConI (refnum, ok, 0);
       ConTx(refnum, ")\n", 2, 0);
     };

   +OPTIONAL
   ConAsc(refnum, "..end)\n", fDEBUG_SQK); //
   -OPTIONAL
  return ok; // value of < 0 signals failure.
}
```

8.3 A legacy function — Bypass

WE MUST GET RID OF Bypass AND REPLACE IT WITH DIRECT CALLS.

```
Int16 Bypass (UInt16 refnum, Int16 iCode, Int16 modifier, Char * aux, Int16 alen,
              Char * STACK, Char * STACKSTRING)
{ Int32 i;
```

```
  switch (iCode)
    {

      case iMARK:
        i = modifier;
        PushItem (STACK, STACKSTRING, (Char *) &i, 4, 'I', 0); // push mark depth!
        return SetMark(STACK);
        // this and ClearMark are used by SELECT processor to limit post-processin
        // of items on stack. Used with MAX, MIN, SORT, DISTINCT...

      case iUNMARK:
        return ClearMark(STACK, STACKSTRING);

      case iUNMARKONLY:
        return UnmarkOnly(STACK); // see code and usage

//      case iDEPTH:
//        return MarkDepth(STACK);

      case iMAX:
        if (modifier != 0) { return -SqErBypass; };
        return DoMax(STACK, STACKSTRING);
        // In our initial implementation of SELECT, we will only permit "SELECT ma
        // without group by. This is translated into "SELECT column" *with* a post
        // 'max' flag set. After values have been extracted, the max flag is exami
        // if set, MAX is invoked.
        // Note that this implies:
        //   a. iMARK stack *before* executing SELECT
        //   b. iUNMARKONLY stack at end, after all post-processing!
        // MIN is similar.

      case iMIN:
        if (modifier != 0) { return -SqErBypass; };
        return DoMin(STACK, STACKSTRING);
        // see notes above under "iMAX:"

      case iDISTINCT:
        // modifier is 'itemsinarow' number:
        i = modifier;  // convert to int32 (big endian) [nb fix for ARM]
        PushItem (STACK, STACKSTRING, (Char *) &i, 4, 'I', 0); // push integer val
        return DoDistinct(STACK, STACKSTRING);
        // DISTINCT execution in initial SELECT is similar to MAX above.
        // We only allow distinct on one column, at present.

/* /// removed: ////////////////////////////////////////////////////////////////
      case iSORT:
        i = modifier;  // convert to int32 (big endian) [nb ? fix for ARM]
          // note: must still agree on internal stack format for ARM. Might be
```

```
          // simply best to store integers as little endian as stack is NOT EXPORT
        PushItem (STACK, STACKSTRING, aux, alen, 'V', 0); // push aux deeper (firs
        PushItem (STACK, STACKSTRING, (Char *) &i, 4, 'I', 0); // push integer val
////        return DoSort(refnum, STACK, STACKSTRING);
        // Sort too is similar to MAX. At present we only sort on one item, but
        // we submit both the number of columns (for now must be 1) AND the
        // sort order, which is just the string "A" or "D" (default A, D is specif
        // if we say "SORT BY tablename.columname DESC").
        // Similar rules apply for mark and unmark.
*/ ////////////////////////////////////////////////////////////////////////////

      default:
      return -SqErBypass;

    };
}
```

9 The SELECT statement — preliminary routines

This routine and its subsidiaries (which come first) are cumbersome and can be fine-tuned. The documentation is turgid and needs a shake-up.

PackConditions stores the conditional part of the SELECT statement in a convenient shorthand notation. It in turn depends on a test for logical connectors (isVLC). We also translate from standard infix notation to a form of prefix notation using the StoreCondition routine.

First isVLC — is one of the following logical connectors present: “AND ” “OR ” or “NOT ”? The antecedent space has already been checked for.

```
Int16 isVLC (Char * sptr, Int16 slen)
{ if (slen < 3)
     { return 0;
     };
  if ( xSame(sptr, 3, "OR ",3))
     { return OPOR;
     };
  if (slen < 4)
     { return 0;
     };
  if ( xSame(sptr, 4, "NOT ",4))
     { if (  (slen > 7) // beware!
           &&(xSame(sptr, 8, "NOT NULL", 8))
          )
          { return 0; // no!
          };
       return OPNOT;
     };
  if ( xSame(sptr, 4, "AND ",4))
     { return OPAND;
     };
  return 0;
}
```

9.1 Preprocessing: condition storage

StoreCondition: This fx translates ”A condition B” into ”CA B” where C is the code for the condition, and a space character now separates A and B. We need solid coding of conditions: see *palmsql3A.h* in *CProgMain.tex*.

We examine “stringA condition stringB”. All data are written to the destination string. We:

1. encode and store the single character condition;

2. store string A
3. write a blank
4. write string B
5. return the total length of the string written to ‘dest’.

We look for a variety of operators including “ IS NULL”, “ IS NOT NULL”, “ > ”, “ < ”, “ = ”, “ >= ”, “ <= ”, “ <> ”;

A return value of 0 signals failure. First we find the space after the operand and before the operator (it must be there or we fail), and copy the operand into the destination buffer:

```
Int16 StoreCondition(UInt16 refnum, Char * dest, Int16 destmax,
                     Char * sqsrc, Int16 srclen)
{  Int16 bpos;
   Char bch;

   bpos = Advance(sqsrc, srclen, ' ');
   if (! bpos)
      { ConTx (refnum, "E:", 2, 0);
        ConTx (refnum, sqsrc, srclen, 0); // debug.
        SayErr (refnum, ErSQLleftSpace);
        return 0;
      };
   if (1+bpos > destmax)
      { SayErr(refnum, ErSqTinyBuffer);
        return 0;
      };
   xCopy (dest+1, sqsrc, bpos); // leave a byte for operator!
   sqsrc += bpos;       // also copy over the terminal space!
   destmax -= 1+bpos;
   srclen -= 1+bpos;    // clumsy
   bch = * (sqsrc);     // get next character
   sqsrc ++;
```

Next we examine the operator. First check for IS NULL and IS NOT NULL, placing the relevant character into the destination buffer (if apt):

```
   switch (bch)
     { case 'I':
        if (xCompare(sqsrc, "S NULL", 6) == 0)// if the same (!)
           { * dest = ISNULL;
             * (dest + bpos + 1) = SEPARATOR;
             return (bpos+2);
           };
```

```
 if (xCompare(sqsrc, "S NOT NULL", 10) == 0)// if the same
    { * dest = ISNOTNULL;
      * (dest + bpos + 1) = SEPARATOR;
      return (bpos+2);
    };
 SayErr(refnum, ErSqBadIcomparator);
 return 0;
```

Next for an equals sign:

```
case '=':
 bch = * (sqsrc);
 sqsrc ++;  // go past the space
 srclen --;
 if (bch != ' ')
    { SayErr(refnum, ErSqEqSpace);
      return 0; // fail
    };
 * dest = ISEQUAL; // might still check srclen?!
 break;
```

Now the whole gamut of less than, less than or equals, and so forth ...

```
case '<':
  bch = * (sqsrc);
  sqsrc ++;
  srclen --;
  if (bch == ' ')
     { * dest = ISLESS;
       break;  // !
     };
  if (bch == '>')
     { * dest = NOTEQUAL;
       sqsrc ++; // could check for ' ' too?
       srclen --;
       break;
     };
  if (bch == '=')
     { * dest = LESSEQUAL;
       sqsrc ++; // ...
       srclen --;
       break;
     };
SayErr(refnum, ErSqLtBad);
return 0;
```

Similarly for greater than:

```
      case '>':
        bch = * (sqsrc);
        sqsrc ++;
        srclen --;
        if (bch == ' ')
           { * dest = ISGREATER;
             break;
           };
        if (bch == '=')
           { * dest = GREATEREQUAL;
             sqsrc ++; // as above
             srclen --;
             break;
           };
        SayErr(refnum, ErSqGtBad);
        return 0;

     default:
        SayErr(refnum, ErSqBadOperator);
        return 0;
     };
```

If there is nothing after a comparator (we've already dealt with IS NULL and IS NOT NULL) we fail. If the destination buffer is too small, also fail.

```
  if (srclen < 1)  // if nothing after comparator
     { SayErr(refnum, ErSqNo2ndOp);
       return 0;
     };

  if (srclen > destmax)
     { SayErr(refnum, ErSqTinyBuffer);
       return 0;
     };
  dest += 1+bpos; // account for ' ' and comparator char.
  xCopy (dest, sqsrc, srclen);
  * (dest + srclen) = SEPARATOR;
  return (2 + srclen + bpos);
}
```

We return the length of the written string — the +2 accounts for the comparator and the separator at the end!

9.2 Preprocessing: packing

PackConditions takes the meaty WHERE clause of a SELECT or UPDATE statement and translates to our 'intermediate format'. We are really interested in AND OR NOT and parentheses. This is messy.

Things are made even more messy because we DON'T want to use recursion (stack considerations)

We establish the following rules:

1. the only valid logical connectors (we'll call these *VLC*s) are the strings " AND ", " OR ", and " NOT ". (Note the leading and trailing spaces. It's easy to preprocess constructions like ")AND(" into this format).

2. We *abandon* the usual convention that AND takes precedence over OR. (this can simply be re-introduced by preprocessing-in parentheses). Parsing is strictly from left to right *as if* AND, OR and NOT had equivalent value. So if we encounter A AND B OR C OR D, processing is as follows: stack up results of A B C D in order, then apply the logic OR,OR,AND to the results ie. (C OR D)→X, (X OR B)→Y, and finally Y AND A.

3. We introduce the idea of an OPERATOR, which is a VLC attached to a CONDITION like "col1 $>$ 5".

4. we FORBID unnecessary spaces, for example, those between parentheses, so "(" is invalid outside a quoted string, as are " AND " as well as "a = b", and "a OR (b AND a = 'xx')" which is invalid only because of the space after the quoted string 'xx'!

5. We disambiguate " NOT (A OR B)" and " NOT A OR B" by insisting that the latter be written as either "(NOT A) OR B" or as " NOT (A) OR B".

You can see that with this approach, we can process from left to right as follows:

1. create a logical stack on which to store ANDs and ORs;

2. move from left to right

3. (keep track of parenthesis depth if you wish)

4. store all text up to the first VLC, *excluding parentheses*

5. push that VLC to the stack

6. when you encounter a right parenthesis, pop and store a VLC as an OPERATOR together with the preceding text as yet unstored!

7. continue until finished

8. At the end, pop all the VLCs (one after the other) and store these as OPERATORs

For example, we are saying that “a AND b OR c” translates to RPN “a b c OR AND” and that “(a AND b) OR c” becomes the RPN-evaluated “a b AND c OR”. There is a tiny wrinkle to this translation of infix notation to RPN — because we will store each item A, B or C as a node, and associate zero or more logical operations with each node, we need to ’break up’ the RPN string and store the logical operations with the nodes. In the examples above we thus say:

“a AND b OR c” ==> “a b c OR AND” ==> a(S) b(S) c(OA)

where S means ’simply store this’, O means OR, and A means AND. Similarly:

”(a AND b) OR c” ==> ”a b AND c OR” ==> a(S) b(A) c(O)

We simply apportion any floating logic to the node to its immediate left. If there is no logical operator to apply, we simply say ’store the blighter’.

An implementation detail is that when we write our intermediate format string, we write the logical operators before the condition, as Sa rather than a(S), and similarly we write OAc rather than c(OA).

9.2.1 Optimisation

The above also gives a hint as to how we might optimise things just a little. There are two alternatives for each evaluation:

1. The logical value is FALSE. This implies that if we AND this value and the next value, then whatever the next value, the outcome is still false. So (logically) we can skip the next item provided the test specified there is AND. If there is no test at the next item, but the following item has ‘AA’ ie. two ANDs, we can skip the next two nodes, and so on.

2. The logical value is TRUE. This implies that if the logical test at the next item is OR, we can skip the next node. In a similar fashion to AND, if the next item has no test, but the *following* item has two ORs, then both can be skipped.

9.2.2 PackConditions

An important routine which packs from source to destination, formatting as it goes. It keeps count of parentheses, and can handle nested logic up to about 30 levels deep (not that we will ever require such convoluted logic)!

PackConditions returns the length of the string written, or zero on failure.

```
Int16 PackConditions (UInt16 refnum, Char * dest, Int16 destmax,
                      Char * sqsrc, Int16 srclen)
{
  Char * logstack;
  Int16 lg;
  Char * cbufP;
  Int16 co;
  Int16 par;
  Char * dP;      // destination ptr
  Char * sP;      // copy of source ptr
  Char c;         // current character
  Int16 vlc;      // logical code for AND or OR
  Boolean quoted;
```

We first create a stack on which we store ANDs and ORs. The first 32 bytes of ‘logstack’ are used to nest logic, the remainder (cbufP) to store a text specification of a condition. The nasty number MAXCBUF comes from the header file (256) and should be big enough for storing conditions temporarily. The variable ‘lg’ tracks the top of the logic stack, and ‘co’ the length of the item in the condition buffer.

```
  logstack = xNew2(refnum, 32+MAXCBUF, 0x102);
  lg=0;
  cbufP = logstack + 32;
  co=0;     // max is MAXCBUF
  par=0;    // parenthesis depth
  dP=dest;  // destination ptr
  sP=sqsrc; // copy of source ptr
  quoted = 0;
```

We work through the whole source buffer, character by character.

```
  while (srclen > 0)
    { c = * sP;               // get a character
      sP ++;
      srclen --;              // track
      * (cbufP+co) = c;       // store (by default)
      co ++;                  //  and bump offset
```

We examine for characters of special interest using a switch statement. The ‘quoted’ variable tells us if we are within a set of single quotes, in which case we are far less picky. A left parenthesis ups the parenthesis count.

```
      switch (c)
        { case '(':
            if (! quoted)
```

```
        { par ++; // bump parenthesis count
          co --;  // discard character
        };        // end 'if !quoted'
break;
```

A right parenthesis not only drops the parenthesis count but also forces us to write a single character (representative of an AND OR or NOT) from the logic stack to the destination buffer! We do this repeatedly if there are multiple right parentheses.

```
case ')':
  if (! quoted)
     { co --;     // discard character
       par --;    // dec parenthesis count
       sP --;     // sneaky
       srclen ++; // track
       while (  (* sP == ')') // many rpars?
              &&(lg > 0)
              &&(par > 0)
              &&(destmax > 0)
             )
         { par --; // pop stack for each ")"
           lg --;
             * dP = * (logstack+lg);// pop stack
            dP++;
            destmax --;
           sP ++;     // move to next character
           srclen --; // track
         };
       if ( (* sP) == ')' ) // too many?
          // signals failure ie par==0 or lg==0
          { Delete2(refnum, logstack);
            SayErr(refnum, ErSq2ManyRpar);
            return 0;
         };
```

Now that we've dispensed with the right parentheses, we format and store the text *from* the cbuf buffer into the destination using StoreCondition (Section 9.1).

```
       co = StoreCondition(refnum, dP, destmax, cbufP, co);
       if (! co)
          { Delete2(refnum, logstack);
            return 0; // fail
          };
       dP += co;
       destmax -= co;
       co = 0; // clear buffer
     }; // end "if ! quoted"
break;
```

If we encounter a single quote, we swop the sense of ‘quoted’.

```
case 0x27:  // ’ character
  quoted = ! quoted;
break;
```

The only way a space can occur outside a quoted string is if it’s associated with a logical connector *or* a test such as ‘=’, ‘>’ and so forth! We first check for the logical connectors AND, OR and NOT using isVLC (Section 9). Otherwise we simply store the space, in anticipation of later processing within StoreCondition, which picks up tests such as ‘=’.

```
case ’ ’:             //
  if (! quoted)
     { vlc = isVLC (sP, srclen);
       if (vlc)
          {
```

Okay, if there has been a recent right parenthesis, we may not have anything left to store, but otherwise we StoreCondition as above. We decrement ‘co’ (ignoring the space character we’ve just stored) and if nonzero, there must be something still to store!

```
co --;
if (co) // if anything to store
   { (* dP) = OPSTORE; // ’store’ operator
     dP ++;
     destmax --;
     co = StoreCondition(refnum, dP, destmax, cbufP, co);
     if (! co)
        { Delete2(refnum, logstack);
          return 0; // fail
        };
     dP += co;
     destmax -= co;
   };
```

In any case, we move to *after* the logical condition string (e.g. “ AND ”) and push the value in ‘vlc’ to the logic stack. We fail if the stack overflows.

```
co = Advance (sP, srclen, ’ ’);
sP += co;
srclen -= co;
co = 0; // buffer MUST BE clear
* (logstack+lg) = vlc; // push vlc
lg ++;
```

```
                    if (lg >= 32)
                       { Delete2(refnum, logstack);
                         SayErr(refnum, ErLogicTooDeep);
                         return 0;
                       };
```

Here's a final hack: if the following command is also a logical command (as in " AND NOT ") then we move back in the source so we can see the leading space!

```
                    if (isVLC(sP, srclen))
                       { sP --;
                         srclen ++;
                       };
                  };  // end 'if vlc'
             };  // end 'if quoted'
        break;
```

Here are the terminating curly braces of the switch and while statements. We also need to check that the temporary buffer hasn't overflowed:

```
         //  default: // do nothing more
       };   // end switch

     if (co > MAXCBUF)
        { Delete2(refnum, logstack);
          SayErr(refnum, ErSqlWhile2Long);
          return 0; // fail
        };
   };       // end while
```

At the end of everything, we must also check that the parentheses and quotes balanced:

```
  if (par > 0)
     { Delete2(refnum, logstack);
       SayErr(refnum, ErSqImbalParens);
       return 0;
     };
  if (quoted)  // imbalanced 'quotes!
     { Delete2(refnum, logstack);
       SayErr(refnum, ErSqImbalQuotes);
       return 0;
     };
```

We should be done APART from any final text and an associated sequence of operators on the stack:

```
  // (7) write remaining text with sequential pop of stack down to nil.
  if (!co)                        // if nothing more
     { if (lg)                    // but still a stack of VLCs
          { Delete2(refnum, logstack);
            SayErr(refnum, ErSqBadLogic);
            return 0;
          };
     } else                       // more to write..
     {  if (lg)                   // and some stack logic.
           { while ((lg > 0) && (destmax > 0))
             // POP REMAINING VLCs
               { lg --;
                 * dP = * (logstack+lg);
                 dP++;
                 destmax --;
               };
           } else
           { * dP = OPSTORE; // if no logical operators, just store!
             dP ++;
             destmax --;
           };
```

We still have to store the condition ...

```
        co = StoreCondition(refnum, dP, destmax, cbufP, co);
        if (co)
           { dP += co;
             destmax -= co;
           } else
           { Delete2(refnum, logstack);
             return 0; // fail
           };
     }; // end: "if co"

  Delete2(refnum, logstack);
  return (Int16) (dP - dest);
}
```

We return the length of the string written (or zero on failure).

9.3 Translate to intermediate format — SQxlate

Translate from standard SQL SELECT to intermediate format:

1. We look for pre-determined FROM and WHERE text and delimit clauses using <SEPARATOR>s.

2. Simply process <conditionals> section. We permit: AND, OR, AND NOT, OR NOT, = > < <= >= <>, IS NULL, IS NOT NULL

There is a preliminary function we need to get under our belt: CfCopy.

9.3.1 CfCopy

This is a Small local optimisation function: We submit a destination string, source string, and object string (itm), verify that source string starts with object string, then copy to the destination everything in the source string *after* the characters we just verified, *up to* (but not including) the next space in the source string.

If the 'next space' is not found, then we simply copy over the whole of the rest of the string.

CfCopy returns the length of the string copied, 0 on failure.

```
Int16 CfCopy ( Char * dest, Int16 destmax, Char * sqsrc, Int16 srclen,
                            Char * itm, Int16 itmlen)
{ Int16 s;

  if (itmlen > srclen)
     { return 0; // sanity check
     };
  if (xCompare(sqsrc, itm, itmlen) != 0)// NOT the same
     { return 0;                        // fail
     };
  sqsrc += itmlen;                  // go PAST ' ' at end of item!
  srclen -= itmlen;                 // caution

  s = Advance (sqsrc, srclen, ' '); // end of table names
  if (s < 1)     // not found
     { s=srclen; // whole string
     } else
     { if (s > destmax)
          { return 0; // fail, buffer too small
          };
       s--; // do NOT copy the blank
     };
  xCopy (dest, sqsrc, s);
  dest += s;
  * dest = SEPARATOR;
  s++; // bump past separator
  return s;
}
```

Now for the actual translation.

9.3.2 SQxlate

This routine accepts:

1. A library reference number;
2. A destination buffer with its maximum length;
3. A source string with its actual length;
4. A by-reference argument to which we write the number of columns identified.

SQxlate returns the number of characters written to ‘sqsrc’, or zero if an error occurred.

```
Int16 SQxlate (UInt16 refnum, Char * dest, Int16 destmax,
                Char * sqsrc, Int16 srclen,
                Int16 * retcols) // by reference.
{ Int16 s;
  Int16 xlen;
  Char * keepstart = dest;  // ugly
```

9.3.3 Copy selection columns

The column names are separated by commas, and *no spaces are permitted anywhere* apart from the terminal space before the FROM clause.

```
  s = Advance (sqsrc, srclen, ' '); // up to" FROM "
  xCopy (dest, sqsrc, s);  // copy all including space
  *retcols = 1+CountItems(dest, s, ',');

  dest += s;
  * (dest-1) = SEPARATOR;  // replace ' ' with separator
  destmax -= s;
  sqsrc += s;
  srclen -= s;
```

CountItems gives us the number of columns to be selected, returned by-reference, as noted above. In the intermediate string we replace the terminal space with the SEPARATOR character.

9.3.4 Process FROM table-names

The list of tables is similar to the list of columns, in that no spaces are permitted between or within table names, and the names are separated by commas.

```
  s = CfCopy(dest, destmax, sqsrc, srclen, "FROM ", 5);
  if (!s) // failed
     { SayErr(refnum, ErNoFrom);
       return 0; // fail.
     };
  dest += s;    // past sSEPARATOR character
  destmax -= s; // dest also now contains table name!
  sqsrc += s+5; // past the blank
  srclen -= s+5;

  if (srclen < 1) // no WHERE clause
     { *dest = 'T';   // signal "always return true" [24-3-2005]
       return 1+(dest-keepstart); // done.
     };
```

By the above the following statement is actually incorrect:
`"SELECT x FROM a "`
. . . because a terminal blank will be disallowed if there is no WHERE!

9.3.5 Process WHERE

We identify the WHERE statement (or fail) and then process this statement, which has the potential to be very complex indeed. PackConditions (Section 9.2) does the work.

```
  if ( xCompare(sqsrc, "WHERE ", 6) != 0 )
     { SayErr(refnum, ErNoWhere);
       return 0; // fail
     };
  sqsrc += 6; // past WHERE
  srclen -= 6;
  xlen = PackConditions(refnum, dest, destmax, sqsrc, srclen);
  if (! xlen)
     { SayErr(refnum, ErSqlPackFail);
       return 0;
     };
  dest += xlen;
  return (dest - keepstart); // ugly.
}
```

10 SQL temporary structures — linked lists

10.1 A minor indexing function

This routine accepts the name of a table, and ensures that an index is present. If index creation fails, zero is returned; if an index is present (or was created) then 1 is returned. A negative value is returned if submitted parameters are rubbish.

```
Int16 FixIndex (UInt16 refnum, Char * tablename, Int16 tbllen)
{
  UInt16 IDXLIB;
  Int16 ok;

  if (! tablename || (tbllen < 1) )
     { return -1; // fail
     };

  IDXLIB = GetIndexRef(refnum);
       // will be zero if indexing disabled.
  if ( IDXLIB &&
       ! ISINDEX( IDXLIB, tablename, tbllen)
     )
     {ok = MAKEINDEX (IDXLIB, tablename, tbllen, 0);
       if ( ok < 1)
          {  TurnOffIndexing (refnum); // disable indexing !
             ConAsc(refnum, "[? idx0 ", 0); // err msg
             ConTx (refnum, tablename, tbllen, 0);
             ConI  (refnum, ok, 0);
             ConAsc(refnum, "]", 0);
             return 0; // failed.
          } else
          { // here create index on primary key
            ok = INDEXCOLUMN (IDXLIB, tablename, tbllen,
                        1, 0, 4, MAXACTUALROWS*4);
            if ( ok < 0 ) // might check for >0 == truncations!
                 { TurnOffIndexing (refnum);
                   ConAsc(refnum, "[? idx1 ", 0); //
                   ConTx (refnum, tablename, tbllen, 0);
                   ConI  (refnum, ok, 0);
                   ConAsc(refnum, "]", 0);
                   // simple error message.
                   return 0;
                 }; // if fails disable indexing!
          };
     };
  return 1; // success
}
```

MAXACTUALROWS is a bit of a hack, specified earlier in the code.

10.2 Find, Open a database

UInt16 cardNo is always zero, in our current schema.

```
LocalID s_DmFindDatabase (Char* nameP)
{ return( DmFindDatabase(MAINCARD, nameP)); // does database exist?
}

DmOpenRef s_DmOpenDatabase (UInt16 refnum, LocalID dbID, UInt16 mode)
{ DmOpenRef dor;
  Err e;
  dor = DmOpenDatabase (MAINCARD, dbID, mode);
  if (dor) { return dor;
           }; // success
  e = DmGetLastErr(); // get error code
  ConTx(refnum, "\n?{", 3, 0);
  ConI (refnum, (Int16) e, 0);
  ConTx(refnum, "}", 1, 0);
  return 0;
}
```

10.2.1 CreateDbLink: Create a data table link

We insert table characteristics into a dbLINK structure. This allows us to access the table easily, including our custom header in record zero.

Early on we create a new dbLINK structure. The dp->current = 0 line fixed a major headache caused by arbitrary numbers being present in this field!

The last parameter of CreateDbLink is interesting. If it's nonzero, then before we create the dbLINK structure we ensure that an index file exists for the database table. If index creation fails, then we disable indexing by resetting the 'global' *CANINDEX* flag to zero, but will still try to create the dbLINK!

Also of interest is *caching*. With very populous OBS and PROCESS tables, things grind to a halt. In an attempt to fix this, I've introduced the creation of little cached files which replace OBS and PROCESS. These are tricky for several reasons:

1. They are only of use (and *must* be removed) when we are looking at a particular patient;

2. We sneak in the p-file (the usual file with"p-" as a suffix) in the call to CreateDbLink, but there's a problem:

3. CreateDbLink not only opens the substitute file (which is what we want), it also records the name, (including the ‘p-’ suffix) which is unfortunate as we only want the ‘real’ name.

We’ve created a small hack to fix this. The following assumes that dstp has space for an ASCIIZ terminator, FWIW (Obsolete).

```
Int16 HackCopy (Char * dstp, Char * srcp, Int16 slen)
{
  if (  (slen < 3)
      ||(*srcp != 'p')
      ||(*(srcp+1) != '-')
     ) // if a normal copy:
     { xCopy (dstp, srcp, slen);
       *(dstp+slen) = 0x0; // ASCIIZ
       return slen;
     };
  // now we know length is 3 or more, AND
  // string starts with "p-":
  // WE ONLY COPY THE REST!
  xCopy (dstp, srcp+2, slen-2);
   *(dstp+slen-2) = 0x0; // ASCIIZ
  return (slen-2);
}
```

In the following I now therefore open the p-file, *but* store the file name without the prefix. This allows table matches etc to work as normal. We enforce the requirement that a call to CreateDbLink must have ASCname as an ASCIIZ string, despite the namelen parameter. Nasty.

```
struct dbLINK * CreateDbLink (UInt16 refnum, Char * ASCname,
                              Int16 namelen, Int16 checkindex)
{
  LocalID lid;  // foul constraint imposed by PalmOS
  DmOpenRef dbase;
  MemHandle headhand;
  Char * headp;
  Char * name;
  struct dbLINK * dp;

// ConTx (refnum, "\nOpen{", 6, 0);  // DEBUGGING 10/3/2008
//       ConTx (refnum, ASCname, namelen, 0);
//       ConTx (refnum, "}", 1, 0); // END DEBUGGING
```

In the following function call (to FixIndex) we implement indexing on the primary key for the stated database table. We invoke the separate IDXLIB indexing

library, unless this has been disabled (GetIndexRef returns a zero value). If the index already exists (ISINDEX returns a nonzero value) well and good, otherwise we try to create the index, and then index column 1, the primary key.

```
// first, check that index file exists:
if (checkindex)
   { FixIndex(refnum, ASCname, namelen);
   };
```

Now to our main business . . .

```
// next, create the link
if ((! ASCname) || (namelen < 1))
   { return 0; // fail
   };

dp = (struct dbLINK *) xNew2 (refnum, sizeof(struct dbLINK), 0x103);
     // could fail (err) ?
dp->current = 0;
name = xNew2(refnum, namelen+1, 0x104);

namelen = HackCopy(name, ASCname, namelen);
// see note above: exclude a prefix of "p-" from name!
// xCopy (name, ASCname, namelen);
// *(name+namelen) = 0x0; // ASCIIZ [ugh]
dp->name = name;
dp->nlen = namelen;

lid = s_DmFindDatabase (ASCname); // ASCname IS ASCIIZ!
// lid = s_DmFindDatabase (name);

if (! lid)
   { ConTx (refnum, "\nE-l-id{", 8, 0);
     ConTx (refnum, ASCname, namelen, 0);
     ConTx (refnum, "}", 1, 0); // bad name
     SayErr(refnum,ErDbNotFound);
     Delete2(refnum, name);
     Delete2(refnum, (Char *)dp);
     return 0;
   };
dbase = s_DmOpenDatabase (refnum, lid, dmModeReadWrite);
        // what about dmModeReadOnly ???
if (! dbase)
   { ConTx (refnum, "\nE-l-db{", 8, 0);
     ConTx (refnum, ASCname, namelen, 0);
     ConTx (refnum, "}", 1, 0); // bad name
     SayErr(refnum,ErCannotOpenDb);
     Delete2(refnum, name);
```

```
       Delete2(refnum, (Char *)dp);
       return 0; //fail
     };
```

A few more frills …

```
  dp->odb = dbase;   // can be null
  headhand = s_DmQueryRecord(dbase, 0); // get header (read only)
  headp = (Char *) s_MemHandleLock(headhand);
  dp->hand = headhand; // keep so we can release this (eugh)!
  dp->head = headp;    // remember header!
  dp->next = 0;                                  // clear
  return dp;
}
```

10.3 Deleting a linked list

Straightforward.

```
Int16 KillDbList (UInt16 refnum, struct dbLINK * dblist)
{ struct dbLINK * dp;
  Char * name;
  DmOpenRef mydb;
  MemHandle headhand;

  while (dblist)
    { dp = dblist->next;
      mydb = dblist->odb;
      headhand = dblist->hand; // eugh.
      if (headhand)
         { s_MemHandleUnlock(headhand);  // ? check for error
         };
      if (mydb)
         { PalmFileClose(mydb);
         };
      name = dblist->name;
      if (name)
         { Delete2 (refnum, (MemPtr) name);  // ??? if fails
         };
      Delete2 (refnum, (MemPtr) dblist);  // could die if fail ?
      dblist = dp;
    };
  return 1; // ok
}
```

10.4 Create a linked list of tables

CreateTableList: This function is only called by SeekMany, as an important part of a SELECT statement.

Given a comma-delimited list of names (no spaces), identify databases (PalmOS db files) and create a linked list of open databases, each of which corresponds to a data table. As things stand, the list of databases must be in our canonical form, sticking to our leaf-branch rule where databases to the left cannot be referenced as foreign keys by databases on the right.

Note: there is a terminal comma on the list of names. We return zero (null) on failure. A recent addition is that we create an index for each table referred to, unless one already exists.

Even newer is caching. The idea is that in a local context (looking at a particular patient) where we are performing multiple SELECTs, things become very turgid. So we create our own little tables (p-PROCESS, p-OBS) to speed things up. Instead of referring to the original tables, *if in the right context*, CreateTableList will instead refer to the little tables! We accomplish this through a call to the Cache library (CACHELIB) call: CACHEFINDTABLE. Here's the subsidiary function which always assumes 3 spare bytes at the top of the tname buffer.

```
Int16 CheckCache (UInt16 refnum, Char * tname, Int16 tlen)
{
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  if (! gl->CANCACHE)
     { return tlen;  // default
     };
  return CACHEFINDTABLE(gl->CACHELIB, tname, tlen);
  // returns altered length and name, where appropriate, otherwise tlen.
}
```

Note that CreateDbLink, which creates a structure used to access the database, is also called by the SQL UPDATE command. At present we do *not* CheckCache for that usage.

Here's the main function:

```
struct dbLINK * CreateTableList (UInt16 refnum, Char * commalist,
                                 Int16 commalen)
{ struct dbLINK * root = 0;
  struct dbLINK * dbp=0;  // aagh keep compiler happy
```

```
  struct dbLINK * dbnew;

  Int16 commapos;
  Char * tname;
  Int16 tlen;

  commapos = Advance (commalist, commalen, ',');
```

We split up the list based on the commas.

```
  while (commapos)
    { tname = xNew(commapos+2); // length+3
      tlen = commapos-1;
      xCopy (tname, commalist, tlen);
      *(tname+tlen) = 0x0; // asciiz.
      tlen = CheckCache(refnum, tname, tlen);
      dbnew = CreateDbLink(refnum, tname, tlen, 0); // NO index check, for now
      Delete (tname);

      if (! dbnew)
         { SayErr(refnum,ErDbNotMakeLink);
           KillDbList(refnum, root);  // clean up and fail.
           return 0;
         };
      // now dbLINK *does* exist.
      // first, check that index exists on ... ??? [noooo!]

      if (! root)  // despite compiler warning
         { root = dbnew; //  on first pass, this is taken
         } else          //  not
         { dbp->next = dbnew; // this, so dbp *is* initialised
         };                   //   before it is used
      dbp = dbnew;            //   ie intialised here.
      commalist += commapos;
      commalen -= commapos;
      commapos = Advance (commalist, commalen, ',');
    };
  return (root);
}
```

10.5 Make a query node

A trivial routine to make our cQUERY structure.

```
struct cQUERY * MakeQueryNode (UInt16 refnum)
{ struct cQUERY * ncq;
  Int16 lgth = sizeof(struct cQUERY);
```

```
  ncq = (struct cQUERY *) xNew2(refnum, lgth, 0x105);
  if (! ncq) { return 0; };  // fail
  xFill ((Char *) ncq, lgth, 0x0);// clear whole struct
  return ncq;
} // quick & dirty.
```

10.6 Find a column

Given dotted TABLENAME.COLUMNAME, identify table and column (by name) and return a cQUERY structure describing the column in detail. BOTH table AND column names must be submitted!

We submit a dbLINK structure to allow us to find the tables by name, and the dotted name/length in the qname/qlen pair.

```
struct cQUERY * FindColumn (UInt16 refnum, struct dbLINK * dblist,
                            Char * qname, Int16 qlen)
{
  struct dbLINK * clink;
  struct cQUERY * cQ;
  Int16 dotpos;
  Int16 notfound;
  Int16 colcount = 0;
  Int16 cols;
  Int16 coloff;
  Char * colname;
  Int16 cnamelen;
  Char * hptr;
  Int16 descriptor;
  clink = dblist;

  dotpos = Advance (qname, qlen, '.');
```

First we advance to after the table name (all are dotted). We then move through the database list looking for that name.

```
  if (dotpos > 1)
     { dotpos --;   // don't count dot
       notfound = 1;
       while (notfound)
         { if (! clink)
              { SayErr(refnum,ErDbNoColumnLink);
                return 0; // fail
              };
           if (  (clink->nlen == dotpos)
```

```
            &&(xSame(qname, dotpos, clink->name, dotpos))
           )  { notfound = 0;
              } else
              { clink = clink->next;
              };
      };
   } else
```

If there is no dot, we assume that the table referred to is the first one (!) By putting -1 into dotpos, we later force it to zero, taking the whole name:

```
   { dotpos = -1; // will become 0 below, so whole name taken!
   };
```

Here goes — examine the table for the presence of the column:

```
dotpos ++;
qname += dotpos;  // move to column name
qlen -= dotpos;
hptr = (clink->head);
cols = ReadInt16X(hptr+0xE); // get number of columns
```

Within each column descriptor, the offset of the column name is at zero, the length at +2.

```
while (colcount < cols) // index of 1st column is zero
   { descriptor = ReadInt16X( hptr + 0x10 + 2*colcount);
     coloff     = ReadInt16X( hptr + descriptor + 0);
                  // should be 0x10 !
     cnamelen   = ReadInt16X( hptr + descriptor + 2);
     colname    = hptr + descriptor + coloff;
     if (coloff != 0x10)
        { SayErr(refnum,ErDbBadColumnOffset);
          return 0;
        };
```

We test for the name (if the qlen condition fails, then the xSame won't be invoked). On success, we create a new node, with null default values for all fields. We also link the node to the database table. At present we do not check on the dependency, so a really bad line of code might do all sorts of mischief.

```
     if (  (qlen == cnamelen)
         && xSame(colname, cnamelen, qname, cnamelen)
        )
        { cQ = MakeQueryNode(refnum);
          cQ->col = colcount;
```

```
            cQ->len = ReadInt16X( hptr + descriptor + 4);
            cQ->scale = (Int16) (* (hptr + descriptor + 7));
            cQ->type = * (hptr + descriptor + 6);
            cQ->dtable = clink; // link to database table
            if (cQ->type == 'I')
               { // here should perhaps check on dependency table:
               };
            return (cQ);
          };
       colcount ++;
     };

  SayErr(refnum,ErFailLocateColumn);
   ConAsc(refnum, "[", fDEBUG_ALWAYS); //
   ConTx(refnum, qname, qlen, fDEBUG_ALWAYS); //
   ConAsc(refnum, "]", fDEBUG_ALWAYS); //
 return 0;
}
```

We signal failure to locate a correct column, and write the offending column name to the console.

10.7 Make a list of result columns

FindResultColumns uses FindColumn to make a linked list of result columns. This is fairly self-explanatory. To make things easier there must be a comma at the end of the list of column names.

```
struct cQUERY * FindResultColumns (UInt16 refnum,
         struct dbLINK * dblist, Char * qname, Int16 qlen)
{ struct cQUERY * rsltnode=0;
  struct cQUERY * nxtnode=0; // aagh keep compiler happy
  struct cQUERY * newnode;
  Int16 commapos;
  commapos = Advance (qname, qlen, ',');

  while (commapos)
    { newnode = FindColumn (refnum, dblist, qname, commapos-1);
      if (! newnode)
         { SayErr(refnum,ErFailMakeColumnNode); // ? need
           return 0; // fail
         };
      if (! rsltnode)              // CLUMSY:
         { rsltnode = newnode;     // retain 1st node
         } else
         { nxtnode->next = newnode;//
         };
```

```
      nxtnode = newnode;
      qname += commapos;
      qlen -= commapos;
      commapos = Advance (qname, qlen, ',');
    };
  return (rsltnode);
}
```

Straightforward, simply moving from comma to comma. No intervening spaces are allowed.

11 SELECT: Encoding of data

11.1 Primitive routines

11.1.1 Float encoding: textEncodeFloat

Assumes 8 byte destp (at least) THERE IS NO FORMAT CHECKING OF FLOATING POINT NUMBER. BUGGER. (Crippled PalmOS routines).

```
Int16 textEncodeFloat (UInt16 refnum, Char * destp, Char * srcp, Int16 srclen)
{
  Char * dbuf;
  dbuf = xNew2(refnum, srclen+1, 0x106);                          /* hideous asciiz
  xCopy(dbuf,srcp,srclen);
  * (dbuf+srclen) = 0x0;
  FlpBufferAToF( (FlpDouble *)destp, dbuf);      /* returns null  */
  Delete2(refnum, dbuf);
  return 1;
}
```

We must introduce better routines at some stage.

11.1.2 textEncodeDate

Rudimentary. Assumes that destination is a simple string, not a PalmOS buffer, so don't need s_DmWrite. [FIX ME! ALSO NEED TO VALIDATE THE bloody DATE] The following was stolen from utility.hpp: FormatDate.

```
Int16 textEncodeDate(Char * destp, Char * srcp, Int16 srclen)
{ if (srclen != 10) // clumsy
     { return 0;
     };
  xCopy (destp, srcp, 4); // for NOW just assume YYYY and copy
  destp += 4;
  srcp += 4;
  if ( (* srcp) != '-')
     { return 0;          // fail if bad separator
     };
  srcp ++;
  xCopy (destp, srcp, 2); // assume MM (not just M), copy
  destp += 2;
  srcp += 2;
  if ( (* srcp) != '-')
     { return 0;          // fail
     };
  srcp ++;
```

```
  xCopy (destp, srcp, 2); // assume DD
return 1;                 // success
}
```

11.1.3 textEncodeTime

Similarly rudimentary. See textEncodeDate. Also stolen from utility.hpp: FormatTime.

```
Int16 textEncodeTime(Char * destp, Char * srcp, Int16 srclen)
{
  xCopy (destp, srcp, 2); // ASSUME HH
  destp += 2;
  srcp += 2;
  if ( (* srcp) != ':')
     { return 0;          // fail if bad separator
     };
  srcp ++;
  xCopy (destp, srcp, 2); // assume MM
  destp += 2;
  srcp += 2;
  if ( (* srcp) != ':')
     { return 0;          // fail
     };
  srcp ++;
  xCopy (destp, srcp, 2); // assume SS
  destp += 2;
  srcp += 2;

/* // for now, ignore stuff after any decimal
  xCopy (destp, "000000", 6); // default all zeroes! ie HHMMSS000000
  if (srclen > 8)             // if decimal .ffffff
     { if (srclen > 15)
          { srclen = 15;      // truncate!
            // might generate minor warning
          };
       if ( (* srcp) != '.')
          { return 0;
          };
       srcp ++;     // past "."
       srclen -= 9; // get number of characters
       xCopy (destp, srcp, srclen);
         // is zero length ie "HHMMSS." ok? (yes)
     };
*/ // end of ignore decimal.

 return 1; // ok
```

```
}
```

textEncodeTimestamp: likewise. Similar to textEncodeDate, textEncodeTime. originally FormatTimestamp from utility.hpp.

```
Int16 textEncodeTimestamp(Char * destp, Char * srcp, Int16 srclen)
{ if (! textEncodeDate (destp, srcp, 10))
      { return 0;
      };
  destp += 8;
  srcp += 10;  // go past date
  if ( (* srcp) != ' ' )
     { return 0;
     };
  srcp ++;
  return textEncodeTime (destp, srcp, srclen-11);
}
```

11.2 Main encoding routine

11.2.1 EncodeAll

Encodeall is *only* called by XFormat.

Format data, given an ASCII string. Given a target string, datum and its type, convert from raw ascii datum to encoded data and store in target. NULL is also acceptable. We return the *LENGTH OF THE ENCODED STRING*, which may be zero (for null). NB. the XFormat function which preceded this returned length +1. Return of a *negative number* signals an error.

Issues:

1. What about scale?
2. THE ENCODE DESTINATION MUST BE A CHAR * AND NOT WITHIN A PALMOS RECORD.
3. In original XFormat, we also made sure that a submitted length was not exceeded. We DO NOT do this here, relying on the caller to make any such checks.

```
Int16 EncodeAll (UInt16 refnum, Char * destin, Int16 destsize,
                 Char * datum, Int16 datlen,
                 Char typeofdatum, Int16 scale)
{ Char * P;
  Char c;
```

```
  Int32 i;
  Int16 dot;

  if (! datlen)
      { SayErr (refnum, SqErEncodeNil);
        return -1;
      };
  if (  (datlen == 4)
      &&(xSame2 (datum, "NULL", 4))
     )
     { return 0; // null length
     };

  switch(typeofdatum) // depending on datum type
    {
      case 'V':    // VARCHAR
      if (  (*datum != 0x27)
          ||(*(datum+datlen-1) != 0x27)
         )
      { SayErr (refnum, SqErUnquoted);
        return -1;
      };
      datlen -= 2;  // for quotes
      datum ++;     // past 1st quote
      /* We next go through whole string looking for occurrences of '' and
         replacing each one with a single quote (')
      */
      P = destin;
      while (  (datlen > 0)
             &&(destsize > 0)
            )
        { c = (*destin++ = *datum++);
          if (c == 0x27)
             { if (*datum == 0x27)// ok IFF next character also a 'quote
                  { datum ++;     // skip this
                    datlen --;    // track
                  } else
                  { SayErr (refnum, SqErOneQuote);
                    return -1;
                  };
             };
          datlen --;
          destsize --;
        };
      if (destsize == 0)
         { SayErr (refnum, SqErTruncated);
           return -1;
         };
      return ((Int16) (destin-P));  // return length of chars written
```

```
/*................................................*/
case 'I':   // INTEGER
  if (destsize < 4)
     { SayErr (refnum, SqErIntegerSpace);
       return -1;
     };
  i = Ascii2I32(datum,datlen);
  if ((i < 0) || (i > 999999999))
     { SayErr (refnum, SqErIntegerBad);
       return -1;
     };
  xCopy(destin, (Char *)&i, 4); // ARM processor: must reverse i
  return 4; // length is always 4

/*................................................*/
case 'N':   // NUMERIC
/* We want to:
   1. clip leading zeroes including those after the decimal point.
   2. truncate (!) any digits after the scale has run out.
   3. have at least 1 digit (it may be a zero?!)
*/

if (scale < 0)
   { SayErr (refnum, SqErNumericScale);
     return -1;
   };
scale ++;   // artificial bump: see usage below
P = destin; // keep copy
dot = 0;    // places after decimal point

// here must remove leading zeroes
while ((* datum == '0') && (datlen > 0))
      { datlen --;
        datum ++;
      };
if (! datlen) // if only zeroes
   { * destin = '0';    // put single zero
     return 1;          // length is just 1.
   };

while (  (datlen > 0)
       &&(destsize > 0)
       &&(scale > 0)
       )
  { c = * datum;
    datum ++; // clumsy
    datlen --;

    * destin = c;
```

```
      destin ++;
      destsize --;

      if (c > '9')
         { SayErr (refnum, SqErBadNumeric1);
           return -1;
         };
      if (c < '0')
         { if (c != '.')
              { SayErr (refnum, SqErBadNumeric2);
                return -1;
              };
           if (dot)                      // if dot nonzero, second '.'
              { SayErr (refnum, SqErBadNumeric3);
                return -1;
              };
           destin --;   // ignore transferred dot
           destsize ++; // track
           dot ++;      // set dot to 1
         };
      if (dot)          // if processing after decimal
         { scale --;
         };
    };
  scale --;
  if (! destsize)       // if no more space //
     { if (  (! datlen) // if no more data
           &&(! scale)  // AND no more significant 0's after .
          )
          { return ((Int16) (destin-P)); // return length of chars written
          };                             // ie. was acceptable
       SayErr (refnum, SqErBadNumeric4);
       return -1;
     };
  while (scale > 0)
    { *destin = '0';           // write zeroes AFTER decimal!
      destin ++;
      scale --;
    };
  return ((Int16) (destin-P)); // return length of chars written

  /*.....................................................*/
  case 'D':
    if (destsize < 8)
       { SayErr (refnum, SqErDateSpace);
         return -1;
       };
    if (! xSame2 (datum, "DATE '", 6))
       { SayErr (refnum, SqErDate);
```

```
        return -1;
      };
    if ( *(datum+datlen-1) != 0x27)
      { SayErr (refnum, SqErDateTerminal);
        return -1;
      };
    if (! textEncodeDate(destin, datum+6, datlen-7))
      { SayErr (refnum, SqErDateFormat);
        return -1;
      };
    // Do NOT need destination length, as know it's >= 8,
    //  and we only need 8 chars
    return 8; // length is always 8

  /*.....................................................*/
  case 'T':
    if (destsize < 6)  // [okay 12 best]
      { SayErr (refnum, SqErTimeSpace);
        return -1;
      };
    if (! xSame2 (datum, "TIME '", 6))
      { SayErr (refnum, SqErTime);
        return -1;
      };
    if ( *(datum+datlen-1) != 0x27) // no terminal quote
      { SayErr (refnum, SqErTimeTerminal);
        return -1;
      };
    if (! textEncodeTime(destin, datum+6, datlen-7))
      { SayErr (refnum, SqErTimeFormat);
        return -1;
      };
    return 6; // [NOT return 12; for now NO decimals] ???

  /*.....................................................*/
  case 'S':
    if (destsize < 14)  // okay, 20 ideal ??? [watch out]
      { SayErr (refnum, SqErStampSpace);
        return -1;
      };
    if (! xSame2 (datum, "TIMESTAMP '", 11))
      { SayErr (refnum, SqErStamp);
        return -1;
      };
    if ( *(datum+datlen-1) != 0x27) // no terminal quote
      { SayErr (refnum, SqErStampTerminal);
        return -1;
      };
    if (! textEncodeTimestamp(destin, datum+11, datlen-12))
```

```
            { SayErr (refnum, SqErStampFormat);
              return -1;
            };
          return 14; // [ NOT: return 20; for now no decimals]

       case 'F':
          if (destsize < 8)
            { SayErr (refnum, SqErFloatSpace);
              return -1;
            };
          if (! textEncodeFloat(refnum, destin, datum, datlen))
            { SayErr (refnum, SqErFloatFormat);
              return -1;
            };
          return 8;

       // default:
      };
  SayErr (refnum, SqErEncode);
  return -1;
}
```

11.3 Formatting

The following XFormat routine was multiply represented in both this library and in the main program described in *CprogMain.tex*. It's now been removed from the latter.

Invocation of XFormat is cumbersome to say the least. It is called indirectly by the legacy function qFormatDatum, and directly by SQL3UPDATE, and TransferFormatDatum (which is called by SQLins).

qFormatDatum is only called by MakeQueryItem.

```
Int16 XFormat(UInt16 refnum, Char * destin, Int16 destsize,
              Char * datum, Int16 datlen, Char coltype,
              Int16 colscale, Int16 maxcolsize)
{
  Int16 ok;

  ok = EncodeAll (refnum, destin, destsize, datum, datlen, coltype, colscale);
  if (ok < 0)
     { return 0;
     };
  if (ok > maxcolsize)
     { SayErr(refnum,ErTruncateMaxColSize);
       ok = maxcolsize;
     };
```

```
  return (1+ok);  // THIS FUNCTION RETURNS +1 !
}
```

11.4 Format one datum

```
Int16 qFormatDatum (UInt16 refnum, Char * destp, Int16 dlen,
         struct cQUERY * cp, Char * srcp, Int16 srclen)
{  // cumbersome call to XFormat. A legacy.
Int16 ok;
ok = XFormat(refnum, destp, dlen, srcp, srclen,
             (Char) cp->type, cp->scale, cp->len);
return (ok-1);  // XFormat returns length + 1 ?!
}
```

12 Query lists: creation and deletion

12.1 Delete query list

'KillQuery': kill a whole linked list of cQUERY structures. Do NOT need to kill the dbLINKs, as these are in separate linked list but must delete opstring. GutMe 'removes the guts' of a cQUERY node prior to its deletion.

```
Int16 GutMe(UInt16 refnum, struct cQUERY * cp)
{
  if (cp->opstring)
     { if (! Delete2(refnum, cp->opstring))
          { return 0;
          };
     };
  if (cp->cnstant)
     { if (! Delete2(refnum, cp->cnstant))
          { return 0;
          };
     };
  return (Delete2 (refnum, (Char *) cp));
};
```

Here's the actual KillQuery routine:

```
Int16 KillQuery (UInt16 refnum, struct cQUERY * cp)
            // cp is allowed to be 0 (null)
{ struct cQUERY * nxt;
  while (cp)
    { nxt = cp->up; // NO recursion, but check, delete this!
      if (nxt)
         { if ((nxt->next) || (nxt->up))
              { //ConAsc(refnum, "?F1", ZERO); //
                SayErr(refnum,ErRecursionInQuery);
                return 0;  // fail; MUST also signal error
              };
           if (! GutMe(refnum, nxt))
              { return 0;
              };
         };
      nxt = cp->next;
      if (! GutMe(refnum, cp))
         { return 0;
         };
      cp = nxt;
    };
  return 1;  // ok.
}
```

12.2 Special condition check

CheckSpecial returns nonzero if string starts with “DATE ” or “TIME ” or “TIMESTAMP ” otherwise, 0. (nonzero: D=date, T=time, S=timestamp). // in the following we had an error as we’d left out the == 0 for xCompare, fixed 20/8/2006.

```
Int16 CheckSpecial (Char * strg, Int16 slen)
{ Char c;
  c = * strg;
 if (slen < 13)  // okay DATE = 5, then need 4+1+1 + 2 separators
    { return 0;  // eg "DATE 2004-2-3"
    };
 if (c == 'D')
     { if (xCompare (strg, "DATE ", 5) == 0)
          { return 'D';
          };
       return 0;
     };
  if (c == 'T')
     { if (xCompare (strg, "TIME ", 5) == 0)
          { return 'T';
          };
       if (slen < 24)  // timestamp = 10,
                       // then + 1 + 8 for date and min of 5 for time
          { return 0;  // e.g. "TIMESTAMP 2004-2-3 3:4:5"
          };
       if (xCompare (strg, "TIMESTAMP ", 10) == 0)
          { return 'S';
          };
     };
  return 0;
}
```

12.3 Translate from IF to linked list

MakeQueryItem translates a single clause from intermediate format to an item in a linked list. We submit the inevitable libref, the string to translate (with its length, ilen), and the linked list of database tables we will need to identify operands which are data columns.

```
struct cQUERY * MakeQueryItem (UInt16 refnum,
                               Char * istring, Int16 ilen,
                               struct dbLINK * tbllist)
{
  Char sptype;
  struct cQUERY * newnode;  // current cQUERY node.
```

```
struct cQUERY * upnode;   // only for 2nd reference (up)

Char t;        // test
Char c;        // generic test character
Char * cconst = 0; // 'constant' string
Int16 nlen;    // name length
Char * opst=0; // operator *string*
Int16 oL=0;    // string length will be kept here
               // allocation keeps compiler happy
Int16 olen;    // temporary length counter
Char o;
Int16 ok;
olen = 0;
```

if the total string length is under 5, we fail without further ado.

```
if (ilen < 5)            // minimum: 'XYN V'
   { SayErr(refnum,ErStmtTooShort);
     return 0;
   };
```

We next determine the length of the 'logical operator string'. We only permit alphanumerics. (this is determined in the pre-formatting routine). On success, we set aside space for the operator string. We insert two extra bytes at the end of the string (which we don't record in its length) and clear both to zero. These bytes are for later SQL optimisation.

```
while (olen < ilen-4)          // can't be more: must allow for 'YN V'
  { o = * (istring+olen);      // get operator (if it is one!)
    if ((o < 'A') || (o > 'Z'))
       { if (olen < 1)         // if FAIL..
            { SayErr(refnum,ErBadStmtLogic);
              return 0;
            };
         opst = xNew2(refnum, olen+2, 0x107);
         xCopy (opst, istring, olen);
         * (opst+olen) = 0x0; // braces &
         * (opst+olen+1) = 0x0; //  belt.
         oL = olen;  // keep length
         istring += olen;    // )
         ilen -= olen;       // )move past used characters NB.
         olen = ilen;        // force exit from 'while'
       };
    olen ++;
  };
```

NOW opst MUST be defined; its WORKING length is oL (actual length is +1). Next we identify the name:

```
t = * istring;  // get test
istring ++;
ilen --;
nlen = Advance (istring, ilen, ' ');// find name length, up to blank
if (nlen < 1)
   { SayErr(refnum,ErBadColNameLen);
     Delete2(refnum, opst);
     return 0;
   }; // [?? "is null" : see below]
nlen --; // do not include space in name
newnode = FindColumn (refnum, tbllist, istring, nlen);
          // identify column, create node
```

After creating a new column node using FindColumn, we populate this node:

```
if (! newnode)
   { SayErr(refnum,ErBadColumnNodeCreation);
     Delete2(refnum, opst);
     return 0;
   };
newnode->test = t;
newnode->opstring = opst; //
newnode->olen = oL;       // length of opstring
nlen ++;                  // revert to PAST blank
istring += nlen;
ilen -= nlen;             // move past blank at name end!
```

In the following section we check for IS NULL, clearing the cnstant/cnstlen pair if this is present.

```
if ((t == ISNULL) || (t == ISNOTNULL))
      // if IS NULL, or IS NOT NULL
   { newnode->cnstant = 0;
     newnode->cnstlen = 0;
     return (newnode);
   };

c = * istring;
```

We next need to look for unquoted numerics, and a string in quotes. We won't get to the latter for some time. We must also store the relevant datum type! Because newnode-> type is defined, formatting of numerics as float/integer/formatted text is easy!

```
if ( (c == '-') || ((c >= '0') && (c <= '9'))  )              // if numeric
   { switch (newnode->type)
       { case 'I':
```

```
        cconst = xNew2(refnum, 4,0x108);
        ok = qFormatDatum (refnum, cconst, 4,
                  newnode, istring, ilen); // clumsy
        ilen = 4;
      break;

      case 'N':
        cconst = xNew2(refnum, ilen, 0x109);  // [? ilen=0]
        ok = qFormatDatum (refnum, cconst, ilen,
                           newnode, istring, ilen);
        ilen = ok; // ok is *length*
      break;
```

The following float formatting is hideous because we allocate 9+ilen to a buffer where 8 bytes should do. [Fix me].

```
      case 'F':
        cconst = xNew2(refnum, 8+1+ilen, 0x10A); // aagh!
        ok = qFormatDatum (refnum, cconst, 8+1+ilen,
                           newnode, istring, ilen);
        ilen = 8;
      break;
```

If all else fails, so do we:

```
      default:
        ok = 0;  // fail
    };

  if (ok)
     { newnode->cnstant = cconst;
       newnode->cnstlen = ilen;
       return (newnode);
     };
   SayErr(refnum,ErBadTranslation);
   ConAsc(refnum, "[", fDEBUG_SQL); //
   ConTx(refnum, istring, ilen, fDEBUG_SQL); //
   ConAsc(refnum, "]", fDEBUG_SQL); //
   KillQuery(refnum, newnode);
   return 0;
};
```

We next check for special indicators: DATE, TIME and TIMESTAMP. It's really unfortunate that qFormatDatum *also* checks for DATE, TIME etc.[3] a redundancy which causes no harm other than slight slowing. We should completely remove CheckSpecial.

[3] [FIX ME!]

```
sptype  = (Char) CheckSpecial(istring, ilen);
if (sptype)
   { Int16 flen=0; // keep compiler happy!
     switch (sptype)
       { case 'D':    // DATE
           flen = 8;  // yyyymmdd
           break;

         case 'T':    // TIME
           flen = 12; // hhmmssffffff
           break;

         case 'S':   // TIMESTAMP
           flen = 14; // yyyymmddhhmmss [ffffff OMITTED AT LEAST FOR NOW]
           break;
         // default: should fail on next test!
       };
```

We will still fail if the type is incorrect, but otherwise we format the datum and return.

```
      if (newnode->type != sptype)
               { SayErr(refnum,ErBadDatumType);
                 +OPTIONAL
                  ConAsc(refnum, "[", fDEBUG_SQJ); //
                  ConTx(refnum, &sptype, 1, fDEBUG_SQJ); //
                  ConAsc(refnum, ":", fDEBUG_SQJ); //
                  ConTx(refnum, & newnode->type, 1, fDEBUG_SQJ); //
                  ConAsc(refnum, "]", fDEBUG_SQJ); //
                 -OPTIONAL
                 KillQuery (refnum, newnode); // clean up
                 return 0;
               };
      cconst = xNew2(refnum, flen, 0x10B);
      if (! qFormatDatum (refnum, cconst, flen,
                newnode, istring, ilen)) // if fail
         { SayErr(refnum,ErDatumFormatFailed);
           KillQuery (refnum, newnode);  // clean up
           return 0;
         };
      newnode->cnstant = cconst;
      newnode->cnstlen = flen;
      return (newnode);
    };
```

At last, the single quote:

```
if (c == 0x27) // a single quote! (2)
```

```
     { c = * (istring + ilen -1); // get final character!
       if (c != 0x27)
          { SayErr(refnum,ErBadItemQuote);
            KillQuery (refnum, newnode); // clean up
            return 0;             // fail
          };
       cconst = xNew2(refnum, ilen-2, 0x10C);
          // don't include exterior quotes
       xCopy (cconst, istring+1, ilen-2);
       newnode->cnstant = cconst;
       newnode->cnstlen = ilen-2;
       return (newnode);
     };
```

The only possible option now (seeing as we are disinclined to interpolate constant strings into our SQL :-) is that we are dealing with a second node, something along the lines of
`TABLEA.col1 = TABLEB.col2`.

We set things up so that the first node points to the second.

```
   upnode = FindColumn (refnum, tbllist, istring, ilen);
   if (! upnode)
      { SayErr(refnum,ErBadJoinNode);
        KillQuery (refnum, newnode);
        return 0;
      };
   newnode->up = upnode;                // reference 2nd column
```

It's vitally important that we have at least 0x10 bytes in the cnstant string, as this string is used in the join process (database search).

```
   cconst = xNew2(refnum, 0x10, 0x10D);
   newnode->cnstant = cconst;
   newnode->cnstlen = 0x10;
   return (newnode);
}
```

12.4 Creating a query list

12.4.1 MakeQueryList

We accept a string (istring) in intermediate format:

<opstring><test><dotted.varname><blank><comparator><0x1F> ...

In the above <opstring> is ALWAYS made up of uppercase alphas, and test can NEVER be one of these. The <test> is always a single non-uppercase character. The intermediate format string always ends with the character 0x1F.

We translate this into a linked list of cQUERY nodes. We must also supply the dbLINK list created using CreateTableList. We use FindColumn to identify each column (and associated table) Note the problem generated by e.g.

WHERE tblA.value = tblB.value

We cannot comfortably store both items in one cQUERY structure, so we use the ->up cQUERY pointer to point to the second item!

```
struct cQUERY * MakeQueryList (UInt16 refnum, Char * istring,
                  Int16 ilen, struct dbLINK * tbllist)
{ Int16 eoi; // end of item
  struct cQUERY * topnode = 0;   // first node
  struct cQUERY * cnode=0;  // current node. aagh keep compiler happy
  struct cQUERY * newnode;       // and new one

  while (ilen > 0)
    { eoi = Advance (istring, ilen, 0x1F);
```

We look for the end of the string. If the string is tiny, and ends in 'T', then we know there was no WHERE statement, so we will select all! The MakeQueryNode generates a node populated by nulls.

```
      if (eoi < 1)
         { if (*istring == 'T')
              { return (MakeQueryNode(refnum));
              };
           SayErr(refnum,ErBadQueryList);
           return 0;
         };
```

otherwise we carry on, whirling around linking in newly minted query nodes.

```
      newnode = MakeQueryItem (refnum, istring, eoi-1, tbllist);
      if (! newnode)
         { SayErr(refnum,ErQueryItmFailed);
           KillQuery (refnum, topnode); // kill all!
           return 0;
         };
      if (! topnode)
         { topnode = newnode;
           cnode = newnode;
         } else
```

```
          { cnode->next = newnode;
            cnode = newnode;
          };
        istring += eoi; // move to next item
        ilen -= eoi;
      };
  return topnode;
}
```

That's it.

12.5 Optimising a query list

As mentioned in section 9.2.1, there is some potential for optimisation of our SQL queries. If a node evaluates to FALSE and the next logical operator is an AND, we do not need to evaluate the next node. Similarly (but for the opposite reason) if the node evaluates to TRUE and the next operator is OR.

This seems straightforward, but what of say three nodes (each of which will normally store a result) followed by a sequence of two ANDs? We only need to evaluate the first to FALSE, and we can 'skip' the next two nodes, simply putting 0's on the evaluation stack as the results of testing the remaining two nodes is immaterial. What about four nodes followed by AAO? The sequence here would normally be to stack up the four results, take the first AND and apply it to the topmost two results, then the next AND and finally the OR. But if the first value evaluates to TRUE, we have no need to do any other evaluation.

We have therefore left two bytes of space after each command sequence. This allows us to 'look forward' and eliminate unnecessary evaluation.

The first byte of the two tells us whether the command to optimise is an AND or an OR. If this byte is F, then we're looking at an AND, and if it's T, we want an OR. So if we encounter an F and the most recent result is FALSE, we can optimise (and likewise for TRUE and T). In keeping with our terrible practice of 'S' being called OPSTORE, we've represented 'F' as OPNO and 'T' as OPYES.

The number in the second byte (hexadecimal, not ASCII) tells us how many nodes to skip forward. If we've met OPNO the most recent result is false, and the following byte is one (0x1) then we move to the next node, assume the result of that node is FALSE without evaluating it, and continue evaluation. Conversely for OPYES and true we assume the result was TRUE without evaluation, and bash on.

What of numbers over 0x1? Clearly here we must stick to the logic, for the required number of nodes! So we go through all the motions, we simply don't evaluate the node itself as long as the magic number is 0x1 or greater. At every node we decrease the magic number by one.

Our process is as follows:

1. Keep the first node in a variable MEMINI;
2. ALPHA: Set I = 1; Set NN=MEMINI;
3. BETA: Set NN to the next node after NN;
4. If NN is null, go to GAMMA.
5. Examine the (I-1)th byte in the NN opstring.
6. If this byte is OPAND, write OPNO to the top of the MEMINI opstring, write count to the next byte, and go to GAMMA;
7. If the (I-1)th byte is OPOR, write OPYES, count, and go to GAMMA;
8. Otherwise increment I, and repeat BETA.
9. GAMMA: move MEMINI to next node AFTER MEMINI
10. Exit if MEMINI is null otherwise goto ALPHA.

By default, the two bytes after then end of each cQUERY opstring are 0x0. See Section 13.6 for the actual evaluation. Of course we still have to set up this optimisation. Here's the routine:

```
Int16 OptiList (UInt16 refnum, struct cQUERY* MEMINI)
{
  struct cQUERY* NN;
  Int16 i;
  Char * optr;

      +OPTIONAL
       // ConAsc(refnum, "{", fDEBUG_SQJ); //
      -OPTIONAL

  while (MEMINI->next)
    { NN=MEMINI;
      optr = (MEMINI->opstring)+MEMINI->olen;
        // can this ever fail?
      i = 1;
      +OPTIONAL
       ConAsc(refnum, ":", fDEBUG_SQJ); //
      -OPTIONAL
      while (NN)
        { NN=NN->next;
```

```
          +OPTIONAL
           ConAsc(refnum, ".", fDEBUG_SQJ); //
          -OPTIONAL
          if (NN
              &&(NN->olen >= i) //was error: "> i"
             )
             { if (* ((NN->opstring)+i-1) == OPAND)
                  { * (optr) = OPNO;
                    * (optr+1) = i;
                    +OPTIONAL
                    ConAsc(refnum, "&", fDEBUG_SQJ); //
                    ConI  (refnum, i, fDEBUG_SQJ);
                    -OPTIONAL
                    NN = 0; // force inner while exit
                  } else
                  { if (* ((NN->opstring)+i-1) == OPOR)
                       { * (optr) = OPYES;
                         * (optr+1) = i;
                         +OPTIONAL
                          ConAsc(refnum, "|", fDEBUG_SQJ); //
                          ConI  (refnum, i, fDEBUG_SQJ);
                         -OPTIONAL
                         NN = 0; // force exit
                       };
                  };
             };
          i ++;
        }; // end while NN
      MEMINI = MEMINI->next;
    }; // end while MEMINI

      +OPTIONAL
      // ConAsc(refnum, "}", fDEBUG_SQJ); //
      -OPTIONAL
  return 1; // success
}
```

12.6 Other subsidiary routines

12.6.1 Count number of records

MyCountRecords has two advantages over the previous function we defined called CountRecords, which was based on the PalmOS DmDatabaseSize function.

1. It doesn’t need a LocalID (required by DmDatabaseSize);
2. It’s 10 000 times faster!

In order to implement our count, we need to store the number of database records in the database header. There are several caveats:

- The Perl program must store this count on in a number in our header record on creating the PDB file;
- We need to access this number in order to read the count;
- If we add a row to a table, we must increment the number;
- Were we to delete a row, we would have to decrement the number;
- We need to be careful that at no time does PalmOS do funny things with records without us being aware, as this will result in confusion about the number of rows, and potential disaster.

The requirement for storing the number in the header is met by storing the number as a *negative value* at offset +8 in our header record (record zero of each table). (See the section 'Our own Header' in the file *PerlPgm.tex*, and the routine MakeOnePDB there). This 32 bit number must always be -ve, so to store the record count we simply negate the number of records after *adding one* (that is, +1 for the header record itself).

```
Int16 MyCountRecords(UInt16 refnum, struct dbLINK * dbtbl)
{
  UInt32 recs; // (never over 16K)
  Char * hptr;

  if (! dbtbl)
     { SayErr(refnum,ErBadRecordCount);
       return 0; //fail
      };
  hptr = dbtbl->head;
  recs = - (ReadInt32(hptr+0x8));
  if ((recs < 0) || (recs > 16383))
     { SayErr(refnum,ErBadRecordCount);
       return 0; //fail
      };

  return ((Int16) recs);
}
```

The use of ReadInt32 should be safe as the pointer is on an integral boundary divisible by 4. We might dress things up by allocating two distinct error codes in the above.

13 Actual SQL searching and comparison

13.1 Do a join

Here we join using a primary key. Our initial strategy was to use the DmComparF function *janet*, but now we're implementing an indexed approach.

The cQUERY structure jnode contains a reference to a SQL 'data table' which in turn references the current row. This data row is referenced by the dbLINK structure db1, and we obtain the primary key for the row from the correct offset. The *up* component of jnode also contains a dbLINK structure we'll call db2. This references a second SQL data table we will search to find the corresponding primary key we'll use for the join.

PerformJoin must:

1. Determine value from first node;
2. Get table from 2nd node, use binary search to find corresponding 2nd item;
3. Fill in value.

The first value must be appropriately structured. PerformJoin is only called by TestLogic.

Note that in accessing our index file for a particular data table, the dbLINK pointer referring to a particular table contains the name of the data table, and its length (→name and →nlen respectively).

```
Int16 PerformJoin (UInt16 refnum, struct cQUERY * jnode, DmComparF * janet)
{
  struct dbLINK * db1;
  Char * srcp;
  Int16 sstart;
  Int32 jkey;
  Int32 j2; // for debugging, mainly
  UInt16 sortpos;
  struct dbLINK * db2;
  DmOpenRef pdb2;
  struct cQUERY* UPnode;
  MemHandle hmyrow;
  Char * rowp;

  Int16 idxpos;
  UInt16 IDXLIB;
```

We get the relevant data table from the node provided, and obtain the current row in that table. As joins are always on integer primary keys, it's easy to obtain the primary key (jkey) from the relevant column in that row.

```
db1 = jnode->dtable;  // get data table
srcp = db1->current;  // get current row
sstart = ReadInt16X( srcp+0x10 + 2*(jnode->col));
jkey = SafeReadInt32(srcp+sstart); // read the datum
```

Next we actually find a record (if it exists). We get the database table referred to by the second node, and find the matching record using our callback function janet.

```
UPnode = jnode->up;
if (! UPnode) // bad structure
   { SayErr(refnum,ErNoJoinUp);
     return 0;
   };
db2 = UPnode->dtable; // NB the *second* node.
pdb2 = db2->odb;
```

We try our indexing function. Given the value in jkey, we search the relevant index for the desired record.

```
IDXLIB = GetIndexRef(refnum);

if (IDXLIB)
   { idxpos = FINDROW (IDXLIB, db2->name, db2->nlen, 1,
                       (Char *)&jkey, 4, 0);
     if (idxpos > 0) // did not fail: [hmm]
        { hmyrow = s_DmQueryRecord(pdb2, idxpos);
          rowp = (Char *) s_MemHandleLock(hmyrow);
          j2 = ReadInt32(rowp+8);
          if (jkey == j2) // a match (braces+belt)
             { db2->current = rowp; // link to new row! (hooray)
               return 1; // SUCCESS!
             }; // else write failure to console:
          // failure is serious so ...
          TurnOffIndexing (refnum); // disable indexing
          ConAsc(refnum, "[? match ", 0); // err msg
          ConTx (refnum, db2->name, db2->nlen, 0);
          ConI  (refnum, idxpos, 0);
          ConAsc(refnum, "|", 0);
          ConI  (refnum, jkey, 0);
          ConAsc(refnum, "|", 0);
          ConI  (refnum, j2, 0);
          ConAsc(refnum, "]", 0);
          // might even make more of a song+dance?
          s_MemPtrUnlock(rowp);
        };
   };
```

If indexing failed or is disabled, we do things the hard way. The first line of the following is crucial. We store the integer at offset +8, so that the callback function 'janet' will see the integer at the appropriate position! We previously set aside 0x10 bytes in the ->cnstant.

```
  WriteInt32((jnode->cnstant)+8, jkey);
  sortpos = s_DmFindSortPosition (pdb2, (void *) jnode->cnstant,
                                  0, janet, 0);

  if (sortpos > 1)
     { sortpos --; // PalmOS returns value 1 above!
       hmyrow = s_DmQueryRecord(pdb2, sortpos);
       rowp = (Char *) s_MemHandleLock(hmyrow);
       if (jkey == ReadInt32(rowp+8)) // not a match
          { db2->current = rowp; // link to new row! (hooray)
            return 1; // ok.
          };
       s_MemPtrUnlock(rowp); // failed, so not needed.
     };
  // SayErr(refnum,ErSqJoinFail);
  // temp disabled the above !

  return 0; // fail.
}
```

If the search fails, the position points to the node *above which* insertion would be, so we still need (eugh) to *check* that values are identical.[4] We must subtract 1 to point to the actual node (success results in a pointer 1 higher).

The value at rowp+8 is on an integral boundary divisible by 4.

A return of zero signals failure — if it fails then whole line must fail, not just this 'condition'! In fact, if we fail, we should signal this breach in relational integrity.

13.2 compare two text pointers

Given two pointers, compare them using the 'mode'. We return 1 on success, 0 on failure, and a negative number on error.

```
Int16 SeekText (Char * current, Int16 clen, Char * target,
                Int16 tlen, Char mode)
{
  switch (mode)
```

[4] Technically, if our database is intact, then we *cannot* fail, but it would be naive to believe in fairies!

```
    { case ISEQUAL:
        return xSame (current, clen, target, tlen);

      case ISGREATER:
        if (clen <= tlen)  // if first not longer, simply compare
           { if (xCompare (current, target, clen) == 1)
                { return 1;
                };
             return 0;
           };  // need this and the following to prevent buffer overrun!
        if (xCompare (current, target, tlen) >= 0) // NB. if test equal, then
           { return 1;     //  current is GREATER (length greater)!
           };
        return 0;

      case ISLESS:
        if (tlen <= clen)  // compare this with GREATER
           { if (xCompare (current, target, tlen) == -1)
                { return 1;
                };
             return 0;
           };
        if (xCompare (current, target, clen) <= 0)
           { return 1;     // if tlen > clen and 'equal', LESS!
           };
        return 0;

      // default: // fail
      //  return -SqErTextComparisonMode;

    };
  return -SqErTextComparisonMode;
}
```

SeekTextX compares two text strings of given length. Note that if the lengths don't match a buffer overrun and error might result. Because the length is specified, we do *not* look for an ASCIIZ 0x0 termination.

```
Int16 SeekTextX (UInt16 refnum, Char * current, Int16 clen,
                 Char * target, Int16 tlen, Char mode)
{
  Int16 ok;

+OPTIONAL
//   ConAsc(refnum, "<", fDEBUG_SQJ); // ???
//   ConTx(refnum, current, clen, fDEBUG_SQJ); //
//   ConAsc(refnum, ":", fDEBUG_SQJ); //
//   ConTx(refnum, target, tlen, fDEBUG_SQJ); //
```

```
//   ConAsc(refnum, ">", fDEBUG_SQJ); //
-OPTIONAL

  ok = SeekText(current, clen, target, tlen, mode);

+OPTIONAL
   ConI  (refnum, ok, fDEBUG_SQJ);
-OPTIONAL


  if (ok < 0)
     { SayErr(refnum,ErComparisonFailed);
       return 0;
     };
  return ok;
}
```

13.3 Compare numerics

SeekNumber is similar to SeekText. We are dealing with fixed-point numerics. The comparison functions should be revised in terms of IEEE854/754r

```
Int16 SeekNumber (Char * current, Int16 clen,
                  Char * target, Int16 tlen, Char mode)
{
  switch (mode) //
    { case ISEQUAL:
       return (xSame (current, clen, target, tlen));

      case NOTEQUAL:
      if (clen != tlen)
         { return 1;   // must be 'notequal' if dissimilar lengths
         };
       return (! xSame (current, clen, target, tlen));
```

Simple cases are equal and non-equal.

```
      case LESSEQUAL:        // ie NOT greater
        if (clen != tlen)    // pass on if lengths are equal!
           { if (clen > tlen)// number MUST BE greater if length is greater!
                { return 0;
                };
             return 1;
```

```
          };
        return (! SeekText (target, tlen, current, clen, ISGREATER));
        // a <= b implies !(a > b)

      case ISGREATER:
        if (clen != tlen)    // pass on if lengths are equal!
           { if (clen > tlen)// number MUST BE greater if length is greater!
                { return 1;
                };
          return 0;
           };
      break;
```

Otherwise we use a variety of invocations of SeekText to get the job done.

```
      case GREATEREQUAL:     // ie NOT less
        if (clen != tlen)    // pass on if lengths are equal!
           { if (clen > tlen)// number MUST BE greater if length is greater!
                { return 0;
                };
          return 1;
           };
        return (! SeekText (target, tlen, current, clen, ISLESS));
        // a >= b implies !(a < b)

      case ISLESS:
        if (clen != tlen)    // pass on if lengths are equal!
           { if (clen > tlen)// number MUST BE greater if length is greater!
                { return 1;
                };
          return 0;
           };
      break;

      default:
       return -SqErIntComparisonMode;
    };
  // note the break statements above:
  return SeekText (current, clen, target, tlen, mode);
};
```

13.4 Compare using node

Given a node, determine the referenced value in the current string, and perform required comparison. Return 1/0.

```
Int16 SeekCompare(UInt16 refnum, struct cQUERY * node)
{ struct dbLINK * dtable;
  Char * thisp;
  Char * target;
  Int16 tlen;
  Char mode;
  Char dtype;
  Char * current;
  Int16 clen;
  Int16 sstart;
  Int16 fromhere;
  Int16 ok;
```

First we access the relevant record from the data table (thisp points to it). We go to the relevant offset (columns are ordered, so simply get the column, multiply by two, and add 0x10). We can determine the length of the item referenced by accessing the offset of the *next* column.

```
  dtable = node->dtable;
  thisp = dtable->current;  // clumsy
  fromhere = 0x10 + 2*(node->col);
  sstart = ReadInt16X( thisp+fromhere);
  clen = ReadInt16X( thisp+2+fromhere) - sstart;
          // find length of item to compare
  current = thisp + sstart; // point to item
```

We next compare the value to the 'target' one, which at present is limited to the `cnstant` value stored in the node.[5] The 'mode' value tells us what comparison to make.

```
  target = node->cnstant;
  tlen = node->cnstlen;
  mode = node->test;
  dtype = node->type;

+OPTIONAL
   ConAsc(refnum, "\n?(", fDEBUG_SQJ); // ???
   ConTx(refnum, &mode, 1, fDEBUG_SQJ); //
   ConAsc(refnum, ")", fDEBUG_SQJ); //
-OPTIONAL

  if (mode == ISNULL)
       { if (clen == 0)
            { return 1;
            } else
```

[5] Needs an upgrade!

```
            { return 0;
            };
      };
```

We move through NULL, NOT NULL, ...

```
if (mode == ISNOTNULL)
     {
       if (clen == 0)
          { return 0;
          } else
          { return 1;
          };
     };
```

...and various comparisons involving numerics, floats, integers, and so forth. We can largely just regard everything as a numeric string with an appropriate length (8 bytes for floats, 4 bytes for integers, for example). SeekNumber takes into account the peculiarities of our current numerical format.

```
    switch (dtype)
      {
        case 'N':
          ok = SeekNumber(current, clen, target, tlen, mode);
          if (ok < 0)
             { SayErr(refnum,ErNumberSeekFailed);
             return 0;
             };
          return ok;

        case 'F': // might check lengths?
          tlen = 8;
          clen = 8;
          goto SCcommon; // ugly but useful

        case 'I':   // usually only KEYs
          tlen = 4;
          clen = 4; //
        // DO NOT BREAK, continue to default...
        case 'V':   // (all of these)
        case 'D':
        case 'T':
        case 'S':
```

Here we have the common comparison section. For LESSEQUAL and GREATEREQUAL we negate the result, as not ($>$) is the same as less than or equal.

```
SCcommon:
       if (mode == LESSEQUAL)
          { return(! SeekTextX (refnum,
                                current, clen, target, tlen, ISGREATER));
          };
       if (mode == GREATEREQUAL)
          { return (! SeekTextX (refnum,
                                 current, clen, target, tlen, ISLESS));
          };
       if (mode == NOTEQUAL)
          { return (! SeekTextX (refnum, current, clen,
                                 target, tlen, ISEQUAL));
          };
       return(SeekTextX (refnum, current, clen,
                         target, tlen, mode));

      // default:  // no default, just fail! (bad type)
      }
     SayErr(refnum,ErGenericSeekFailure);
     return 0; // fail.
}
```

13.5 Logical processing

LoopLogic is used by TestLogic. Apart from the refnum used to write externally, it accepts bit logic on a stack, and a node which provides a series of commands (at *optr*) of length olen.

```
UInt32 LoopLogic(UInt16 refnum,  struct cQUERY * testnode, UInt32 stk)
{ Char op;
  Char * optr;
  Int16 olen;

+OPTIONAL
   ConAsc(refnum, "[", fDEBUG_SQJ); //
-OPTIONAL

  optr = testnode->opstring;
  if (*optr == OPSTORE)
     {
+OPTIONAL
   ConAsc(refnum, "]", fDEBUG_SQJ); //
-OPTIONAL
       return stk; // do nothing
     };
  olen = testnode->olen;
```

```
  while (olen > 0)
    { op = * optr;  // get first operation to perform
      switch (op)
        { case OPAND:
          +OPTIONAL
           ConAsc(refnum, "&", fDEBUG_SQJ); // ???
           ConI  (refnum, (stk & 3), fDEBUG_SQJ);
          -OPTIONAL
            if ((stk & 3) != 3)
               { stk &= 0xFFFFFFFD; // clear bit #1
               };
            break;

          case OPOR:
          +OPTIONAL
           ConAsc(refnum, "|", fDEBUG_SQJ); // ???
           ConI  (refnum, (stk & 3), fDEBUG_SQJ);
          -OPTIONAL
            if (stk & 3)  // either bit set?
               { stk |= 0x00000002; // set bit #1
               };
            break;

          //----------------------//
          // what about NOT ?
          //----------------------//

          default:
          SayErr(refnum,ErBadLogicalOp);
          return 0; // 'fail'
        };

      stk /= 2;   // get rid of rightmost bit
      optr ++;
      olen --;
    };

+OPTIONAL
   ConAsc(refnum, ":", fDEBUG_SQJ); //
   ConI  (refnum, stk, fDEBUG_SQJ);
   ConAsc(refnum, "]", fDEBUG_SQJ); //
-OPTIONAL

  return stk;
}
```

13.6 TestLogic — The main logic test

TestLogic uses a prepared sequence of test nodes and a bit-stack stored in a 32 bit integer. Determines whether a particular row evaluates to true or false: we return 1 if true, 0 if false. It is called both by SQL3UPDATE and by SeekMany, which in turn is called by SeekWrap and SQL3SELECT.

At present we use no short-cuts, but note that eg. a AND b AND c evaluates to F if ANY item is false; we could make use of such logic to speed evaluation considerably. [FIX ME!]

Whenever we join on tables: tableA.value1 = tableB.value2[6] we must:

1. retrieve the left value
2. find a corresponding unique (key) value in the second table using a binary search
3. Insert the corresponding row into the dbLINK->current value!

```
Int16 TestLogic (UInt16 refnum,
      struct cQUERY * testnode, DmComparF * janet)
{ UInt32 stk = 0;
  Int16 ok;
  Int16 yesno;
  Int16 magic;
```

In the absence of a WHERE statement, we will always return ‘success’, thus:

```
if (! testnode->type)
   {
     return 1; // null used to say NO "WHERE" component
   };
```

We otherwise move across the whole of the chain of test nodes, evaluating each one using a *while* statement. We use *up* nodes to join across tables. In all other cases we perform standard comparisons using SeekCompare.

```
  while (testnode)
    {
      if (testnode->up) // signals a join
         { if (! PerformJoin(refnum, testnode, janet))
              { return 0; // failed join FORCES failure.
              };
```

[6] Perhaps signal this using testnode->olen = 0, and NOT some other rather arbitrary signal?!

The following is a bit of a hack. Because AND is seen with the join condition we must put in a 1 on stack if join succeeds. This is *ugly*.

```
      stk *= 2; // shift left
      stk |= 1; // force in a 1 for the join!
      //
    } else // standard comparison:
    {
      ok = SeekCompare(refnum, testnode);
```

We shift the logical 'bit-stack' and put the result (ok) into the lowest bit.

```
      stk *= 2; // shift left
      stk |= ok; // OR in new bit
    }; // END OF THE IF THEN ELSE..
```

If the command is simply OPSTORE (S for store), we move on, but otherwise we evaluate the logic using the commands in optr combined with the bit-stack. (LoopLogic now tests for the OPSTORE).

```
+OPTIONAL
 ConAsc(refnum, ".", fDEBUG_SQJ); //
 ConI  (refnum, stk, fDEBUG_SQJ);
-OPTIONAL
stk = LoopLogic(refnum, testnode, stk);
```

EVERY node has two extra optimisation bytes at the end of optr. These are NOT included in olen. After performing LoopLogic (if needed) we can NOW use these bytes to good effect (if they are nonzero). See section 12.5 for more details. The routines make the reasonable assumption that the magic number is valid, and won't cause progression to a null 'next' node!

If OPYES is encountered and there is a 1 (true) on the stack, we know that an *OR* was specified, and we don't need to test a result, as either would have given a true, and we already have a true!

Conversely if OPNO is encountered, then AND will fail provided the stack value is zero.

```
yesno = *(testnode->opstring+testnode->olen);
if (yesno)
   {
     +OPTIONAL
      // ConAsc(refnum, "{", fDEBUG_SQJ); //
     -OPTIONAL
     if (yesno == OPYES)
        {
```

```
                +OPTIONAL
                 ConAsc(refnum, "|", fDEBUG_SQJ); //
                -OPTIONAL
                if (stk && 1) // if true
                   {
                     magic = *((testnode->opstring)+1+testnode->olen);
                     +OPTIONAL
                      ConAsc(refnum, "+@", fDEBUG_SQJ); //
                      ConI  (refnum, magic, fDEBUG_SQJ);
                     -OPTIONAL
                     while (magic > 0)
                       { testnode = testnode->next;
                         stk *= 2;
                         stk |= 1; // put in a one
                         stk = LoopLogic(refnum, testnode, stk);
                         // we ignore the magic in this node!
                         magic --;
                       };
                   };
             } else // assume OPNO [?]
             {
               +OPTIONAL
                ConAsc(refnum, "&", fDEBUG_SQJ); //
               -OPTIONAL
               if (! (stk && 1)) // if false
                   {
                     magic = *((testnode->opstring)+1+testnode->olen);
                     +OPTIONAL
                      ConAsc(refnum, "-@", fDEBUG_SQJ); //
                      ConI  (refnum, magic, fDEBUG_SQJ);
                     -OPTIONAL
                     while (magic > 0)
                       { testnode = testnode->next;
                         stk *= 2; // shift in a zero
                         stk = LoopLogic(refnum, testnode, stk);
                         // we ignore the magic in this node!
                         magic --;
                       };
                   };
             };
           +OPTIONAL
           // ConAsc(refnum, "}", fDEBUG_SQJ); //
           -OPTIONAL
         };
      testnode = testnode->next;
    }; // END WHILE
  return (stk & 1);
};
```

In the above we *must* test and run the invocation of LoopLogic if an AND condition occurs with a join. This is ugly, as noted above.

We return the topmost stack flag (least bit) only.[7]

13.7 Clear prior joins

For each database, if $->$current isn't cleared, clear it! MUST start with node$->$ primary database. (NB).

```
Int16 ClearJoin (struct dbLINK * dbp)
{ MemPtr mp;
  while (dbp)
    { mp = (MemPtr) dbp->current;
      if (mp)
         { s_MemPtrUnlock(mp); // could check for failure ???
           dbp->current = 0;   // force zero.
         };
      dbp = dbp->next;
    };
  return 1; // success.
}
```

13.8 WriteAnswer

Writing of SQL queries. We have replaced previous writes to the 'scratch' buffer with stack writes.

```
Int16 WriteAnswer(UInt16 refnum, struct cQUERY * ans,
                  Char * STACK, Char * STACKSTRING)
{
  struct dbLINK * db;
  Char * srcp;
  Int16 itm;
  Int16 dstart;
  Int16 dlen;
  Int16 ok;

  while (ans)
    { db = ans->dtable;  // get data table
      if (! db)
         { SayErr(refnum,ErBadDataTable);
           return 0;
         };
```

[7] We might check that there is nothing higher, or even have another bit flag for the top?!

```
      srcp = db->current;  // get current row
      if (! srcp)
         { SayErr(refnum,ErJoinConditionOmitted);
           return 0;
         };
      itm = 0x10 + 2*(ans->col);                // for offset of datum

      dstart = ReadInt16X( srcp+itm);  // read offset of datum
      dlen = ReadInt16X( srcp+2+itm) - dstart;// and get length
      if (ans->col == 0)               // if key column
         { dlen = 4;                   // force length to 4
         };                            // (we will probably regret this often)
```

In the following we have the option of writing the (raw) debug string to the console, if deeper SQL debugging is set.

```
      +OPTIONAL
      ConAsc(refnum, "\n <-(", fDEBUG_SQK); //
      ConTx (refnum, srcp+dstart, dlen, fDEBUG_SQK); // raw
      ConAsc(refnum, ")", fDEBUG_SQK); //
      ConTx (refnum, & ans->type, 1, fDEBUG_SQK); // type
      ConAsc(refnum, "[", fDEBUG_SQK);
      ConI  (refnum, ans->scale, fDEBUG_SQK);  // [scale]
      ConAsc(refnum, "]", fDEBUG_SQK);
      -OPTIONAL
      ok = PushItem(STACK, STACKSTRING,
                  srcp+dstart, dlen, ans->type, ans->scale);
      if (ok != 1)
         { SayErr(refnum,ErFailedDatumPush);
           ConI(refnum, -ok, ZERO);
           ConTx(refnum, &(ans->type), 1, ZERO); //
           ConI(refnum, dlen, ZERO);
           // [FIX ME: JVS: HERE SHOULD WRITE A *NULL* to stack! (least silly!?)]
           return 0;
         };
      ans = ans->next;  // move to next one
    };
  return 1;
}
```

13.9 Select with multiple results — SeekMany

SeekMany performs the SELECT operation. It is a monster. In order to ease functioning, we assume a sequence of table names — in other words, we flout the ‘unordered’ nature of lists of table names inherent in true SQL. This isn’t as bad as it sounds, in that it is generally a trivial task to take an arbitrary SQL statement and order the table names as we desire! *Our* SQL code should work in any database.

Our **constraining rule** is that we work from *leaf to branch*. In other words, the first table mentioned can reference the primary key of tables to the right, but not the other way around![8] We *never* do a true Cartesian join (as such joins are excessively silly).

SeekMany accepts the following arguments:

refnum for the library reference;

tablename, tbllen A comma-delimited list of table names, with the list length;

rsltname, rnamelen A list of columns to return (rsltname) and its length;

srchstring, srchlen A query string and its length;

onlyone A modifier, to return only one result;

STACK, STACKSTRING The stack and extended string stack;

janet A callback comparison function.

In order for this function to work, the list of table names must be comma-delimited, and *contain no spaces*. To implement later use of indexing, we here ensure that each table referred to has an appropriate index. This indexing check is performed within CreateTableList.

```
Int16 SeekMany (UInt16 refnum, Char * tablename, Int16 tbllen,
            Char * rsltname, Int16 rnamelen,
            Char * srchstring, Int16 srchlen,
            Int16 onlyone,
            Char * STACK, Char * STACKSTRING,
            DmComparF * janet)  // results are written to STACK
{ struct dbLINK * dbp;
  struct cQUERY * rsltnode; // later make this a linked list ?!
  struct cQUERY * testnode;
  Int16 recs;
  Int16 optim = 0;
  MemHandle thishand;
  Char * thisp;
  Int16 rec=1;  // start at 1 (not 0 which is header)
  Int16 hits = 0;
  DmOpenRef pdb1;
  Char * con16;
```

Our linked lists of nodes are rooted in the dbp, testnode and rsltnode nodes. We can also optimise things *a lot* if our sole search criterion is on the primary key of a table, as we can then use a binary search to locate this key (tables are ordered by primary key). We use the optim flag (=1) to signal this fortunate circumstance!

[8] We still need to consider cyclical references ...

Create a table list

First, we create a linked list of PalmOS databases, that is, data tables, rooted in ‘dbp’.

```
//  ConAsc(refnum, "\n\nT:", fDEBUG_SQJ); // debug table list
//  ConTx (refnum, tablename, tbllen, fDEBUG_SQJ);

  dbp = CreateTableList(refnum, tablename, tbllen);
  if (! dbp)
     { SayErr(refnum,ErFailMakeDbList);
       return  0; // fail
     };
```

Create result node list

We root the result column list in ‘rsltnode’.

```
 // ConAsc(refnum, "\nR:", fDEBUG_SQJ); // rslt col list
 // ConTx (refnum, rsltname, rnamelen, fDEBUG_SQJ);

  rsltnode = FindResultColumns (refnum, dbp, rsltname,
                                rnamelen);
```

Now for the query node(s)

The query nodes are rooted in ‘testnode’. This sequence of nodes precisely describes our query (search).

```
//  ConAsc(refnum, "\nQ:", fDEBUG_SQJ); // query
//  ConTx (refnum, srchstring, srchlen, fDEBUG_SQJ);

  testnode = MakeQueryList (refnum, srchstring,
                            srchlen, dbp);
 if (! testnode)
    { KillQuery (refnum, rsltnode);
      KillDbList(refnum, dbp);
      return 0;  // fail [??error msg]
    };
```

We fail if the query list creation failed. Otherwise, we set the optim flag (if appropriate).

```
  if (    onlyone  // NOT if more than one may be desired!
      && (! testnode->next)
      && (testnode->type == 'I')
      && (testnode->col == 0)
```

```
     )
     { optim = 1; // clumsy
     };
```

Next, find the number of records in the leftmost ('primary') table. If this number is under 2 (only a header, so no match, return 1 (ok, but no record, unless CountRecords failed, in which case we return 0).[9]

```
 recs = MyCountRecords(refnum, dbp); // includes header
  +OPTIONAL
  ConAsc(refnum, "\nN=", fDEBUG_SQL);
  ConI  (refnum, recs, fDEBUG_SQL);
  -OPTIONAL

 if ( recs < 2 )  // 1 means 'only header' so NO match!
    { KillQuery(refnum, testnode); // 10-7-2006 fix
      KillQuery (refnum, rsltnode);
      KillDbList(refnum, dbp);
      return recs;  // 0 = fail 1=ok but no record
    };
```

A special case

If we only want one record based on a primary key (optim=1) we return this. We do so by setting aside 0x10 bytes of memory in which we store the desired key (as a big-endian 32 bit number) at offset +8. This memory area is 'con16'. We then use the *janet* callback to locate the desired record in our database, the handle of which is in 'pdb1'.

Once we think we've located the record (the callback returns us to either one above the record, or *where the record would be if it existed*) we must still make sure that the record is actually present.

```
if (optim)
   { con16 = xNew2(refnum, 0x10, 0x10E);
     xCopy(con16+8, testnode->cnstant, 4);
     pdb1 = dbp->odb;
     rec = s_DmFindSortPosition (pdb1, (void *) con16, 0, janet, 0);
     if (rec > 0)
        { rec --; // go to actual record
          thishand = s_DmQueryRecord(dbp->odb, rec);
          // might test for failure (zero thishand)
          thisp = (Char *) s_MemHandleLock(thishand); // point to it
          if (xSame(thisp+8, 4, con16+8, 4))
```

[9] We must remember to free up the nodes; failure to do this previously resulted in embarrassing memory leaks!

```
            { ConAsc(refnum, "!", fDEBUG_SQL);
              dbp->current = thisp; // clumsy
              WriteAnswer(refnum, rsltnode, STACK, STACKSTRING);
              hits = 1;
            };
          s_MemPtrUnlock(thisp);
        };
     Delete2(refnum, con16); // free me
```

In the above, if we have found the answer, we write the result to the stack using WriteAnswer (Section 13.8). We remember to delete our temporary con16 buffer.

The general case

Alternatively, in the more general case where we need to examine several records, we do so. For each rec[ord] we open the record as read only, and store a pointer to the record in dbp− >current.

The invocation of OptiList in the following code introduces some SQL logical optimisation, but profiling demonstrates little effect of such optimisation on most SQL execution.[10]

```
   } else

   {
     OptiList(refnum, testnode); // optimise query for TestLogic!
     while (rec < recs) // for each record
       {
         thishand = s_DmQueryRecord(dbp->odb, rec);
               // get current record, read only.
         if (thishand) // unless failed
            { thisp = (Char *) s_MemHandleLock(thishand);
              dbp->current = thisp; // clumsy
```

We now apply our logic tree using TestLogic (Section 13.6). This returns a 1 or 0 value. On success we invoke WriteAnswer, and if we only ever wanted one result, we force an exit. After *each* test, we clear the dependent row references (ClearJoin: rather slow, examine me).

```
              if ( TestLogic (refnum, testnode, janet))
                 { hits ++;
+OPTIONAL
   ConAsc(refnum, "!!", fDEBUG_SQJ); // signal a hit!
```

[10] Smart testing before joins, where possible, may improve this effect!

```
-OPTIONAL
                    WriteAnswer(refnum, rsltnode, STACK, STACKSTRING);
                    if (onlyone)
                       { rec = recs; // force exit.
                  };   };
              ClearJoin(dbp);
            } else
            { SayErr(refnum,ErRecQNotFound);
              ConI(refnum, rec, ZERO);
            };
          rec ++;
        };
   }; // end if then else.
```

Because we invoke ClearJoin in the above, we do *not* need to MemPtrUnlock *thisp* in the above.

Cleaning up

We delete the query nodes and return . . .

```
  ConAsc(refnum, "+=", fDEBUG_SQL);
  ConI  (refnum, hits, fDEBUG_SQL);
  if (   !KillQuery(refnum, testnode)
      || !KillQuery(refnum, rsltnode)
     )
     { SayErr(refnum,ErCollectionFailed);
       return 0;
     };
  if (! KillDbList(refnum, dbp))
     { SayErr(refnum,ErDbFreeFailed);
       return 0;
     };
  return hits+1;
}
```

We return one added to the number of hits, to allow us to signal failure with a zero!

13.9.1 Wrapper for SeekMany

SeekWrap. Given string in intermediate format, stripped of SELECT, FROM and WHERE, perform the actual query. This function is a wrapper for the SeekMany function above (Section 13.9). We first clip the intermediate-format string into its various components, then submit these to SeekMany.

```
//
Int16 SeekWrap (UInt16 refnum, Char * seekTemp, Int16 skL,
                Int16 onlyone,
                Char * STACK, Char * STACKSTRING, DmComparF * janet)
{ Char * tbl;
  Char * rslt;
  Int16 rlen;
  Int16 L;
```

Get target columns

ASCII column names are separated by commas, and there is a *terminal comma*.
The list of column names ends in a SEPARATOR character.

```
  rlen = Advance (seekTemp, skL, SEPARATOR); // move to 1 AFTER separator
  if (rlen < 1)
     { //ConAsc(refnum, "?E1", ZERO); //
       SayErr(refnum,ErNoSeparator);
       return 0;       // fail; hmm should display error?
     };
  rslt = seekTemp;  // target column
  * (seekTemp+rlen-1) = ',';  // force TERMINAL COMMA!
  seekTemp += rlen;
  skL -= rlen;
```

Get table names

Table names are in a similar format to column names.

```
  L = Advance (seekTemp,skL , SEPARATOR);
  if (L < 1)
     { //ConAsc(refnum, "?E2", ZERO); //
       SayErr(refnum,ErDudTableNames);
       return 0;  // as above ?
     };
  tbl = seekTemp;
  * (seekTemp+L-1) = ','; // force TERMINAL COMMA!
  seekTemp += L;
  skL -= L;
```

Invoke SeekMany

seekTemp now points to search criterion string, skL is the length of this string.

```
  return SeekMany (refnum, tbl, L, rslt, rlen, seekTemp, skL, onlyone,
                   STACK, STACKSTRING, janet);
}
```

13.10 Preformatting

The following modification of SQL statements slows things down a bit, but is a bit softer on formatting. We translate into a format that subsequent parsing recognises. All of the following applies only outside SQL 'quotes'.[11]

Preformatting includes:

1. No double(+) spaces
2. ")AND" becomes ") AND", "AND(" becomes "AND (", and likewise for OR
3. "NOT(" becomes "NOT ("
4. "a, b" becomes "a,b" ie no comma followed by a space
5. "a¿b" becomes "a ¿ b" ie spaces around conditions like ¡ ¿ = ¿= ¡=
6. "((" becomes "((" (Must still check out NOT very carefully ???)
7. "SELECT DISTINCT" becomes just SELECT, and the DISTINCT post-processor is set up if distinct used, then at present only one column must be returned, although this will later be modified to allow several. At present we submit col count =1.
8. "SELECT MAX(columname) BECOMES SELECT columname, and MAX post processor is set
9. likewise for SELECT MIN(..)
10. "... ORDER BY colname" should result in use of SORT post-processor, with only 1 column selected AS WELL as submitting the default "A" sort configuration string.
11. "... ORDER BY colname DESC" is similar to ORDER BY colname but submit "D" sort config string!
12. later must write a GROUP BY post-processor fx.

WE RETURN the length of the formatted string in 'dest'. This includes the 16 bytes at the start (see below). A return value of zero indicates failure.

Good to simply to transfer from source string to destination string (more space, less time). Destination string should be bigger than source, although usually the

[11] It would be mildly advantageous to perform such pre-processing prior to storing SQL statements.

output will be shorter than input string. We will arbitrarily reserve the first 16 bytes of dest for our own purposes:

1. +0 First byte is zero (must be)
2. +1 Next byte is post-processing flags
3. +2 Length of following sort configuration string
4. +4–13 bytes are reserved (sort config string)

The actual string to interpret follows from byte 16 onwards! The post-processor signals are shown in table 1. We cannot have sort and max or distinct (?) and max.

Value	Meaning
0	success, NO post-processing required
1 fSORT	SORT (ORDER BY ..)
2 fMAX	MAX
4 fMIN	MIN
8 fDISTINCT	DISTINCT
9	DISTINCT, THEN SORT

Table 1: Post-processor signals

Before we look at preformatting, a checking function: CHECKsame.

13.10.1 A check function

```
// Intial checking function:
Int16 CHECKsame (Char * P0, Int16 maxP0, Char * p1, Int16 lgth)
{
  if (maxP0 < lgth)
     { return 0;
     };
  return LUPSAME (P0, lgth, p1, lgth);
}
```

13.10.2 Get sort column number

The following routine has only one role. Given a string starting with a list of columns (note that the SELECT and the space after it have already been clipped off), and another string points to the column name *within* the string:

```
... ORDER BY columname
```

... determine which one of the selected columns is the same as *columname* and return the number of this one column (1 being the first, 2 the 2nd, and so on). The length of the column is specified, but for the list of columns, we only have the length of the entire string. The columns are all comma-delimited and the last one will be followed by a space, so we can determine the length of this list.

Note that a column name to ORDER BY must (at least for now) be *identical* to the name at the start of the SELECT statement, *including* the table name dotted onto the column name, if this is specified previously.

```
Int16 GetSortColNo ( UInt16 refnum, Char * cols, Int16 colsmax,
                     Char * thiscol, Int16 thislen)
{
  Int16 colcount = 1;
  Int16 comma;
  Int16 colslen;

  colslen = Advance(cols, colsmax, ' ');
  if (! colslen) // if failed
     { return 0; // MUST FAIL
     };

/*  // DEBUG: ////////////////////////
  ConTx (refnum, "\n[", 2, 0);
  ConI (refnum, colsmax, 0);
  ConTx (refnum, ":", 1, 0);
  ConTx (refnum, cols, colslen, 0);
  ConTx (refnum, "]", 1, 0);
  ConTx (refnum, thiscol, thislen, 0);
  ConTx (refnum, "\n", 1, 0);
*/ ////////////////////////////////////

  while (colslen > 0)
    { comma = Advance (cols, colslen, ',');
      if (! comma)
         { comma = colslen;
         };
      // if no comma, will include terminal blank!
/* // DEBUG ////////////////////////////
      ConTx (refnum, cols, comma-1, 0);
      ConTx (refnum, "|", 1, 0);
*/ ////////////////////////////////////

      if ( xSame(cols, comma-1, thiscol, thislen) )
         // don't include comma, if present
         { return colcount;
         };
      colslen -= comma;
```

```
      cols += comma;
      colcount ++;
    };
  return -colcount; // fail
}
```

We return a negative number if the column was not found. This doesn't necessarily mean that an error will result, as it is possible to sort on a column without it appearing at the start of the SELECT statement, so the negative number returned is the number of columns detected in the SELECT statement!

13.10.3 The *Preformatting* routine

This routine still must be checked and properly documented. Note that we obtain data from the variable **srcp**, transferring it to the buffer **dest**, but reserving the first 16 bytes of dest for special purposes. We don't alter the value of dest during this routine, rather writing to an auxiliary pointer *dP*.

In the following it's important to track maxlen, as this value is used in determining the final preformatted string length. As you add new characters to the destination buffer, decrement maxlen in a commensurate fashion (nasty).

```
// *** [DO WE HAVE A PROBLEM WITH = ?]

Int16 Preformat ( UInt16 refnum, Char * dest, Int16 dlen,
                     Char * srcp, Int16 srclen)
{ // ASSUMES submitted string has already had ‘‘SELECT " snipped off!

  Char * dP; // destination written to
  Char c;
  Int16 maxlen = dlen;
  Int16 ppflags=0;  // post-processing flags
  Int16 parcount = 0; // check on parentheses

  Boolean killrpar = 0;     // various signals about past characters..
  Boolean quoted = 0;
  Boolean spaced = 0;
  Boolean comma = 0;
  Boolean logic = 0;
  Boolean sorter = 0;

  Int16 sortnumber=0;
  Int16 sortlen=0;
  Int16 inspos; // insertion position (sort)
  Int16 inslen;

  // here clear first 16 bytes of dest: (default):
  xFill(dest, 0x0, 16);
```

```
if (*(srcp+srclen-1) == ';')  // ignore terminal semicolon
   { srclen --;
   };
dP = dest+16; // will write all to here
maxlen -= 16; // track

// a. check for DISTINCT
if (CHECKsame (srcp, srclen, "DISTINCT ", 9))
   { srcp += 9;
     srclen -= 9;
     ppflags = fDISTINCT;
   };

// b. check for MAX
if (CHECKsame (srcp, srclen, "MAX (", 5))
   { ppflags = fMAX;
     srcp += 5;
     srclen -= 5;
     killrpar = 1; // also must kill right parenthesis
     parcount ++;
   } else
   {  if (CHECKsame (srcp, srclen, "MAX(", 4))
         { ppflags = fMAX;
           srcp += 4;
           srclen -= 4;
           killrpar = 1;
           parcount ++;
   };    };

// c. check for MIN
if (CHECKsame (srcp, srclen, "MIN (", 5))
   { ppflags = fMIN;
     srcp += 5;
     srclen -= 5;
     killrpar = 1; // also must kill right parenthesis
     parcount ++;
   } else
   {  if (CHECKsame (srcp, srclen, "MIN(", 4))
           // lpar immediately after!
         { ppflags = fMIN;
           srcp += 4;
           srclen -= 4;
           killrpar = 1;
           parcount ++;
   };    };
```

We've now looked for fancy stuff up front (DISTINCT, MAX, MIN) so what follows should be a comma-delimited list of column names.

```
while ((srclen > 0) && (maxlen > 0))
   // while still stuff to process, and space to store..
  {
    c = *srcp++; // default is simply transfer character!
    *dP++ = c;   // postincrement
    srclen --;
    maxlen --;

    if (quoted)
       { if (c == 0x27)
            { quoted = 0;
            };
       } else
       { c = UPPERCASE(c);
         switch (c)
            { case ' ':
               if ((spaced) || (comma))  // no space after comma!
                  { dP --;
                    maxlen ++; // remove transferred space
                  };
                spaced = 1;
              break;

              case 0x27:   // single quote'
                quoted = 1;
              goto fred;

              case ',':
                if (spaced) // if preceding space:
                   { dP --; // get rid of comma and
                     *(dP-1) = ',';  // put it where space was
                   };
                spaced = 0; // comma=1 similar signal to spaced!
                comma = 1;
                logic = 0;
              break;

// STILL MUST CHECK FOR < > <> = >= <=
              case '=':
                if (! spaced)
                   { *(dP-1) = ' ';  // insert space
                     *dP++ = '=';    // postincrement
                     maxlen --;
                   };
              if (*(srcp) != ' ')  // if no trailing space
                 { *dP++ = ' ';    //  insert it.
                   maxlen --;      // no spaced, won't be used
                 };
              goto fred;
```

```
case '>':
  if (! spaced)
     { *(dP-1) = ' ';  // insert space
       *dP++ = '>';
       maxlen --;
     };
  c = *(srcp);
  if (c == '=')
     { srcp ++;
       srclen --;
       *dP++ = '=';
       maxlen --;
     };
  if (*(srcp) != ' ')
     { *dP++ = ' '; //  insert it
       maxlen --;   // no spaced signal
     };
goto fred;

case '<':
  if (! spaced)
     { *(dP-1) = ' ';  // insert space
       *dP++ = '<';
       maxlen --;
     };
  c = *(srcp);
  if (c == '=')
     { srcp ++;
       srclen --;
       *dP++ = '=';
       maxlen --;
     } else
     { if (c == '>')
          { srcp ++;
            srclen --;
            *dP++ = '>';
            maxlen --;
     };   };
  if (*(srcp) != ' ')
     { *dP++ = ' ';    //  insert it.
       maxlen --;  // don't need spaced=1 (obviously)
     };
goto fred;

case '(':
  parcount ++;
  if (! logic)
   // if preceding "AND " do NOT delete the space!?
     { if (spaced) // if preceding space:
```

```
            { dP --;
              maxlen ++;
              *(dP-1) = '('; // as for comma
            };
        };
goto fred;   // eugh. resets logic to zero.

case ')':
  parcount --;
  if (parcount < 0)
     { return -SqErBadParens;
     };
  if (spaced) // if preceding space:
     { dP --;
       maxlen ++;
       *(dP-1) = ')'; // as for comma
     };
  if (killrpar)   // if killing rpar
     { killrpar = 0;
       dP --;   // kill it!
       maxlen ++;
     };
  goto fred;
```

Next check for AND, OR, NOT, ORDER BY. These MUST be surrounded by spaces! Tricky where we have e.g. ”AND (”.

```
case 'A':
  comma = 0;
  if (CHECKsame (srcp, srclen, "ND ", 3))
     { logic = 1;
       spaced = 1; // suppress following spaces
       xCopy (dP, srcp, 3);
       dP += 3;
       maxlen -= 3;
       srcp += 3;
       srclen -= 3;
     } else
     { if (CHECKsame (srcp, srclen, "ND(", 3))
          { xCopy (dP, "ND (", 4);
            dP += 4;
            maxlen -= 4;
            srcp += 3;
            srclen -= 3;
            spaced = 0;
            parcount ++;
          } else
          { goto fred;
```

```
                    };   };
                break;

                case 'O':
                  comma = 0;
                  if (CHECKsame (srcp, srclen, "R ", 2))
                     { logic = 1;
                       spaced = 1; // suppress following spaces
                       xCopy (dP, srcp, 2);
                       dP += 2;
                       maxlen -= 2;
                       srcp += 2;
                       srclen -= 2;
                     } else
                     { if (CHECKsame (srcp, srclen, "R(", 2))
                          { xCopy (dP, "R (", 3);
                            dP += 3;
                            maxlen -= 3;
                            srcp += 2;
                            srclen -= 2;
                            spaced = 0;
                            parcount ++;
                          } else
                          { if (CHECKsame (srcp, srclen, "RDER BY ", 8))
                               { sorter = 1;
+OPTIONAL
ConAsc(refnum, "\nSORT:(", fDEBUG_SQK); //
-OPTIONAL
                                 dP --;
                                 maxlen ++;
                                 srcp += 8;
                                 srclen -= 8;
                                 // next get the number of the column (tricky):
                                  sortlen = Advance (srcp, srclen, ' ');
                                  if (! sortlen)
                                     { sortlen = srclen;
                                     } else
                                     { sortlen --; // do NOT include blank [2007-1
                                     };
                                  sortnumber = GetSortColNo (refnum, dest+16, (dP
                                                   srcp, sortlen);
                                  if (sortnumber < 1) // not found:
                                     { // do NOT fail here! Insert new column!
                                       inspos = Advance (dest+16, (dP-(dest+16)),
                                       if (inspos < 0) // error
                                          { SayErr(refnum, ErSqDudSort); // fail
                                            return 0;
                                          };
                                       inspos --; // before the space!
```

```
                                        inslen = InsertString(dest+16+inspos, (dP-(
                                                    dlen-16-inspos,
                                                    srcp-1, sortlen+1);
                                        if (inslen <= 0)
                                           { SayErr(refnum, ErSqDudSort); // fail
                                             return 0;
                                           };
                                        * (dest+16+inspos) = ',' ; // insert comma!
                                        dP += inslen; // adjust for insert
                                        maxlen -= inslen; // track (NB)
                                        sortnumber = - sortnumber; // restore
                                        sortnumber |= 0x40; // FLAG: vanishing colu
                                      };
                                   srcp += sortlen; // if ' ' are pointing there!
                                   srclen -= sortlen;
                                   // c = *(srcp-1); // ensure is up to date! (hmm?
                                   // at present we simply store sortnumber at offs
                                   //   AFTER check for DESC / ASC after the space
                                   if (srclen > 0)
                                      { srclen --;
                                        srcp ++; // move past the blank
                                        if (CHECKsame (srcp, srclen, "DESC", 4))
                                           { sortnumber |= 0x80; // FLAG: desc
                                             dlen -= 5; // clip off " DESC"
                                           } else
                                           { if (! CHECKsame (srcp, srclen, "ASC",
                                                { // [HERE WOULD DEAL WITH MULTIPLE
                                                  SayErr(refnum, ErSqDudSort);
                                                  return 0;
                                                } else
                                                { dlen -= 4; // clip.
                                                };
                                      };   };
                                   if (GetUpturn(refnum)==1) // if 'UPTURN' !!
                                      { sortnumber ^= 0x80; // XOR swops flag
                                      };
                                   * (dest+6) = sortnumber;
+OPTIONAL
ConI(refnum, sortnumber, fDEBUG_SQK); // number of sort column:
ConAsc(refnum, ")", fDEBUG_SQK);      //   May be vanishing
-OPTIONAL
                                } else
                                { goto fred;
                           };   };   };
                   break;

                   case 'N':
                     comma = 0;
                     if (CHECKsame (srcp, srclen, "OT ", 3))
```

```
                    { logic = 1;
                      spaced = 1; // suppress following spaces
                      xCopy (dP, srcp, 3);
                      dP += 3;
                      maxlen -= 3;
                      srcp += 3;
                      srclen -= 3;
                    } else
                    { if (CHECKsame (srcp, srclen, "OT(", 3))
                         { xCopy (dP, "OT (", 4);
                           dP += 4;
                           maxlen -= 4;
                           srcp += 3;
                           srclen -= 3;
                           spaced = 0;
                           logic = 0;
                           parcount ++;
                         } else
                         { goto fred;
                    };   };
               break;

               fred:
               default:
                 spaced = 0;
                 comma = 0;
                 logic = 0;
             };
       };

  };

if (srclen != 0)
   { SayErr(refnum, ErSqPreFull);
     return 0;
   };

if ( quoted || logic || parcount)
   { SayErr(refnum, ErSqPreLogic);
     return 0;
   };

if (sorter)
   { ppflags |= fSORT; // clumsy
     // we used to use *(dest+2)=length, now unnecessary!
   }; // clumsy.
*(dest+1) = ppflags;

dlen -= maxlen;
```

```
  if ( *(dest+dlen-1) == ' ')
     { dlen --;  // otherwise may have terminal blank (if " ORDER BY")
     };
  return (dlen);  // return actual buffer length
}
```

13.11 Actual SELECT routine

13.11.1 SQL3SELECT

The principal function, performs an SQL SELECT query. We use cQUERY and dbLINK structures to implement a fairly efficient query.

Given a standard SQL SELECT statement, 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:

1. No spaces in comma lists

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

The “SELECT ” at the start has already been clipped off selectstring in the following. If onlyone is 1, then just the first valid row is returned.

We have deliberately avoided newer style join syntax, mainly because almost any database understands the old clumsy syntax, but many databases do their own thing with the newer style syntax.[12]

The program takes a library refnum, the SQL string and its length, whether only one result needs to be returned, the stack and stackstring (ugh) and a callback comparison function called ‘janet’. Results are placed on the stack (with string extensions on the stack string), and success is indicated in the return variable ‘ok’.

```
Int16 SQL3SELECT (UInt16 refnum, Char * selectstring,
                  Int16 selen, Int16 onlyone,
                  Char * STACK,
                  Char * STACKSTRING, DmComparF * janet)
{
  Int16 skL;
  Int16 ok = 0;  // default is 'failed' (0)
  Int16 RETCOLS; // number of columns that will be returned!
  Char * prebuf;
  Int16 prelen;
  Int16 preflags;
  Int16 sortconfiglen=0; // keep compiler happy!
  Char * seekTemp;
  Int32 sortsequence;

  skL = selen+64;
```

We perform some simple pre-processing before we translate to intermediate format. We create a ’pre-buffer’ somewhat bigger than the length of the query string.

```
 +OPTIONAL
   ConAsc(refnum, "\n: ", fDEBUG_SQL);
   ConTx(refnum, selectstring, selen, fDEBUG_SQL);
 -OPTIONAL

  prebuf = xNew2(refnum, skL, 0x10F);  // preformatting area
  xFill (prebuf, skL, 0x0);
     // ??? for debugging purposes, fill with hex zeroes.
  prelen = Preformat (refnum, prebuf, skL, selectstring, selen);
  if (prelen < 0)
```

[12] We might even write scripts to coerce new syntax into old — very retro!

```
     { SayErr(refnum,ErPreformatFailed);
       Delete2(refnum, prebuf);
       return 0;
     };
 +OPTIONAL
   ConAsc(refnum, "\n ->(", fDEBUG_SQL);
   ConTx(refnum, prebuf, prelen, fDEBUG_SQL);
   ConAsc(refnum, ")", fDEBUG_SQL);
 -OPTIONAL
```

In the following few lines we mark the stack, setting things up for post-processing (if needed). Note that the prebuf buffer contains a 16-byte header section.

```
preflags = *(prebuf+1);
if (preflags)
   {
     ok = Bypass (refnum, iMARK, 0, 0, 0, STACK, STACKSTRING); // all require mark
     //                              ^mark depth is zero.
     if (ok < 0)
        { SayErr(refnum,ErStackMarkFailed);
        };
     sortconfiglen = *(prebuf+2);
   };
prelen -= 16; // for leading buffer
```

We've finished pre-processing, so let's translate to intermediate format (IF).

```
  seekTemp = xNew2(refnum, skL, 0x110);  // arbitrary length ?? if ++ floats
  xFill (seekTemp, skL, 0x0);
     // ??? for debugging purposes, fill with hex zeroes.
  skL = SQxlate (refnum, seekTemp, skL,
                 prebuf+16, prelen, &RETCOLS);
        // Xlate to intermediate format.
```

SQxlate returns zero on failure. Next we perform the actual query using SeekWrap (Section 13.9.1. The value returned by SeekWrap into ok signals the outcome: 0 is an error, 1 is 'nothing retrieved', and a value of two or more signals success with one or more items on the stack.

```
  if (skL)
     {
       +OPTIONAL
        ConAsc(refnum, "\n ->>(", fDEBUG_SQL);
        ConTx(refnum, seekTemp, skL, fDEBUG_SQL);
        ConAsc(refnum, ")", fDEBUG_SQL);
       -OPTIONAL
```

```
      ok = SeekWrap(refnum, seekTemp, skL,
      onlyone, STACK, STACKSTRING, janet);
    } else
    { SayErr(refnum,ErQueryFailed);
      Delete2(refnum, seekTemp);
      Delete2(refnum, prebuf);
      return 0;
    };
  Delete2(refnum, seekTemp); // could check for success of Delete. Hm.
```

Post-processing is at present clumsy, as we imported our Bypass routine, which should be eradicated after replacing it with direct function calls [fix me]. We only perform post-processing if preflags is nonzero *and* a result was returned.[13]

```
if (preflags)
   { if (ok > 1)
     { ok  = -1; // default is 'failed' until proved otherwise!

   +OPTIONAL
   ConAsc(refnum, " FLAGS(", fDEBUG_SQK); //
   ConI(refnum, preflags, fDEBUG_SQK);
   ConAsc(refnum, ")", fDEBUG_SQK); //
   -OPTIONAL

       // IN ALL OF THE FOLLOWING the return code in ok is:
       //   < 0 : error
       //     0 : no item encountered but not an 'error'
       //     1 : success
       if (preflags == fMAX)
          { ok = Bypass (refnum, iMAX, 0, 0, 0, STACK, STACKSTRING);
          } else
          { if (preflags == fMIN)
               { ok = Bypass (refnum, iMIN, 0, 0, 0, STACK, STACKSTRING);
               } else
               { if (preflags & fDISTINCT) // if distinct
                    { ok = Bypass (refnum, iDISTINCT, RETCOLS,0,0,STACK,STACKSTRIN
                    }; // [does it make sense to invoke distinct before sort? --hm
                 if (preflags & fSORT)    // sort and distinct may combine!
                    { sortsequence =(Int32) ( *((UInt8*)(prebuf+6)) ); // hack hac
                      ok = DoSort (refnum, STACK, STACKSTRING,
                           RETCOLS, sortsequence);
                       // Bypass (refnum, iSORT, RETCOLS,
                       // prebuf+3, sortconfiglen, STACK, STACKSTRING); // [fix me
          };   };   };
       // ultimately sortsequence will be big-endian 4 bytes at prebuf[3..6]!
```

[13] Examine this routine: if 'ok' value of over 2 indicated two or more return items, could also skip post-processing on one item.

```
      if (ok < 0)
         { SayErr(refnum,ErPostProcessFailed);
           ok = 0;
         } else
         { ok ++;  // convert to usual success flags: 0=failed 1=nothing 2(+)=suc
         };
      // in keeping with rather silly general convention for SQL3SELECT,
      // we return:
      //    0 = error
      //    1 = nothing found
      //    2(+) = at least one item found
    };
  Bypass (refnum, iUNMARKONLY, 0, 0, 0, STACK, STACKSTRING); // all were marked.
  };
```

In the final statement above we clear the mark but preserve the current stack top, *not* taking things back to the mark!

The above needs some work. In particular, we need to be able to sort on more than one column, and have ‘bystander’ columns associated with but not used in the sort.

```
  Delete2(refnum, prebuf); // NOW can delete prebuf
  return ok;
};
```

14 SQL UPDATE statement

14.1 Translate WHERE

This section is REDUNDANT, is only used by UPDATE, and needs to be removed.

```
// XPackConditions. Given the meaty WHERE clause of a SELECT or UPDATE
// statement, translate to *intermediate format*.
// We are really interested in AND OR NOT and parentheses. This is messy.
// Our ideal approach is to:
// (a) break into sub-clauses according to parentheses, being careful not to
//     mis-identify parentheses in 'quoted '' (strings' as significant;
// (b) further break up, using " AND " as separator
// (c) further refine using " OR "
// A STRONG CASE CAN BE MADE FOR STUFFING THE INTERMEDIATE FORMAT, AND
// TRANSLATING DIRECTLY!
// WHERE "NOT((col1 > 5 OR col2 = '('')')AND NOT(COL2 <= 3 OR NOT COL3 <> 'fred )
// Things are made even more messy because we DON'T want to use recursion (stack c

// We establish the following rules:
// (i)   the only valid logical connectors (we'll call these *VLCs*) are the strin
//       " AND ", " OR ", and " NOT ". (It's easy to preprocess constructions
//       like ")AND(" into this format).
// (ii)  We *abandon* the usual convention that AND takes precedence over OR.
//       (this can simply be re-introduced by preformatting-in parentheses).
//       Parsing is strictly from left to right *as if* AND, OR and NOT had equiva
//       value. So if we encounter A AND B OR C OR D, processing is as follows:
//       stack up results of A B C D in order, then apply the logic OR,OR,AND
//       to the results ie. (C OR D)->X, (X OR B)->Y, and finally Y AND A.
// (iii) We introduce the idea of an OPERATOR, which is a VLC attached to a
//       CONDITION like "col1 > 5".
// (iv)  we FORBID unnecessary spaces, for example, those between parentheses,
//       so "( (" is invalid outside a quoted string, as are "  AND " as well
//       as "a   =   b", and "a OR (b AND a = 'xx' )" which is invalid only
//       because of the space after the quoted string 'xx'!
// (v)  We disambiguate " NOT (A OR B)" and " NOT A OR B" by insisting that the la
//      be written as either "( NOT A) OR B" or as " NOT (A) OR B".
// You can see that with this approach, we can process from left to right as follo

// (0) create a logical stack on which to store ANDs and ORs;
// (1) move from left to right
// (2) [keep track of parenthesis depth if you wish]
// (3)  store all text up to the first VLC, *excluding parentheses*
// (4) push that VLC to the stack
// (5) when you encounter a right parenthesis, pop and store a VLC as an OPERATOR
//      together with the preceding text as yet unstored!
// (6) continue until finished
```

```
// (7) At the end, pop all the VLCs (one after the other) and store these as OPERA

// For example, we are saying that
// "a AND b OR c" translates to RPN "a b c OR AND" and that
// "(a AND b) OR c" becomes the RPN-evaluated "a b AND c OR".
// There is a tiny wrinkle to this translation of infix notation to RPN ---
// because we will store each item A, B or C as a node, and associate zero or
// more logical operations with each node, we need to 'break up' the RPN string
// and store the logical operations with the nodes. In the examples above we
// thus say:
// "a AND b OR c" ==> "a b c OR AND" ==> a(S) b(S) c(OA)
// where S means 'simply store this', O means OR, and A means AND.
// similarly:
// "(a AND b) OR c" ==> "a b AND c OR" ==> a(S) b(A) c(O)
// We simply apportion any floating logic to the node to its immediate left.
// If there is no logical operator to apply, we simply say 'store the blighter'.
//
// An implementation detail is that when we write our intermediate format string,
// we write the logical operators before the condition, as Sa rather than a(S),
// and similarly we write OAc rather than c(OA).

Int16 XPackConditions (UInt16 refnum, Char * dest, Int16 destmax, Char * sqsrc, In
{  // [REDUNDANT: fix me!]
  Int16 ok;
  ok = PackConditions(refnum, dest, destmax, sqsrc, srclen);
  if (ok <= 0)
     { //ConAsc(refnum, "?E11", ZERO); //
       SayErr(refnum,ErFailedPackConditions);
       return 0;
     };
  return ok;
}
```

14.2 Adjust offset pointers &c.

Row adjustment. This is tricky. The AdjustRow routine accepts the usual library reference number and:

1. The row (in a buffer) with
2. its actual size, as well as
3. the maximum size of the buffer (top). This must be big enough to accommodate the initial size *plus* the adjustment;
4. the number of columns;

5. `colidx`, the *index* of the column to be adjusted;

6. The magnitude of the adjustment.

There are certain important assumptions. The most important is that the buffer contains a normally structured database row. This implies that the pointers contained point sequentially to row components, and that each component follows after the next one. The *offsets* of the various components start in the buffer itself at +0x10 relative to the start of the buffer. Each offset occupies two bytes. A `colidx` of zero is not permissible, as this always represents the special case where we are referring to the primary key of the row, which cannot be updated by this routine!

The routine does the following:

1. Clip/inserts the required number of bytes. If inserting, the new data bytes are *not* written (as they aren't even provided to the routine) — space is just made for them;

2. Within the header, adjusts all pointers that point *above* the offset, so that they correctly reflect the adjustment;

3. Adjusts the header reference to the buffer size too;

4. **Returns** the new buffer size.

```
Int16 AdjustRow (UInt16 refnum, Char * BUFROW, Int16 bufsize, Int16 bufmax,
                         Int16 numcols,
                         Int16 colidx, Int16 adjustment)
{
  Char * dP;
  Char * sP;
  Int16 len;
  Int16 coloff;

  if (! adjustment)
      { return bufsize;                          // trivial: no change
      };
  if ((colidx < 1) || (colidx > numcols))
     { //ConAsc(refnum, "?E15", ZERO); //
       SayErr(refnum,ErBadColumnIndex);
       return 0; // fail
     };
  if (bufsize + adjustment > bufmax)             // if new space won't fit
     { //ConAsc(refnum, "?E12", ZERO); //
       SayErr(refnum,ErWontFit);
```

```
      return 0;                                  // fail
    };

coloff = ReadInt16X(BUFROW+0x10+2*colidx);
```

After some initial checks for silliness, we get the actual offset of the desired column. We then check that this makes sense:

```
if ( -(adjustment) > bufsize-coloff) // more than is present?
   { //ConAsc(refnum, "?E13", ZERO);
     SayErr(refnum,ErBadColOffst);
     return 0;
   };
if (coloff > bufsize) // already defective?
   { //ConAsc(refnum, "?E14", ZERO);
     SayErr(refnum,ErDefectiveRecord);
     return 0;
   };
```

We next cater for both insertion (adjustment over zero) and deletion:

```
if (adjustment > 0)            // INSERT BYTES: +ve
   { sP = BUFROW + bufsize -1; // move to last current byte
     dP = sP + adjustment;     // new top
     len = bufsize - coloff;
     while (len)               // [ASM optimisation likely faster]
       { *dP-- = *sP--;        // transfer and decrement
         len --;               // count down
       };
   } else                      // CLIP OUT BYTES.
   { dP = BUFROW + coloff;
     sP = dP-adjustment;       // NB adjustment is -ve!
     len = bufsize - coloff + adjustment; // -ve.
     while (len)
       { *dP++ = *sP++;
         len --;
       };
   };
```

Finally, we update the pointers. We only update pointers which point to *after* the inserted or deleted item. Now what about pointers to null items? If the null item is before the current one (its index is lower) we do *not* update, otherwise we do:

```
dP = BUFROW + 0x10 + 2*(colidx+1);
numcols -= colidx;
while (numcols > 0)
```

```
    { AddInt16X(dP, adjustment);
      numcols --;
      dP += 2;
    }
  bufsize += adjustment;
  WriteInt16X(BUFROW+6, bufsize); // rewrite size!
  return (bufsize);               // new size.
};
```

14.3 UPDATE a row

Given database row in standard format, and linked list of cQUERY nodes describing columns to update and associated values, perform the update. (The format of our database rows is described in detail in the document PerlPgm.tex, under 'PDB Creation'). We repeat it here for convenience:

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

Table 2: Our header format

There is GOOD justification for ordering the cQUERY nodes by column number. This is especially true for multiple updates, as such ordering can profitably be used to go through and only copy data once, rather than multiple shifts of (potentially large) sections of data, as we do now.

There is also the potential for resizing the buffer dynamically here, if we hit the wall. We could even pass BUFROW by reference (ie Char** = ugly).

[!!!!!!!!! WE MUST ALSO CATER FOR ALTERATIONS TO THE KEY: IF SO, CHECK FOR VALIDITY. LATER MUST ALSO CHECK CONSTRAINTS !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!]

We accept the reference number required for debugging statements, the buffer row and a maximum size, as well as the cQUERY structure. We then read the column count; and the actual size from offset +6.

```
Int16 UpdateRow(UInt16 refnum, Char * BUFROW, Int16 maxsize,
                struct cQUERY * uptop)
{ struct dbLINK * db = uptop->dtable;
  Int16 numcols;
  Int16 bufsize;
  Int16 colnum;
  Int16 coloff;
  Int16 colsize;
  struct cQUERY * thisnode;

  Int16 uflags;
  uflags = ReadInt16X(BUFROW+0x4);
  uflags |= 0x02;
  WriteInt16X(BUFROW+0x4, uflags);
```

UPDATE should always set bit 1 of the flags to signal that an update has taken place. We do this here — read the flag word at offset +4 and then rewrite it to the buffer with the bit set.

```
  numcols = ReadInt16X((db->head)+0xE);
  thisnode = uptop;
  bufsize = ReadInt16X(BUFROW+0x6); // get *actual* size

  while (thisnode)
    { colnum = thisnode->col;       // might check validity??
      coloff = ReadInt16X(BUFROW+0x10+ 2*colnum);
      colsize = ReadInt16X(BUFROW+0x12+ 2*colnum) - coloff; // can be 0
      if (colsize != thisnode->cnstlen)
         { bufsize = AdjustRow (refnum, BUFROW, bufsize,
                     maxsize, numcols, colnum,
                     thisnode->cnstlen - colsize);
```

See the AdjustRow routine (Section 14.2 for details. It's noteworthy that we pass the index of the column to AdjustRow, and not the offset itself. If adjustment failed (a zero return) we signal this.

```
           if (! bufsize)   // failed
              { //ConAsc(refnum, "?L6", ZERO);
                SayErr(refnum,ErBufferFailed);
                return 0;
              };
        };
```

```
///      ConTx(refnum, "o=", 2, fDEBUG_SQJ); //
///      ConI(refnum, coloff, fDEBUG_SQJ);
      xCopy(BUFROW+coloff, thisnode->cnstant, thisnode->cnstlen); // fill in
      thisnode = thisnode->next;  // next item
    };
  return (bufsize);
};
```

14.4 The main UPDATE routine

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 G&P page 672.

We must also set a flag in each record we update. This will allow us to identify PROCESSes which have been updated on the PDA (as well as alterations to the PERSON table on death of a patient) when we export data back to the desktop program. The flag is bit #1 of the flags section of the row (not bit #0).

We *cannot* have tablename.columnname as a valid construct here, as the tablename is never explicit in SQL.

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. [FIX ME!]

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! [14] WHAT about "UPDATE table set key=key+1" ugly but powerful. Look at G&P example of this.

[14] 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.

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
SELECT itemlist FROM tablename
       WHERE arbitrary-condition-may-succeed-on-many-rows
```

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

In the following we ASSUME the “UPDATE ” string has been stripped off the front. One should give consideration to having a little ’spare space’ in the actual database row, to allow for activity of updates. This has pros and cons. If we do so, we must not let the spare space grow unchecked when large record is deleted. For now, we don’t implement this, allow underlying OS to worry. WE HAVE BUILT IN THIS *potential* CAPABILITY BECAUSE WE DO RECORD ACTUAL TOP of last record = phantom offset, AS WELL AS BYTE LENGTH OF ROW AT offset +6. We could allow 8 bytes for every null date, 12 per time, 20 per datestamp, 4/integer 8 per null float, and P per number, where P is precision of the number. The varchar is the most problematic,perhaps allow just SPARESPACE=16 or 32 spare bytes. The problem is large ’sparse arrays’, where we allocate a lot of unused space??

Now that we’ve introduced caching in some circumstances, we must also check whether we need to update records in the cache:

```
Int16 CheckCacheUpdate (UInt16 refnum, Char * tname, Int16 tlen,
  Char * dat, Int16 datlen,DmComparF * janet)
{
  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  if (! gl->CANCACHE)
     { return 0;
     };
  return CACHEUPDATE (gl->CACHELIB, tname, tlen, dat, datlen, janet);
}
```

In the following we pass the usual PalmOS library reference number, as well as the statement and its length, and a callback function for binary searching.

```
Int16 SQL3UPDATE (UInt16 refnum, Char * stmt, Int16 slen,
                  DmComparF * janet)
{
  Char * nameptr = stmt;
  Int16 namlen;
  struct dbLINK * ourtable;
  Int16 recs;
  Int16 KEEPMAX; // ?? max size for length of update!
  Int16 setlen;
  Char * setlist;  // setlist points to start of a=b,.. list
  struct cQUERY * thisnode = 0; // NB null to start
  struct cQUERY * uptop = 0;    // clumsy
  struct cQUERY * newnode;
  Int16 itmlen;                 // ??
  Char * iname;
  Int16 colnamlen;
  Int16 BUFLEN;
  Char * BUFROW;
  Char * packedP;
  Int16 plen;
  struct cQUERY * testnode;
  Int16 reclen = 0;   // clumsy
  Int16 rec = 1;      // first record is #1 (0 is header)
  Boolean dirty;
  Char * myrow;
  MemHandle myrec;
  Int16 ok = 1;
  Int16 datlen;
  Int16 newlen;
///  ConTx(refnum, stmt, slen, fDEBUG_SQJ); // [ha ha testing teehee ??]

  // more hacking:
  Char * tname;
  Int16 tnamlen;
```

First, we find the table.

```
  namlen = Advance(nameptr, slen, ' '); // find space after end of name
  if (! namlen)
     { //ConAsc(refnum, "?M9", ZERO); //
       SayErr(refnum,ErBadTableName);
       return 0;
     };
  stmt += namlen;
  slen -= namlen;
  namlen --;                                    // ignore space at end
  if (! xSame(stmt, 4, "SET ", 4))
```

```
    { //ConAsc(refnum, "?M10", ZERO); //
      SayErr(refnum,ErNoSetCmd);
      return 0;
    };
  stmt += 4;
  slen -= 4;

     tnamlen = namlen;
     tname = xNew(tnamlen+1); // for asciiz
     xCopy (tname, nameptr, tnamlen);
     *(tname+tnamlen) = 0x0;
     ourtable = CreateDbLink(refnum, tname, tnamlen, 0); // no index check.
     // the above 'tname' hack forces submission of ASCIIZ string to CreateDbLink
     // see that fx for explanation. nasty.

       // optimal to create dbLINK node. do this.
       // for now, don't create index on UPDATE. LATER 'fix' ???
       //
  if (! ourtable)
    { //ConAsc(refnum, "?M11", ZERO); //
      SayErr(refnum,ErBadTableUp);
      Delete (tname);
      return 0;
    };
  recs = MyCountRecords(refnum, ourtable);  // returns 1+number of rows
  if (! recs)
    { KillDbList(refnum, ourtable);  // fix 2008-03-10
      Delete (tname);
      return 0; // fail
    };
  if (recs < 2)
    { KillDbList(refnum, ourtable);  // fix 2008-03-10
      Delete (tname);
      return 1; // 'success', no records to change!
    };
```

Previously, we forgot to KillDbList, with unfortunate consequences applied to a null database!

Next, create a linked list for column=value. There are several errors and potential in the following. Despite only having one table, and the fact that it's illegal in SQL, we enforce[d] the tablename.columname convention in the WHERE clause [FIXED].

We have other silly restrictions — the allocation in the SET clause has *no* spaces around the equals sign, and confusingly enough, any condition such as '=' in the WHERE clause *must* have spaces on either side! [FIX ME]!

Our search for the WHERE clause doesn't take into account the possibility

that WHERE may be a string contained in some text somewhere, so we need to fix this too [FIX ME!]

We use cumbersome console writes to signal errors e.g. “M12”.

```
  setlist = stmt;
  setlen = Scan(stmt, slen, " WHERE ", 7); // ugly
    // setlen is offset of 1st char after " WHERE " within stmt.
  if (  (setlen < 7)
     )
     { //ConAsc(refnum, "?M12", ZERO);
       SayErr(refnum,ErBadWhere);
       KillDbList(refnum, ourtable);  // clean up
       Delete (tname);
       return 0;
     };
  stmt += setlen;  // move to after the WHERE
  slen -= setlen;  // track
  setlen -= 7;     // exclude the " WHERE " component
  KEEPMAX = setlen;// keep record of length of set stmt!?
  if (KEEPMAX < 20) { KEEPMAX = 20; };  // Yet Another Clumsy Hack.

  BUFLEN = KEEPMAX;
  BUFROW = xNew2(refnum, BUFLEN, 0x111);

  while (setlen > 0)
    { // here create linked list of cQUERY nodes, one per allocation C=V
      colnamlen = Advance(setlist, setlen, '=');
      if (colnamlen < 2)  // no name=value
         { //ConAsc(refnum, "?M13", ZERO); //
           SayErr(refnum,ErBadQueryNodes);
           Delete2(refnum, BUFROW);
           KillQuery (refnum, uptop);
           KillDbList (refnum, ourtable);
           Delete (tname);
           return 0; // no testnode yet.
         };
      iname = setlist;
      setlist += colnamlen;  // move past '='
      setlen -= colnamlen;   // track
      colnamlen --;          // ignore the '='
      newnode = FindColumn (refnum, ourtable, iname, colnamlen);
                // must be tablename.colname ?!
 // WE COULD establish the convention that if table name not specified, then
 // assume it's the first in the list, OR we could even check through all tables
 // for the bloody column, which is cumbersome and allows error with duplicate nam
      if (! newnode)
         { //ConAsc(refnum, "?M14", ZERO); //
```

```
          SayErr(refnum,ErNoNodeInUpdate);
          Delete2(refnum, BUFROW);
          KillQuery (refnum, uptop);                    // EXTREMELY CUMBERSOME
          KillDbList (refnum, ourtable);
          Delete (tname);
          return 0;
       };
     itmlen = Advance(setlist, setlen, ',');
              // look for start of next allocation
     if (! itmlen)          // if last allocation..
        { itmlen = setlen; // item is ALL of the rest
        } else
        { itmlen --;       // don't count the comma!
        };

     datlen = XFormat(refnum, BUFROW, BUFLEN, setlist, itmlen,
                      newnode->type, newnode->scale, newnode->len);
     // hmm, will BUFROW always be big enough? What about "t.f=1" float ???
     if (! datlen)
        { //ConAsc(refnum, "?M15", ZERO); //
          SayErr(refnum,ErBadFormatting);
          Delete2(refnum, BUFROW);
          KillQuery (refnum, uptop); // EXTREMELY CUMBERSOME
          KillDbList (refnum, ourtable);
          Delete (tname);
          return 0;
        };
     datlen --;                       // Xformat returns 1+length!
     newnode->cnstant = xNew2(refnum, datlen, 0x112);// NOT asciiz.
     newnode->cnstlen = datlen;       // clumsy.
     xCopy(newnode->cnstant,BUFROW,datlen);// copy in FORMATTED datum
     itmlen ++;             // move past comma OR terminal space!
     setlist += itmlen;
     setlen -= itmlen;      // at end setlen will be -1 !
     if (! thisnode)
        { uptop = newnode; // clumsy
        } else
        { thisnode->next = newnode;
        };
     thisnode = newnode;    // keep track of current node
   };                       // END WHILE setlen ..
```

Next, process the WHERE statement.

```
// (c) process the WHERE statement (stmt and slen define it)
packedP = xNew2(refnum, slen+64, 0x113);
```

```
// packed destination should be shorter [CHECK ME]!
plen = XPackConditions(refnum, packedP,
          slen, stmt, slen); // FIRST pack to intermediate format!
if (! plen)
   { Delete2(refnum, BUFROW);
     //ConAsc(refnum, "?M16", ZERO); //
     SayErr(refnum,ErBadUpdWhere);
     Delete (tname);
     return 0;
   };
testnode = MakeQueryList (refnum, packedP,
              plen, ourtable); // create search list
Delete2(refnum, packedP); // finished with this!
if (! testnode)
   { Delete2(refnum, BUFROW);                                    //
     KillQuery (refnum, uptop);
     KillDbList (refnum, ourtable);
     Delete (tname);
     return 0; // fail
   };
```

For each row, apply the condition and modify as required. We will need a buffer area in which to process the modified row. Rather arbitrarily, its length will be the length of the old row, PLUS KEEPMAX. Every time we encounter a row smaller than this, we clumsily re-allocate the buffer! [EXAMINE AND FIX ME]

In the following we open for read and write, so cannot use DmQueryRecord.

```
while (  (rec < recs) // for each record
       && ok          // while no error
      )
  { myrec = s_DmGetRecord(ourtable->odb, rec);  // find record
    myrow = (Char *)s_MemHandleLock(myrec);  // pointer to it
    ourtable->current = myrow;
    dirty = 0;
```

HERE we might conceivably test the flags of the row (for e.g. deleted)! Then if TestLogic succeeds, we apply the update – for each cQUERY node, replace the current value with the cnstant node value. [FIX ME: later we should re-engineer so that we can evaluate the answer, and put it into node-¿cnstant for each item: once we start doing this then KEEPMAX will become a severe problem].

```
    if (TestLogic (refnum, testnode, janet))
       { dirty = 1;
         reclen = ReadInt16X( myrow+6);
                  // get length of database row (max size)
         if (reclen + KEEPMAX > BUFLEN) // make an adequate buffer
```

```
            { Delete2(refnum, BUFROW);
              BUFLEN = reclen+KEEPMAX;  // new buffer size
              BUFROW = xNew2(refnum, BUFLEN, 0x114);
            };
          xCopy(BUFROW, myrow, reclen);
            // copy over whole row into temp buffer
          newlen = UpdateRow(refnum, BUFROW, BUFLEN, uptop);
```

Note that UpdateRow (Section 14.3) *must* retain the actual size in the header of our row. For the time we will always resize to the actual size, be it more or less. Aagh.[15]

In the following, failure of DmResizeRecord might potentially cause a fatal error, according to the PalmOS documentation. We must *not* release the handle (despite unlocking it) before we resize, we just do it. We then lock our pointer to the resized record.

```
          if (! newlen)
             { ok = 0; // fail
               //ConAsc(refnum, "?L5", ZERO);
               SayErr(refnum,ErFailRowUpd);
             } else
             { if (newlen != reclen)
                  { s_MemHandleUnlock(myrec);  // unlock
                    myrec = DmResizeRecord(ourtable->odb, rec, newlen); // resize
                    myrow = (Char *)s_MemHandleLock(myrec);  // resized
                  };
                if (! s_DmWrite( (void *) myrow, 0, BUFROW, newlen, newlen))
                     // 5th argument is a maximum (check) used elsewhere.
                  { ok = 0; // failed.
                  } else
                  { // here check whether we need to update cache row too!
                    CheckCacheUpdate (refnum, tname, tnamlen, BUFROW, newlen, jar
                  };
             };  // end 'if ! newlen' else..
        };       // END IF TestLogic..
     s_MemHandleUnlock(myrec); // unlock
     s_DmReleaseRecord(ourtable->odb,rec,dirty);
        // mark rewritten record as dirty.
     rec ++;
   };   // END WHILE rec < ..

Delete (tname);
Delete2(refnum, BUFROW);
KillQuery (refnum, testnode);
KillQuery (refnum, uptop);
```

[15] The underlying O/S will probably not be so parsimonious.

```
  KillDbList (refnum, ourtable);
  return ok;
}
```

15 The INSERT statement

The SQLinsert routine and all its acolytes has been moved from the main program to its logical place — here.

15.1 Preparation

First, the minor stuff . . .

15.1.1 Subsidiary comma location routine

Given a data string of maximum length, advance to next separating comma, *or* terminal right parenthesis. (We might simplify by preprocessing string to have final comma!) MUST account for SQL peculiarities:

1. ’ ’ is rendered as single quote
2. ignore comma ’contained. within quotes’
   Try e.g. `"’abc’’x’’bcde,f’,"`

Fortunately we can easily work this out:

```
Int16 FindNextComma(UInt16 refnum,
    Char * datum, Int16 maxlen)
{ Int16 quoting = 0;  // start as ’not quoting’
  Char ch=0;          // aagh keep compiler happy
  Char * dp = datum;

  while (maxlen > 0)
    { ch = *dp++;
      if (ch == 0x27) // 27 is quote mark
         { quoting = (! quoting);
         } else
         { if (  (! quoting)
               &&(ch == ’,’)
              )
              { return (Int16) (dp - datum);
                 // offset is 1 BYTE PAST the comma!
              };
         };
      maxlen --;
    };  // when ’while’ fails, at end:
  if (  (ch == ’)’)
      &&(! quoting)
     )
```

```
    { return (Int16) (dp - datum); // ? 1 after top!
    };
  return 0; // fail
}
```

Given the string ‘datum’ and its maximum length, move to the next true comma and return this offset, *plus one*.

15.1.2 Write formatted datum

Given the destination, column descriptor, and index data pull out the datum, format it, and write to destination. Returns ONE PLUS the length of datum copied, or zero if failed. [FLESH OUT DOCUMENTATION]

```
Int16 TransferFormatDatum( UInt16 refnum,
      Char * newdata, Int16 freesize,
      Char * cDescrip,
      Char *listIdx, Int16 cc,
      Char *  colnames, Char * startdata)
{
   Char * colname;
   Char coltype;
   Int16 colnamelen;
   Int16 colscale;
   Int16 maxcolsize;

   Char * testname;
   Int16 testlen;
   Int16 countdown = cc; // cc is number of columns to scan

   Char * mydatum;
   Int16 datlen;

   colname = cDescrip + ReadInt16X(cDescrip+0);
       // value at cDescrip will be 0x10, could check?
   colnamelen = ReadInt16X(cDescrip+2);
       // byte length of column
   maxcolsize = ReadInt16X(cDescrip+4);
       // maximum width of column item(bytes)
   coltype = *(cDescrip+6);  // single character type of column
   colscale = *(cDescrip+7); // and scale too!
   if (colnamelen < 1)
      { return 0; // *MUST* FAIL IF ZERO COLUMN LENGTH
      };

   while (countdown > 0)
     { testlen = ReadInt16X(listIdx+2);
```

```
        // listIdx has 8 bytes per column
      if (colnamelen == testlen) // don't bother unless lengths the same!
         { testname = colnames + ReadInt16X(listIdx+0);
           if (xSame2 (colname, testname, colnamelen))
              { countdown = 0;  // success, force exit with countdown=-1
              };
         };
      listIdx += 8;  // move to next column 'index' details
      countdown --;
    };

  if (! countdown) // failed if countdown not -1
    { //ERRmsg(ErSQLinsertNoColumn);
      //ERRSTRING(colname, colnamelen);
      //return 0;  // fail!
      // NO. if not found, force to null?!:
      mydatum = "NULL";
      datlen = 4;
    } else
    { listIdx -= 8; // back to this one
      WriteInt16X(listIdx+2,0);
         // SIGNAL THIS COLUMN WAS USED!
      mydatum = startdata + ReadInt16X(listIdx+4);
         // add offset of datum
      datlen = ReadInt16X(listIdx+6);
         // read its length [CAN BE ZERO?]
    };

  // the following function also checks for NULL
  // [??? ! LATER WILL ALSO REQUIRE TO INSERT DEFAULT VALUES ..]
  datlen = XFormat(refnum, newdata, freesize, mydatum, datlen, coltype, colscale,
                   maxcolsize);
          // XFormat likewise returns 1+ datum length!
  return datlen; // returns  ONE PLUS number of bytes transferred
} // +1 allows us to transfer zero bytes with success!
```

15.1.3 Make a record

First a PalmOS wrapper:

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

Next, our actual record creation.

```
Int16 MakeFileRecord (UInt16 refnum,
            DmOpenRef myDB, UInt16 idx,
            const Char* recdata, Int16 datalen)
{ MemHandle mynewrec;
  void * recPtr;
  UInt16* pIdx;
  pIdx = &idx; //point to index

  mynewrec = s_DmNewRecord(myDB, pIdx, datalen);
  if (! mynewrec)
     { // ERRmsg(ErFailNewRecord);
       return 0; //fail
     };
  recPtr = s_MemHandleLock(mynewrec); // could fail?
  if (! s_DmWrite(recPtr, 0, recdata, datalen, datalen))
     { // ERRmsg(ErFailWriteRecord);
       return 0;
     };
  if (! s_MemHandleUnlock(mynewrec))
       // do NOT use recPtr!
     { // ERRmsg(ErFailUnlockRecord);
       return 0;
     };
  if (! s_DmReleaseRecord(myDB, idx, true))
     { // ERRmsg(ErFailReleaseRecord);
       return 0;
     };
  return 1;  // success.
}
```

15.1.4 Insert data row

Given open database, locate correct spot and put in line!

```
Int16 DataRowInsert (UInt16 refnum,
         DmOpenRef pdb, char * dat, Int16 dlen, DmComparF * janet,
         Int32 * NEWKEY)
{
  Int16 sortpos;
  Int32 rowkey;
  MemHandle prev;
  Char * prevP;  // or MemPtr
  Int16 eql;

  *(NEWKEY) = 0; // byref. By default, clear.

  sortpos = s_DmFindSortPosition (pdb, (void *) dat, 0, janet, 0);
  if (! sortpos)
```

```
     { // ERRmsg(ErRowInsFailed);
       return 0;
     };

  rowkey = ReadInt32( dat+8 );  // ib 4 = ok.
  sortpos --;  // go to PRECEDING RECORD
  if (sortpos > 0) // if not inserting prior to top record!
     { prev = s_DmQueryRecord(pdb, sortpos);
       prevP = (Char *) s_MemHandleLock(prev);
       eql = (rowkey == ReadInt32(prevP+8));
             // 0 = no, 1=yes, key already exists!
       s_MemPtrUnlock(prevP);
       if (eql)
          { // ERRmsg(ErDuplicateKey);
            return 0;                              // fail.
          };
     }; // (if only prior record is header record, then must succeed)
  sortpos ++;                                      // restore.
  if ( ! MakeFileRecord (refnum, pdb, sortpos, dat, dlen) )
     { // ERRmsg(ErRowInsFailed2);
       return 0;
     };
  *(NEWKEY) = rowkey; // byref. return this value!
  return sortpos;  // success, AND key value!
}
```

15.2 Subsidiary INSERT routine

This is a meaty (and rather nasty) routine, and needs documentation, as well as perhaps breaking up into smaller subsidiary routines. The NEWKEY variable is a byref call, and a 32 bit key value is returned within it (ugh).

We also submit the actual table name and length of this name to SQLins, only because we require these in our caching invocation of CACHEINSERT.[16] So first, the cache insertion routine:

```
Int16 TestCache (UInt16 refnum, Char * tname, Int16 tlen,
   Char * dat, Int16 datlen, DmComparF * janet)
{
  // or might check for relevant files here (easier, quicker) ??

  SysLibTblEntryPtr entryP;
  Sql3Lib_globals *gl;
  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
```

[16] We might move this to the invoker, but all of the byref stuff (NEWKEY) etc is starting to (a)ppall.

```
  if (! gl->CANCACHE)
     { return tlen;  // default
     };
  return CACHEINSERT (gl->CACHELIB, tname, tlen,
                   dat, datlen, janet);
}
```

Here's SQLins itself.

```
Int16 SQLins (UInt16 refnum,
       Char * srcp, Int16 slen,
       DmOpenRef pdb, Char * hdrP,
       Char * listIdx, Char * newrow,
       Int16 newrowlen, DmComparF * janet,
       Int32 * NEWKEY,
       Char * tablename, Int16 tbllen)
{
  Int16 colcount;
  Char * colnames; // point to start of column names
  Int16 colnlen;          // length of column name list

  Char * startdata; // remember start of data.
  Int16 datlen;   // length of a datum

  Int16 cc = 0;   // actual column count
  Int16 namesize; // length of single column name
  Char * iP; // pointer to index list
  Int16 coloff = 0; // offset of next column name
  Int16 datoff = 0; // offset of current datum
  Int16 ns;
  Int16 ds;
  Int16 co;
  Int16 doff;

  Char * newdata; // data component of new row
  Int16 hdrsize;
  Int16 freesize; // keep track of free space ?!
  Char * newptr;  // points to pointer area in new row

  Int16 colsleft; // colsleft records remaining cols to process
  Char * coloffs;
  Char * cDescrip;

  Int16 xferred;
  Int16 rowlen;
  Int16 ok;

  colcount = ReadInt16X(hdrP+0x0E); // get col count
```

```
  colnames = srcp;

  // Index input list
  colnlen = Advance(colnames, slen, ')');
  if (colnlen < 1)
     { // ERRmsg(ErInsColsNoRpar);
       return 0;
     };
  srcp += colnlen; // move past column list in SOURCE
  slen -= colnlen; // track
  colnlen --;      // ignore right parenthesis itself
  if (!xSame2 (srcp, "VALUES(", 7)) // check keyword and lpar
     { // ERRmsg(ErInsNoVALUES);
       return 0;
     };
  srcp += 7;      // move to first datum
  slen -= 7;
  startdata = srcp; // remember start of data.
  iP = listIdx;

  while (  (colnlen > 0)
           && (cc < colcount) // cc >= colcount is ERROR!
        )
    {
      namesize = Advance(colnames+coloff,
                 colnlen, ','); // length up to next comma!
      if (! namesize)           // clumsy: no comma ???
         { namesize = 1+colnlen;// +1 for after terminus!
         };

// 27-2-2005: hack:
  ns = namesize-1;
  while ( *(colnames+coloff+ns) == ' ' )  // trim terminal blanks
        { ns --;
        }; // [fix me: mess if only blanks]
  co = coloff;
  while ( *(colnames+co) == ' ') // trim leading blanks
        { co ++;
        };
// end hack

 datlen = FindNextComma(refnum, startdata+datoff, slen);
          // find datum length (complex)

// 27-2-2005: hack:
  ds = datlen-1;
  while ( *(startdata+datoff+ds) == ' ' )  // trim terminal blanks
        { ds --;
```

```
       }; // [fix me: mess if only blanks]
  doff = datoff;
  while ( *(startdata+doff) == ' ')          // trim leading blanks
       { doff ++;
       };
// end hack

 WriteInt16X(iP,co);  // ? neednt be bigendian (slower)
 WriteInt16X(iP+4,doff); // datum offset from startdata
 WriteInt16X(iP+6,ds); // datlen is smarter than namesize (??)
      if (namesize) // NOT last item:
         { WriteInt16X(iP+2,ns); // actual size less terminal blanks
           iP += 8;   // move on
           colnlen -= namesize;
           coloff += namesize;
           datoff += datlen;
           slen -= datlen; // keep track.
         } else            // IS last item.
         { WriteInt16X(iP+2,colnlen);
              // terminal portion [??? check me?]
           colnlen = 0; // force exit
         };
      cc ++; // bump count for this column
    };
  if (colnlen > 0) // error if remaining colname data:
     { // ERRmsg(ErInsRemainingColData);
       return 0;
     };

  // WORK THROUGH HEADERS:
  hdrsize = 0x10 + 2*(colcount+1);
   // new row header area = (0x10 + pointer area)
  freesize = newrowlen - hdrsize;
   // keep track of free space ?!
  newptr = newrow + 0x10;
   // points to pointer area in new row

  // the following refer to the existing database header:
  colsleft = colcount; // colsleft records remaining cols to process
  coloffs = hdrP + 0x10; // move to offset of column

  // FIRST, write the key. This is always first column.
  //   Result goes into header (@+8)
  cDescrip = hdrP + ReadInt16X(coloffs);
     // point to first(key) column
  coloffs += 2; // bump source
  colsleft --;
  newdata = newrow + 8; // will write key to header offset +8 !
```

```
if (! TransferFormatDatum(refnum, newdata, 4,
        cDescrip, listIdx, cc,
        colnames, startdata)) // free size forced to 4
   { // ERRmsg(ErInsBadKey);
     return 0;
   };
WriteInt16X(newptr, 8); // offset is +8 in header!
newptr += 2; // still have pointer to this!

// NEXT, write remaining data
newdata = newrow + hdrsize; // rest of data will sequentially go here.
while (colsleft > 0)
  { // go to first column, fill in datum, et seq.
    // (1) get offset of column descriptor
    cDescrip = hdrP + ReadInt16X(coloffs);
        // point cDescrip to column descriptor
    coloffs += 2; // bump
    xferred = TransferFormatDatum(refnum, newdata, freesize,
                cDescrip, listIdx, cc,
                colnames, startdata);
    // find column, extract data, format, and insert!
    // Also clears *length* of used datum!
    // returns 1+number of bytes transferred!
    if (! xferred)
       { // ERRmsg(ErBadInsDatum);
         return 0;
       };
    xferred --;  // compensate for +1
    WriteInt16X(newptr,
                (newdata - newrow));  // write offset of datum
    newptr += 2;
    newdata += xferred;
    freesize -= xferred;
    colsleft --;
  };  // here should check if failed (?!)
rowlen = (newdata - newrow);
WriteInt16X(newptr, rowlen); // write phantom offset.

// Check for extraneous columns!
iP = listIdx + 2; // move to first name *length*
while (cc > 0)
  { namesize = ReadInt16X(iP); // get length
    if (namesize) // if non-zero MUST BE ERROR!
       { // ERRmsg(ErInsExtraCol);
         return 0;
       };
    iP += 8; // move to next 'name length'
    cc --;
```

```
    };

  // Fix up header. [WHAT ABOUT having an EARLIER KEY CHECK ???]
  newptr = newrow;
  WriteInt32(newptr+0, 0); // no CRC, for now
  WriteInt16X(newptr+4,1); // flags. 01-->no CRC
  WriteInt16X(newptr+6, rowlen); // size of row
  // KEY IS ALREADY AT +8.
  WriteInt32(newptr+0xC,0); // all zeroes

  // Write row to database.
  ok = DataRowInsert(refnum, pdb, newrow, rowlen, janet, NEWKEY);
  if (! ok)
     { return 0;
     };
  // NB return value is sort position in Palmos database, or
  // zero on failure.
  // HERE UPDATE CACHE, IF APPROPRIATE:
  TestCache (refnum, tablename, tbllen, newrow, rowlen, janet);

  return ok;
}
```

15.2.1 Open PalmOS ‘file’ — another subsidiary

First we need the following *absolutely horrendous* function, which creates a temporary CRUTCH2 buffer, copies the name to it, and turns the name into an ASCIIZ string, a la PalmOS:

```
LocalID a_DmFindDatabase(UInt16 refnum, Char * nameP, Int16 nlen)
{
  Char * CRUTCH3;
  LocalID lid;

  CRUTCH3 = xNew2(refnum, nlen+1, 0x115);
  xCopy (CRUTCH3, nameP, nlen);
  *(CRUTCH3+nlen) = 0x0;
  lid = s_DmFindDatabase(CRUTCH3);
  Delete2(refnum, CRUTCH3);
  return (lid);
}
```

Next, locate and open a PalmOS database.

```
DmOpenRef PalmFileOpen (UInt16 refnum, Char *databaseName, Int16 namlen)
{ LocalID mydbid;
```

```
  DmOpenRef mydbref;
  mydbid = a_DmFindDatabase(refnum, databaseName, namlen);
  if (! mydbid)
     { //
       //ERRmsg(ErPalmDbNotFound);
       //ERRSTRING(databaseName, namlen); // [??]
       return 0; //fail
     };
  mydbref = s_DmOpenDatabase (refnum, mydbid, dmModeReadWrite);
  if (! mydbref)
     { // ERRmsg(ErPalmDbNotOpen);
       return 0; //fail
     };
  return mydbref;
}
```

15.3 Main Insert routine

This function is a wrapper for the SQLins function. In the following we assume that 'INSERT INTO' and the subsequent space have already been stripped off the front of the string to be parsed. The rest of the INSERT statement is contained in the srcp pointer, with length specified in slen. The tablename should start this string and be immediately followed by a left parenthesis, with no white space between the two.

Owing to future design planning, we should limit the length of a table name to 15 or fewer characters [FIX ME].

We cannot escape the need to systematically examine each column (via its header) and locate the corresponding new datum, then inserting that datum into the target row in its correct position.

We thus:

1. Locate database, open header (done already);
2. Set aside memory for new row, and format its header etc according to number of columns! (blank pointers?);
3. Index input list of columns to allow easy access and comparison, at the same time indexing the data provided;
4. Work through each header column in turn, identifying input column where present, otherwise inserting either null or default; (We might even check first (key) is unique and not null NOW, and even find insertion position!);
5. Check that there aren't extraneous columns left in input list (fail then);

6. Fixup header, and (g) write record in correct place and exit.

As Described in MyCountRecords (*Sql3Lib.tex*) and MakeOnePDB (*PerlPgm.tex*), we keep count of the number of rows in each database at offset +8 within our header record. SQLinsert must thus keep tally, incrementing this value by one when successfully inserting a row.

```
Int16 SQL3INSERT (UInt16 refnum,
                  Char * srcp, Int16 slen, DmComparF * janet)
{
  Char * tablename; // find data table
  Int16 tbllen;
  Int32 NEWKEY;
  Int16 newpos;

  DmOpenRef pdb;
  MemHandle hdr;
  Char * hdrP; // or MemPtr
  Char * listIdx;
  Char * newrow;
  Int16 newrowlen;
  Int32 rowcount;

  UInt16 IDXLIB;
  Int16 ok=1;  // default is 'ok'.


  if (* (srcp+slen-1) == ';')
     { slen --;
     }; // allow but ignore terminal semicolon

  tablename = srcp; // find data table
  tbllen = Advance(tablename, slen, '(');
  if (! tbllen) // ? might check length <= 15
     { // ERRmsg(ErInsertNoTblname);
       return 0; // fail
     };
  srcp += tbllen; // go past parenthesis
  slen -= tbllen; // track length
  tbllen --;      // ignore parenthesis

  // here we make sure that the table has the relevant index(es)
  FixIndex(refnum, tablename, tbllen);

  // resuming:
  pdb = PalmFileOpen(refnum, tablename,tbllen);
  if (! pdb)
      { // ERRmsg(ErNoInsertTbl);
```

```
      return 0;
    };
hdr = s_DmGetRecord(pdb, 0); // NOT DmQueryRecord !
hdrP = (Char *) s_MemHandleLock(hdr);
if (! hdrP)
   { // ERRmsg(ErInsertNoHdrAccess);
     PalmFileClose(pdb); // clumsy rtn
     return 0;
   };

listIdx = xNew2(refnum, 8*MAXCOLUMNS, 0x116);
  // 2 lengths, 2 offsets, each 2 bytes
newrowlen = 0x10 + 2*MAXCOLUMNS + slen;
newrow = xNew2(refnum, newrowlen, 0x117); // [? size]

newpos = SQLins(refnum, srcp, slen,
            pdb, hdrP,
            listIdx, newrow,
            newrowlen, janet, &NEWKEY,
            tablename, tbllen);
if (! newpos)
   {ok = 0; // fail
   };
```

It's now time to update the row count stored at offset +8 in *our* header record. Subtracting one from the negative value has the same effect as incrementing the positive value.

```
rowcount = -1 + (ReadInt32(hdrP+8));
if (! s_DmWrite(hdrP, 8, &rowcount, 4, 12)) // rewrite!
   { ok = 0; // might use more specific error
   };


Delete2 (refnum, listIdx);
Delete2 (refnum, newrow); // might check these
if (! s_MemHandleUnlock(hdr))
   { ok = 0;
   };
if (! s_DmReleaseRecord(pdb, 0, true))
   { ok = 0;
   };
if (! PalmFileClose(pdb))
   { ok = 0;
   };  // clumsy
```

Because we need to rewrite our header we opened the header record (0) using DmGetRecord and not DmQueryRecord, with a corresponding unlock and release.

Finally, we can update the index (if required):

```
  if (ok && NEWKEY) // either 0 means 'failed'
     { IDXLIB = GetIndexRef(refnum);
       if (IDXLIB) // unless disabled
          { ok = NEWINDEXITEM ( IDXLIB, tablename, tbllen,
                    1, newpos, (Char *) &NEWKEY, 4, 0);
            if (ok < 0) // if failure (aagh!)
               { TurnOffIndexing(IDXLIB);
                 ConAsc(refnum, "[? idx2 ", 0); // err msg
                 ConTx (refnum, tablename, tbllen, 0);
                 ConI  (refnum, ok, 0);
                 ConAsc(refnum, "]", 0);
               };
          };
          // column 1 signals 'primary key'.
     };

  return ok;
}
```

And that's that, really!

16 Header file: SQL3.h

We start with a few simple defines:

```
#ifndef SQL3EXTRA_H
#define SQL3EXTRA_H

#include <LibTraps.h>
#include <FloatMgr.h>

#ifndef SQL3_TRAP
#define SQL3_TRAP(trapno)  SYS_TRAP(trapno)
#endif

#define sysLibTrapSQL3SELECT    sysLibTrapCustom+0
#define sysLibTrapSQL3UPDATE    sysLibTrapCustom+1
#define sysLibTrapSQL3PASSCON   sysLibTrapCustom+2
#define sysLibTrapSQL3PASSBUG   sysLibTrapCustom+3
#define sysLibTrapSQL3INSERT    sysLibTrapCustom+4
#define sysLibTrapSQL3UPTURN    sysLibTrapCustom+5
```

The actual routines, as seen externally.

```
Err SQL3Open (UInt16 refNum)
    SQL3_TRAP(sysLibTrapOpen);

Err SQL3Close (UInt16 refNum, UInt16 *numappsP)
    SQL3_TRAP(sysLibTrapClose);

Int16 SQL3SELECT (UInt16 refnum, Char * selectstring,
                  Int16 selen, Int16 onlyone,
                  Char * STACK,
                 Char * STACKSTRING, DmComparF * janet)
    SQL3_TRAP(sysLibTrapSQL3SELECT);

Int16 SQL3UPDATE (UInt16 refnum, Char * stmt, Int16 slen,
                   DmComparF * janet)
    SQL3_TRAP(sysLibTrapSQL3UPDATE);

Int16 PassConsole (UInt16 refnum, UInt16 cons, UInt16 errlib,
                   UInt16 idxlib, UInt16 cachelib )
    SQL3_TRAP(sysLibTrapSQL3PASSCON);

Int16 PassBug (UInt16 refnum, UInt16 bugs)
    SQL3_TRAP(sysLibTrapSQL3PASSBUG);

Int16 SQL3INSERT (UInt16 refnum,
                 Char * srcp, Int16 slen, DmComparF * janet)
```

```
    SQL3_TRAP(sysLibTrapSQL3INSERT);

Int16 SQL3UPTURN (UInt16 refnum, Int16 turn)
    SQL3_TRAP(sysLibTrapSQL3UPTURN);
```

Some constants, including error constants.

```
#define fSORT     1
#define fMAX      2
#define fMIN      4
#define fDISTINCT 8
// above are pre-processor flags


#define MAXCBUF 256
  // size of a condition-storing buffer used in WHERE statement processing


#define  iSORT       43
#define  iDISTINCT   44
#define  iMAX        45
#define  iMIN        46
#define  iMARK       47
#define  iUNMARK     48
#define  iUNMARKONLY 50


#define SqErMarkEmpty              173
#define SqErMarkArg                174
#define SqErMarkMax                239
#define SqErMarkMin                240
#define SqErNoMax                  241
#define SqErNoMin                  243
#define SqErNoDistinct             244
#define SqErBadDistinct            245
#define SqErBadSortItems           246
#define SqErNoSort                 247
#define SqErBadSortMode            248
#define SqErSortMajor              250
#define SqErSortCorruptStack       251

#define SqErBypass                 252
#define SqErSQLsortorder           253
#define SqErFullSQL                254
#define SqErSQLimbalance           255
#define SqErBadParens              256

#define SqErSQLBadCondition        199
```

```
#define SqErSQLComparator2          202
#define SqErSQLComparator3          203
#define SqErSQLComparatorStop       204
#define SqErSQLNoCfSpace            205
#define SqErImbalancedPars          206
#define SqErBadConditional          207
#define SqErBadConditional2         208
#define SqErLogicOverflow           209
#define SqErWhileOverflow           210
#define SqErFewParentheses          211
#define SqErQuoteImbalance          212
#define SqErBadLogic                213
#define SqErStoreCondition          214
#define SqErBadTypeCompare          242
#define  SqErInsortFailed           224


#define SqErTextComparisonMode      194
#define SqErIntComparisonMode       195
#define SqErWarnSillyEqual          196
#define SqErWarnNotEqual            197




#define SqErPushNoStack              11
#define SqErPushBadLength            12
#define SqErPushNothing              13
#define SqErPushStackFull            14
#define SqErPushFailed               15
#define SqErLongPushFailed           17
#define SqErPushLong                 20

#endif
```

The end of the header file.

17 The Makefile

This is relatively straightforward, similar to other library makefiles such as that for the main scripting library.

```
LIBPATH = c:/palmdev/sdk-4
CREATOR = JxVS
LIBPATH = c:/palmdev/sdk-4
VERSION = 1

CC = m68k-palmos-gcc -Wall -g -O2 -mdebug-labels
AS = m68k-palmos-as

all: SQL3-syslib.prc

SQL3-syslib.prc: SQL3.def SQL3
build-prc -o $@ SQL3.def SQL3
ls -l *.prc

SQL3_objs = SQL3.o SQL3-dispatch.o

SQL3: $(SQL3_objs) Makefile
$(CC) -shared -nostartfiles -nostdlib -o $@ $(SQL3_objs) -lnfm -lgcc
m68k-palmos-objdump --section-headers SQL3

SQL3.o: SQL3.c SQL3.h

SQL3-dispatch.o: SQL3-dispatch.s

SQL3-dispatch.s: SQL3.def
m68k-palmos-stubgen SQL3.def

clean:
rm -f *.o *.prc *-dispatch.? SQL3
```

Appending the term ‘ -mdebug-labels’ to CC in the above makefile inserts names into the final code, permitting debugging and (notably) profiling.

18 The DEF file: SQL3.def

```
syslib { "SQL3 Library" 4sql }

export {
  SQL3Open SQL3Close nothing nothing
  SQL3SELECT SQL3UPDATE PassConsole PassBug SQL3INSERT SQL3UPTURN
  }
```

19 Change Log

From version 0.95, we introduce a change log.

19.1 Version 0.95