Analgesia Database:
Scripting Library for PalmOS PDA

Version 0.95

J.M. van Schalkwyk

February 27, 2009

Contents

1 The C file: SCRIPTING.c

This library handles all scripting routines apart from those which are numeric (For these, see *NumericLib.tex*).

1.1 OVERVIEW

There are several 'logical' sections in this shared library:

1. Basic functions for library entry, exit — most of these are stubs
2. A set of utility functions used elsewhere
3. Initialisation
4. Stack push and pop functions; also marking
5. Translation routines
6. Data encoding.
7. String intercalation (complex)
8. The Resolve function
9. Logical functions (and tests)
10. Arithmetic functions
11. Flow control and stack modification
12. String-handling
13. Pushing a number!
14. Miscellaneous functions including time
15. Actually interpreting a script instruction: DoScript

```
#include <SystemMgr.h>
#include <PalmOS.h>
#include "MathLib.h"
// for floor fx.

#define TOOBIG 1073741823
#define TOOSMALL -1073741824
```

```
#define MAXDIGITS 9
#define SHORTTIME_macro 1
#define EPSILON 1E-6

/* to look for fx and where they occur, simply "grep fxname *.h "
within: cd /PalmDev/sdk-4/include/Core/System
*/

#define SCRIPTING_TRAP(trapno)
#include "SCRIPTING.h"
#include "../palmsql3A.h"
#include "../console/CONSOLE.h"
#include "../err/ERRDEBUG.h"
```

As for e.g. Sql3Lib, we now permit console writes for debugging purposes. The `MenuYOffset` and `MenuMaxHt` values, as well as the menu screen width and height values (`MenuScrWidth` and `MenuScrHt`) are contained in *palmsql3A.h*, and are required because PalmOS includes the menu headers in the $160x160$ screen area — we must take this into account. See PixelY below.

the SHORTTIME value, if defined (which we now do) constrains us to use TIME values (including those in TIMESTAMPs) with no fractional seconds. This is acceptable under the SQL3 standard, makes sense, and results in substantial savings in database size.

```
/* ============================================================================
/* A. BASIC FUNCTIONS. */

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

The value of EPSILON is at present hard-coded, but ideally should be passed to the library. It is used in float to integer conversions to prevent truncation problems. In the following we implement the globals, creating an instance of the ScriptinLib_globals structure.

```
typedef struct {
  UInt16 CONSOLE;
  UInt16 ERRLIB;
  UInt16 DEBUGFLAGS;
  UInt16 refcount;
  UInt16 MATHLIB;
  } ScriptingLib_globals;
```

```
  // include count in expectation of multiple openings!

Err SCRIPTINGOpen (UInt16 refnum)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  ScriptingLib_globals *gl = entryP->globalsP;
  if (!gl)
     {
       gl = entryP->globalsP = MemPtrNew (sizeof (ScriptingLib_globals));
       MemPtrSetOwner (gl, 0); // note use of sys fxs here, above
       gl->CONSOLE = 0;
       gl->ERRLIB = 0;
       gl->DEBUGFLAGS = 0;
       gl->refcount = 0;
       gl->MATHLIB = 0;
     }
  gl->refcount ++;
  return 0;
  }

Err SCRIPTINGClose (UInt16 refnum, UInt16 *numappsP)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  ScriptingLib_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;
  }
```

Here follow rtns very similar to those in Sql3Lib:

```
Int16 PassConsoleScripting (UInt16 refnum, UInt16 cons, UInt16 errlib)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
```

```
  ScriptingLib_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; // AND error library ref
  return 0;
}

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

We also include a routine to accept the code for MathLib, which we need for some floating point routines:

```
Int16 ScriptMathLib (UInt16 refnum, UInt16 MathLibCODE)
{
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  ScriptingLib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return 1; // fail
  };
  gl->MATHLIB = MathLibCODE;
  return 0;
}
```

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

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

  entryP = SysLibTblEntry (refnum);
```

```
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  dbg = gl->DEBUGFLAGS;

  if (bugflag) // 0 forces write!
     {
       if (! (dbg & bugflag))
          { return;
          };
     };

  if (! CONSOLE)
     { return;  // fail
     };
   ConWrite(CONSOLE, txt, txlen);
}
```

The *xNew* function reserves memory:

```
Char * xNew ( Int16 memsize)
{ MemHandle memH;
  MemPtr memP;
  memH = MemHandleNew(memsize);
  if (! memH)
     { return 0;
     };
  memP = MemHandleLock(memH);
  if (! memP)
     { return 0;
     };
  return ((Char *) memP);
}
```

Similar is xNew2, but here we supply an additional refnum parameter which allows us to write a reminder using ErrorWrite.[1] First we need the routine WriteErr, which invokes ErrorWrite.

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

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

[1] We might even write special codes using ErrorWrite on failure too! [? fix me]

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

As usual, the *Delete* function is the converse of *xNew*

```
Int16 Delete (MemPtr memP)
{ if (! memP)
     { return 0; // fail
     };
  if (MemPtrUnlock (memP))   // nonzero = error
     { return 0;
     };
  if (MemPtrFree (memP))     // nonzero = error
     { return 0;
     };
  return 1;                  // clumsy
}
```

Here's Delete2, which undoes xNew2. Note that these must be used as a pair, and should never be used in concert with Delete or xNew.[2]

[2] As with New2, there is the potential to write other failure codes to the cyclical error buffer using WriteErr.

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

We might wrap the MemPtr functions.

1.1.1 SayErr

The following function was lifted from Sql3Lib.tex. It merely writes an error string to the console.

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

1.1.2 w_StrIToA: a wrapper

Another helper fx, which partially addresses the defects in *StrIToA*:

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

See the Sql3Lib.tex file for cognate functions and details of how they work. The following integer write is slooow and should only be used for debugging.

```
void WriteConsoleInteger (UInt16 refnum, Int32 i, UInt16 bugflag)
{ UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  ScriptingLib_globals *gl;
  UInt16 dbg;
  Int16 ilen;
  Char * CRUTCH;

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

  if (bugflag) // 0 forces write!
     {
       if (! (dbg & bugflag))
          { return;
          };
     };

  CRUTCH = xNew(maxStrIToALen+1);
  ilen = w_StrIToA (CRUTCH, maxStrIToALen+1, i); // sys fx.
  ConWrite (CONSOLE, CRUTCH, ilen);
  Delete(CRUTCH);
}
```

We might speed things up a little by having a scratch buffer within the globals.

```
void WriteDebug (UInt16 refnum, Char * asci, Char * txt, Int16 txlen)
{
  UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  ScriptingLib_globals *gl;
  UInt16 dbg;

  if (txlen <= 0)  // check for nonsense?!
```

```
     { return;     // [might extend this]
     };

  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  dbg = gl->DEBUGFLAGS;
  if (! (dbg & fDEBUG_STMT))
     { return;
     };
  if (! CONSOLE)
     { return;  // fail
     };
   ConWrite(CONSOLE, asci, StrLen(asci));
   ConWrite(CONSOLE, txt, txlen);
}
```

1.1.3 w_DmWrite

Write a given string to a record. Return 0 on failure, 1 on success. Similar to the routine in *wraps.cpp*.

```
Int16 w_DmWrite (void *recordP, UInt32 offset, const void *srcP, UInt32 bytes, UIn
{ Int16 ok;
  if (recordP == 0)
   { return 0;
   };
  if (bytes == 0)   // 0-length is ok
   { 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 xCopy (Char * dest, Char * xsrc, Int16 cnt)
{ if (! dest)
```

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


/*-------------------------------------------------------------------------------
Int16 Advance (Char * myptr, Int16 limit, Char target)
{
  Int16 i = limit;
  while (  (limit > 0)
         &&(*myptr++ != target)
        )
        { limit --;
        };
  if (limit < 1)
     { return 0;                                  /* fail */
     };
  return 1+(i-limit);                             /* one AFTER character located */
}


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


Char lowercase (Char c)
{ if (  (c >= 'A')
      &&(c <= 'Z')
      ) { return (c+0x20);
        };
  return c;
}
```

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

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

The following is similar to xSame, but forces left string s0 to UPPERCASE. We submit a second length parameter, even though we generally have an ASCIIZ string in this place, as we then don't need to clumsily determine the length of the second string.[3]

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

Int16 xFill (Char * p0, Int16 slen, Char c)
{   // allows fill of zero length (NB)
  while (slen > 0)
    { * p0 = c;
      p0 ++;
      slen --;
    };
 return 1;           // success
}
```

[3] We might formally check on the time savings, if any.

1.1.4 BetterCompare

Clumsy routine.

```
// BetterCompare:
// used below to compare two strings.
// return -1 if L<R, 0 if L=R, +1 if L>R.
  // WE SHOULD CHECK OUT THIS ROUTINE FOR CHARACTERS > 0x7F [fix me ??]
  // best cast as (UInt8 *)

Int16 BetterCompare(Char * L, Int16 llen, Char * R, Int16 rlen)
{
  if (llen > rlen)
     { while (  (rlen > 0)
              &&( * L++ == * R++ )
             )
         { rlen --;
         };
       if (! rlen)             // end of the line, so L *must* be greater
          { return 1;          // signal L > R
          };
       if (*(L-1) > *(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 = N
          };
       return -1;             // L < R
     };
  if (*(L-1) > *(R-1))
     { return 1;              // L > R
     };
  return -1;                  // L < R
}
```

1.1.5 Writing an integer

This simply writes the integer to the specified stack location, and doesn't alter the top of the stack.

```
Int16 writeinteger (Char * STACK, Int16 itm, Int32 i)
{ Char c;
  c = 'I';                                  /* result must be integer */
  w_DmWrite(STACK, itm+15, &c, 1, SMAX);
  c = 4;
  w_DmWrite(STACK, itm+14, &c, 1, SMAX);
  w_DmWrite(STACK, itm, &i, 4, SMAX);
  return 1;
}
```

In a similar fashion, we write and PUSH an integer to the stack top:

```
Int16 writepushinteger (Char * STACK, Int32 i)
{ Int16 top;
  top = *((Int16 *)(STACK+oTOP));
  writeinteger(STACK, top, i);
  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);
  return 1;
}
```

1.1.6 Bad legacy routines

The following are rubbish and should be replaced wherever found

```
Int16 beReadInt16 (Char * mP)
{
   return(   (*(mP+1) & 0xFF ) + ( (*(mP))<<8 )    );
}

Int16 beWriteInt16 (Char * P, Int16 i)
{  /* big-endian write, byte by byte */
   * P = i/256;                    /* or >> 8; compiler won't care */
   *(P+1) = i & 0xFF;
   return 1;                       /* ok */
}
```

1.1.7 UnInteger

Clumsy and limited at present. We really should allow negative numbers, but limit these to over -999 999 999.

```
Int16 UnInteger (Char * dest, Int16 dlen, Char * srcp)
```

```
{ // given big-endian integer in range +- 999999999 inclusive, write as integer
  // no frills.
  Int32 i;

  if ((! dest) || (dlen < 1))
     { return -1; // fail.
     };
  // 1. read integer
  i = *((Int32 *)srcp); // big endian [?ARM??]
  if ((i > 999999999) || (i < -999999999))
     { return -ScErIntegerWrite4;
     };
  return ( w_StrIToA(dest, dlen, i) );
}
```

1.2 Pop Boolean

take integer off stack. return 1/0/error.

```
Int16 popboolean (Char * STACK)
{
  Int16 bottom;
  Int16 top;
  Int32 i;

  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErBool;                         /* fail: insufficient stack */
     };
  top -= XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        /* pop one item */
  if ( *(STACK+top+15) != 'I' )
     { return -ScErBool;
     };
  i = *((Int32 *)(STACK+top));
  if (!i)
     { return 0;
     };
  if (i == 1)
     { return 1;
     };                                          /* clumsy */
  return -ScErBool;
}



/* ============================================================================
```

```
/* C. INITIALISATION */

// Start of STACK buffer contains 16 reserved bytes. Format is:
//   name    offset     Contents
//   oSTART  0          Start of *actual stack*
//   oTOP    2          current top of stack (first free byte)
//   oMAX    4          maximum stack size (size of stack from 0 to last byte)
//   oFLAGS  6          unused at present
//   oMARKS  8          number of marks set
//   -       10..15     unused

// Between these 16 bytes and oSTART, we store marks. At present, we allow up
// to 32 marks. Each mark is an ABSOLUTE (0-relative) offset into the stack
// every time we mark, we increment the count at oMARKS, and store the previous
// value in oSTART at the next mark position; when we RETURN/UNMARK we reverse
// this process. We need 32*2 bytes to store the 32 marks; the first is stored
// at offset 16. BOTTOMSTACK must thus be at least 96.
// When we pop the marks down to set oMARKS at zero, then oSTART is reset to
// BOTTOMSTACK.

/*-------------------------------------------------------------------------------
Int16 ScriptIni (UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int16 LOCAL = BOTTOMSTACK;
  if (! STACK)                                    /* simple initial tests */
     { return -ScErStackIsNull;
     }
  if (! STACKSTRING)
     { return -ScErStackstringIsNull;
     };
  if (! w_DmWrite(STACK, oSTART, &LOCAL, 2, SMAX))
     { return -ScErFailedWriteStackStart;
     };
  if (! w_DmWrite(STACK, oTOP,  &LOCAL, 2, SMAX))
     { return -ScErFailedWriteStackTop;
     };
  LOCAL = SMAX;
  if (! w_DmWrite(STACK, oMAX, &LOCAL, 2, SMAX))
     { return -ScErFailedWriteStackMax;
     };
  LOCAL = 0;
  if (! w_DmWrite(STACK, oFLAGS, &LOCAL, 2, SMAX))
     { return -ScErFailedWriteStackFlags;
     };

  // enough of this? [DO WE REALLY NEED ALL OF THE ABOVE TESTS? [no] ]
  w_DmWrite(STACK, oMARKS, &LOCAL, 2, SMAX);
```

```
  LOCAL = BOTTOMSTACKSTRING;
  if (! w_DmWrite(STACKSTRING, oSTART, &LOCAL, 2, MAXSS))
     { return -ScErFailedWriteSSStart;
     };
  if (! w_DmWrite(STACKSTRING, oTOP,  &LOCAL, 2, MAXSS))
     { return -ScErFailedWriteSSTop;
     };
  LOCAL = SMAX;
  if (! w_DmWrite(STACKSTRING, oMAX, &LOCAL, 2, MAXSS))
     { return -ScErFailedWriteSSMax;
     };
  LOCAL = 0;
  if (! w_DmWrite(STACKSTRING, oFLAGS, &LOCAL, 2, MAXSS))
     { return -ScErFailedWriteSSFlags;
     };
  return 1;                                        /* ok */
}
```

1.3 Pushing and Popping the stack

1.3.1 Push the stack

Our conservative strategy here is to try our best — unless a push is *impossible* we always push something, but if nonsense was specified, we push null. SayErr will write any error to the console. PushItem returns 0 on failure, 1 on success.

```
Int16 PushItem (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * itm, Int16
                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";

  // a. only fail BAAADly if cannot push something!

  if (! STACK)
      { SayErr (refnum, ErPushNoStack);
        return 0;
      };
  top = *((Int16 *)(STACK+oTOP));
  max = *((Int16 *)(STACK+oMAX));
```

```
if ((max - top) < XVI)
   { SayErr (refnum, ErPushStackFull);
     return 0;
   };

// b. Else, push null, which might become the default
if (  !(w_DmWrite(STACK, top, null8, 8, SMAX)) // slow but safe
    ||!(w_DmWrite(STACK, top+8, null8, 8, SMAX))
   )
   { SayErr (refnum, ErPushFailed);
     return 0;
   };
top += XVI; // write new stack top.
w_DmWrite(STACK, oTOP, &top, 2, SMAX);
top -= XVI; // restore.

if (! itm)
   { return 1; // not an error: 'push nothing' ie null!
   };
if (ilen < 0)
   { SayErr (refnum, ErPushBadLength);
     return 0;
   };

w_DmWrite(STACK, top+15, &itype, 1, SMAX);
sc = (Char) scale;
w_DmWrite(STACK, top+13, &sc, 1, SMAX);

if (ilen <= 14)
   { w_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)
        { SayErr (refnum, ErLongPushFull);
          return 0;
        };
     w_DmWrite(STACK, top, &ilen, 2, SMAX);
     w_DmWrite(STACK, top+2, &strtop, 2, SMAX);
     if (! w_DmWrite(STACKSTRING, strtop, itm, ilen, MAXSS) )
        { SayErr (refnum, ErLongPushFail);
          return 0;
        };
     strtop += ilen;
     w_DmWrite(STACKSTRING, oTOP, &strtop, 2, MAXSS);
   };
```

```
  w_DmWrite(STACK, top+14, &plen, 1, SMAX);
  return 1;   // success
}


/*-------------------------------------------------------------------------------
Int16 PopItem (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * dest, Int16
{  // note that RLEN is byreference.
   // SUBMIT maximum length, return actual length
   // if fail will return -ve code.
   // if succeed, return type of datum ie cast Char to Int16. rather ugly
   // THERE IS no CHECKING THAT TYPE IS VALID!
   // if dest is null DO NOT COPY!


  Int16 top;
  Int16 bottom;
  Int16 ilen;
  Int16 strg;
//  Int16 strtop;
  Int16 destmax;
  Char MYTYPE;

  if (! STACK)
     { return -ScErPopNoStack;
     };
  destmax = * RLEN;

  bottom = beReadInt16(STACK + oSTART);
  top = beReadInt16(STACK + oTOP);
  if (bottom >= top)
     { return -ScErPopStackEmpty;
     };

  top -= XVI;
  MYTYPE = *(STACK+top+15);
  ilen = 0x0F & ((Int16) *(STACK+top+14));  // ????????? is this Int16* correct Hm
  if (ilen < 15)
     { if (dest)
          { if (ilen > destmax)
               { return -ScPopFull;
               };
            xCopy(dest, STACK+top, ilen);
          };
     } else
     { ilen = beReadInt16(STACK+top);
       strg = beReadInt16(STACK+top+2);
       if (dest)
```

```
          { if (ilen > destmax)
               { return -ScPopFullLong;
               };
            xCopy(dest, STACKSTRING+strg, ilen);
          };

/*
// if we allow eXtract to function as it now does, we cannot do the following:
       strtop = beReadInt16(STACKSTRING+oTOP);
       if ((strg + ilen) != strtop)
          { return -ScErPopStack;
          };
*/

       w_DmWrite(STACKSTRING, oTOP, &strg, 2, MAXSS);
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);
  * RLEN = ilen;
  return (Int16) MYTYPE;
}


/*-------------------------------------------------------------------------------
/* StackPeek:
   Formerly simply looked at length of item.
   Now perform complete PopItem, BUT do not alter stack.
   Useful for pulling something off and then popping it, or checking stack!
   NOTE that if dest is NULL, then no error occurs (but of course, nothing is writ

   NB. *DOES* REQUIRE a (maximum) value in RLEN!
*/

Int16 StackPeek (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * dest, Int
{
  Int16 top;
  Int16 bottom;
  Int16 ilen;
  Int16 strg;
//  Int16 strtop;
  Int16 destmax;
  Char MYTYPE;

  if (! STACK)
     { return -ScErPopNoStack;
     };
  destmax = * RLEN;                              /* clumsy */

  bottom = beReadInt16(STACK + oSTART);
  top = beReadInt16(STACK + oTOP);
```

```
  if (bottom >= top)
     { if (bottom > top)   // should NOT happen, but if so, attempt restitution
          { return -ScErStrangeBottom;          // do not attempt to fix!
          };
       return -ScErPopStackEmpty;               /* nothing to pop */
     };

  top -= XVI;
  MYTYPE = *(STACK+top+15);
  ilen = 0x0F & *(STACK+top+14);      /* just in case, AND out higher nybble */
  if (ilen < 15)                               /* if short item */
     { if (ilen > destmax)
          { return -ScPopFull;              /* won't fit into destination */
          };
       if (dest){ xCopy(dest, STACK+top, ilen); };
     } else                                    /* long item */
     { ilen = *((Int16 *)(STACK+top));
       strg = *((Int16 *)(STACK+top+2));
       if (ilen > destmax)
          { return -ScPopFullLong;          /* won't fit */
          };
       if (dest) { xCopy(dest, STACKSTRING+strg, ilen); };
/*
// the following will conflict with eXtract..
       // here follows test of stack integrity
       strtop = *((Int16 *)(STACKSTRING+oTOP));
       if ((strg + ilen) != strtop)
          { return -ScErPopStack;               // stack error ? unmatched BURY
          };
       // end of stack integrity test
*/
     };

  * RLEN = ilen;                               /* write length */
  return (Int16) MYTYPE;                       /* return 'char' type, encoded as
}
```

1.3.2 Push NULL to stack

We put a NULL onto the top of the stack, incrementing the top by XVI (one item).
The default null type is 'V'.

```
Int16 PushNull(Char * STACK)
{ Int16 top;
  Int16 bottom;
```

```
  Char c;

  top    = *((Int16 *)(STACK+oTOP));
  bottom = *((Int16 *)(STACK+oSTART));
  if (top < bottom)
     { top = bottom; // clumsy hack
     }; // [hope will never need]
        // [should write error too]

  c = 0;
  w_DmWrite(STACK, top+14, &c, 1, SMAX); // signal null (no length)
  c = 'V';
  w_DmWrite(STACK, top+15, &c, 1, SMAX);
  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // push
  return 1;
}
```

1.3.3 Write NULL to stack

WriteNull is similar to PushNull, but the item at the top of the stack is overwritten. There is no push.

```
Int16 WriteNull(Char * STACK)
{ Int16 bottom;
  Int16 top;
  Char c;
  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top < bottom)
     { return WriteNull(STACK);  // hack
     };
  c = 0;
  w_DmWrite(STACK, top+14, &c, 1, SMAX); // signal null
  c = 'V';
  w_DmWrite(STACK, top+15, &c, 1, SMAX); // default null type
  return 1;
}
```

1.3.4 Mark stack

```
/* MARK:
  Primitive stack-limiting function: sets new value for STACK+oSTART
  Takes an integer argument:
    if 0, then stack is limited at current position (after 0 taken off)
    if *POSITIVE* INTEGER (ALTER PERL)  mark this number of XVIs *down* on stack!
```

```
    NOTES:
    1. The CURRENT bottom is stored in a mark stack
    2. The current stack top is taken and we subtract n items, where n is the
       number supplied as the sole argument of MARK. We set the *new* bottom to
       this adjusted value
       [STACK+oMARKED] is the pointer to the first unused item on the top of
       the mark stack!
    3. When we UNMARK, we CLEAR THE STACK DOWN TO THE CURRENT BOTTOM, and then
    restore the previous bottom (from the mark 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 -ScErMarkEmpty;                    // fail: insufficient stack
     };

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

  top -= XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        // pop just one item
  if (* (STACK+top+15) != 'I')
     { return -ScErMarkArg;
     };
  i = (Int16) (*((Int32 *)(STACK+top)));         // get int32 but as int16
  if (i > STACKCOUNT)                            // -> (won't work)
     { return -ScErMarkEmpty;                    // fail: insufficient stack
     };
  i *= XVI;
  if (top - i < bottom)
     { return -ScErMarkEmpty;                    // clumsy
     };
```

```
  bottom = top - i;
  w_DmWrite(STACK, oSTART, &bottom, 2, SMAX);    // set new bottom
  return 1;
}



/*-------------------------------------------------------------------------------
/* UNMARK:
   Undoes the above AND resets 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 -ScErMarkMin;
     };

  // here find out if stackstring is referenced, and 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 stackstr
           w_DmWrite(STACKSTRING, oSTART, &SStop, 2, MAXSS); // update SS
           B = T; // force exit
         };
      B += XVI;
    };


  // sort out top..
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);    // remove whole chunk from stack

  marked --;                                     // go back to stored 'bottom' valu
  w_DmWrite(STACK, oMARKS, &marked, 2, SMAX);   // write diminished count
```

```
  bottom = *((Int16 *)(STACK+16 + 2*marked));
  w_DmWrite(STACK, oSTART, &bottom, 2, SMAX);    // reset bottom

  return 1;
}



/*-------------------------------------------------------------------------
// UnmarkOnly:
//
// Occasionally, we wish to clear the mark, but NOT take the stack top back to
// the bottom (ie. preserve the current stack).
// Use UnmarkOnly. At present, only available for internal use!

Int16 UnmarkOnly(Char * STACK)
{ Int16 bottom;
  Int16 marked;
  marked = *((Int16 *)(STACK+oMARKS));           // get index of where to store cu
  if (marked < 1)                                // nada
     { return -ScErMarkMin;
     };
  marked --;                                    // go back to stored 'bottom' valu
  w_DmWrite(STACK, oMARKS, &marked, 2, SMAX);   // write diminished count
  bottom = *((Int16 *)(STACK+16 + 2*marked));
  w_DmWrite(STACK, oSTART, &bottom, 2, SMAX);    // reset bottom
  return 1;
}



/*-------------------------------------------------------------------------
/* DEPTH:
   how many items are there on stack?
*/

Int16 MarkDepth(Char * STACK)
{
  Int16 bottom;
  Int16 top;
  Int16 mlen;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  mlen = top-bottom;
  if (mlen < 0)
     { return -ScErBadDepth;
     };
```

```
  mlen /= XVI;
  writeinteger(STACK, top, mlen);
  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);
  return 1;
}

// even more direct is the following:
// PeekDepth:
//
// return integer = stack depth. May even be negative if gross stack error, as no



Int16 PeekDepth(UInt16 refnum, Char * STACK)
{
  Int16 bottom;
  Int16 top;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  return ((top - bottom)/XVI);
}




/*-------------------------------------------------------------------------------
/* RETURN:
   If stack is marked, then pop all of those items (and UNMARK)
   otherwise just 'return'
   [11/3/2005: NNOOOOOOOOOO! we do NOT force mark. Only unmark if commanded to do
   ]
*/

/*
Int16 ReturnMark(Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 Stop;
  Int16 ilen;

  bottom = *((Int16 *)(STACK+oSTART));
  if (bottom == BOTTOMSTACK)                              // if not marked
     { return 1;
     };

  // if marked, much more complex.
```

```
  // we have to
  // (a) pull STACK down to mark
  // (b) go up from mark on stack, and find first pointer to stackstring,
  //     and then pull STACKSTRING down to start of this (discarded) item!

  top = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)                              // nothing to do (might warn if
     { return 1;
     };
  w_DmWrite(STACK, oTOP, &bottom, 2, SMAX);     // pull stack back

  while (bottom < top)
    { ilen = 0x0F & *(STACK+bottom+14);
             // is it long?
      if (ilen > 14)
         { Stop = *((Int16 *)(STACK+bottom+2));  // get STACKSTRING pointer
           w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS);
           return 1;                             // lowest, so *done*
         };
      bottom += XVI;                              // move on up
    };
  return ClearMark(STACK, STACKSTRING); // MUST clear mark on return!
}
*/
```

1.4 Translation routines and so forth

1.4.1 Thirty bit integer read

Nominally 32 bits, but we only allow numbers in the range of TOOBIG . . . TOOSMALL.

```
Int32 readint32 (Char * P, Int16 ilen)
{ Int32 i;
  Char c;
  Int16 sgn=0;

  if (!P)
     { return TOOSMALL;
     };

  if (*P == '-')
     { sgn = 1;
       P++;
       ilen --;
     };
```

```
  if (ilen < 1)
     { return TOOSMALL;
     };

  if (ilen > MAXDIGITS)
     {return TOOBIG; // fail
     };

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

  if (sgn)
     { if (i > TOOBIG+1)
          { return TOOSMALL;
          };
       return -i;
     };

  if (i > TOOBIG)
     { return  TOOBIG;
     };
  return i;
}
```

1.4.2 Integer encoding

As usual, we limit integers to the range $\pm$(one billion minus 1).

Although it would seem that we don't ever need stackstring as an argument, if we had a long numeric this might in total be too long to fit into 13 chars (It might have a big scale, or be in error, and the stackstring also needs to be popped) If the submitted value is already integer, it's left unchanged.

In keeping with our more recent 'closed' philosophy, it's probably wise to return NULL if encoding of the integer fails. In addition this will allow us to validate an integer in a string! If we write a compensatory null, then we don't return an error, we return iCOMPENSATED!

First two minor readers of two and four digit unsigned decimal integers. There is *no* format check.

```
Int16 Read2Digits (Char * P)
{
  return (((*(P+1))-0x30) + 10*((*P)-0x30));
}

Int16 Read4Digits (Char * P)
{
  return Read2Digits(P+2) + 100*(Read2Digits(P));
}
```

Here's our Julian conversion routine. We submit a pointer to the formatted timestamp (YYYYMMDDHHMMSS) and return a float value, placing it in the float provided. The localtime and daylightsave are signed integer values in hours (daylightsave should really only be either 0 or 1). The variable stamp is 1 if just a date is to be read, 2 if only a time is to be read, and 3 if both are to be read! Baum's page at http://vsg.cape.com/ pbaum/date/jdalg2.htm is useful for the following. The algorith below should not be used for dates where z is negative (See code) ie. very early proleptic Gregorian dates.

```
Int16 JulianTimestamp (UInt16 refnum, Char * P,
                       double * dp, Int16 stamp)
{
      // for now we hack the values for localtime and daylightsave.
      // LATER WE MUST provide these to the library! [FIX ME?]
      // problem: what about converting older dates: need to LOOK AT ACTUAL DATE
      // to determine DAYLIGHTSAVE !! [fix me!!!]
      // WILL NEED TO HAVE COMPLEX ALGO, AND PROVIDE LOCATION!
 Int32 i, z, f;
 double D;
 Int16 yy, mo, dd, hh, mi, ss;
 Int16 localtime, daylightsave;

  // 1. get yyyy mo dd hh mi ss
  localtime = +12;
  daylightsave = 0;

 if (stamp & 1)
    { yy = Read4Digits(P);
      mo = Read2Digits(P+4);
      dd = Read2Digits(P+6);
      P += 8; // move to time!

      // 1.  Z = Y + fix[(M-14)/12]
      z = yy + ((mo-14)/12); // int

      // 2. F = (979*(M-12*((M-14)\12))-2918)\32
      f = (mo-14)/12; // int
```

```
      f = (979*(mo-12*f)-2918)/32 ; // int

      // 3. D+F+365*Z+[Z/4]-[Z/100]+[Z/400] + 1721118.5
      // in the above [x] denotes the floor of x.
      // provided Z isn't -ve, int(x) = floor(x) so we simply say:
      i = 365*z + z/4 - z/100 + z/400;
      D = dd + f + i + 1721118.5;

      // remove effects of local time, daylight saving:
      D -= ((double)(localtime+daylightsave))/24;
    } else
    { yy = 0;
      mo = 0;
      dd = 0;
      D = 0;
    };

  if (stamp & 2)
     { hh = Read2Digits(P);
       mi = Read2Digits(P+2);
       ss = Read2Digits(P+4);
     } else
     { hh = 0;
       mi = 0;
       ss = 0;
     };

  // 4. add in hours, min, sec:
  D += (hh + (mi + ((double)ss)/60)/60)/24;

  * dp = D;
  return 1;
}
```

For dates and times, we follow our Perl program's lead:

1. TIMESTAMP, DATE: Julian day;

2. TIME: convert HH:MM:SS to seconds;

Note that in contrast to Perl, which is untyped, we will *not* automatically coerce text formats other than an integer into a date/time and *then* convert to an integer. This is a failing of the Perl rather than this library!

We return 1 on success, 0 on failure.

```
Int16  EncodeInteger (UInt16 refnum, Char * STACK, Char *  STACKSTRING)
{
```

```
Int16 bottom;
Int16 top;
Int16 ilen;
Int16 scale;
Int16 Stop;
Char * P=NULL; // humour compiler
Char c;
double d;
Int32 i=0; // hmm.
d = 0; // minimise variation.

bottom = beReadInt16(STACK+oSTART);
top = beReadInt16(STACK+oTOP);
if (top <= bottom)
   { // should we push a null anyway? yep.
     PushNull (STACK);
     SayErr (refnum, ErEncodeIntegerEmpty);
     return 0; // fail
   };
top -= XVI;
c = *(STACK+top+15); // get type
ilen = 0x0F & (*(STACK+top+14));

switch (c)
  {
    case 'V':  // read integer from text
      P = STACK+top;
      i = readint32(P, ilen);
      break;

    case 'N':  // turn numeric into integer
      if (ilen > 13)
         { ilen = *((Int16*)(STACK+top));
           Stop = *((Int16 *)(STACK+top+2));
           P = STACKSTRING + Stop;
           w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS); // clean stackstring
         } else
         { P = STACK+top;
         };
      scale = *(STACK+top+13);
      ilen -= scale;
      if (ilen <= 0)
         { i = 0; // final value is ZERO
         } else
         { i = readint32(P, ilen);
         };
      break;
```

```
      case 'I':  // ok, so just exit!
        return 1;

      case 'F':
        d = * ((double *)(STACK+top)); // clumsy

        /* ************************************************************
        // AT PRESENT DO *NOT* ROUND, BUT MIGHT CONSIDER EG:
        if (d < 0)                     // we might jack up this rounding ???
           { d -= 0.5;                 // (von Neumann?) (options?)
           } else
           { d += 0.5;
           };
        */ ////////////////////////////////////////////////////////////

        i =  (d+EPSILON);
          // hmm (Int16) cast screws this! [why?]
          // should also watch for overflow [? check me]
        break;

      case 'D':
      // [might check ilen]
      JulianTimestamp(refnum, P, &d, 1);
      // similar to TIMESTAMP below, but only date.
      i = (d+EPSILON);
      break;

      case 'T':
      JulianTimestamp(refnum, P, &d, 2);
      // read only the time!
      d *= 86400; // convert to seconds!
      i = (d+EPSILON);
      break;

    case 'S':
      // [might check ilen]
      JulianTimestamp(refnum, P, &d, 3);
      i = (d+EPSILON);
      break;

    default:
      SayErr (refnum, ErEncodeIntegerType);
      return 0;
    };

  if ( (i >= TOOBIG)||(i <= TOOSMALL) )
     { WriteNull(STACK);
       SayErr (refnum, ErEncodeIntegerOverflow);
```

```
      return 0;
    };

  w_DmWrite(STACK, top, (Char *)&i, 4, SMAX);
  // above byte order needs to be reversed on ARM microprocessor
  // we could check that the above succeeded
  // we might even fill bytes 4..13 with zeroes
  c = 'I';  // integer
  w_DmWrite(STACK, top+15, &c, 1, SMAX);
  c = 4;    // length IS 4
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  return 1; // ok
}
```

1.4.3 Encoding a floating point number

As with EncodeInteger, we here need to be able to deal with various data types:

1. integers become floats without ado;
2. strings are read using the clumsy internal PalmOS function [fix me!];
3. dates and timestamps are converted to Julian dates;
4. times are converted to seconds.

Here's a small ancillary function:

```
Int16 w_FlpBufferAToF( void * dP, Char * P)
{
 FlpBufferAToF((FlpDouble *)dP, P);
 return 1;
}
```

We MUST REPLACE the above WITH BETTER CODE! The PalmOS code limits number of digits in mantissa and exponent! also doesn't report stuffups.

```
Int16 EncodeFloat (UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int32 i;
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 iscale;
  Int16 itype;
  Int16 ssoff;
  Char * P;
```

```
Char * S;
Char c;
double d;

bottom = *((Int16 *)(STACK+oSTART));
top = *((Int16 *)(STACK+oTOP));
if (top <= bottom)
   { PushNull (STACK); // 'fix'
     SayErr (refnum,ScErEncodeFloatEmpty);
     return 0;
   };
top -= XVI;
itype = *(STACK+top+15);
iscale = *(STACK+top+13);
ilen = 0x0F & *(STACK+top+14);

c=0x0;
if (ilen > 14)  // long string
   { ilen = beReadInt16(STACK+top);
     ssoff = beReadInt16(STACK+top+2);
     if (ilen > MAXIMUMFLOATLENGTH)
        { WriteNull (STACK); // replace c null
          SayErr (refnum,ScErFloatLength);
          return 0;
        };
     S = STACKSTRING+ssoff;
     w_DmWrite(STACKSTRING, oTOP, &ssoff, 2, MAXSS);
      // string used so trash it
   } else
   { S = STACK+top;
   };
P = xNew(MAXIMUMFLOATLENGTH+1+2);
   // clumsy, +1 for asciiz +2 for "0."

switch(itype)
  {
    case 'V':
      xCopy(P,S,ilen);
    break;

    case 'N':
      if (iscale == 0)                          // actually use a generic write r
         { xCopy(P,S,ilen);                     // [fix me!]
         } else
         { if (iscale < ilen)
              { ilen -= iscale;
                xCopy(P,S,ilen);
                *(P+ilen) = '.';
```

```
                  xCopy(P+ilen+1,S+ilen,iscale);
                  ilen += iscale+1;
                } else
                { iscale -= ilen;
                  xCopy(P,"0.",2);
                  xFill(P+2, iscale, '0');        // pad with zeroes
                  xCopy(P+2+iscale,S,ilen);
                  ilen += 2+iscale;
                };
             };
      break;

      case 'I':
          i = *((Int32 *)(S)); // watch out: ARM processor?
          d = i; // [or simply allocate d directly]
          goto dammit;  // or variants without the goto.
      break;

      case 'T':
      JulianTimestamp(refnum, S, &d, 2);
      d *= 86400; // convert to seconds!
      goto dammit;

      case 'D':
      JulianTimestamp(refnum, S, &d, 1);
      goto dammit;

      case 'S':  //
      JulianTimestamp(refnum, S, &d, 3);
      goto dammit;

      default:
        Delete(P);
        WriteNull (STACK);
        SayErr (refnum,ScErEncodeFloatType);
        return 0;
    };

  *(P+ilen) = 0x0; // asciiz for PalmOS
  // now P points to source, ilen is length of this source
  w_FlpBufferAToF( (void *)&d, P);  // ugly

dammit:
  w_DmWrite(STACK, top, (Char *) &d, 8, SMAX);  // copy over float
  Delete(P);
  c=8; //float length is 8
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  c='F';
```

```
  w_DmWrite(STACK, top+15, &c, 1, SMAX); // ugly
  return 1;  // ok
}
```

textEncodeFloat

This is similar to the above; 'later on' will use unified, better routines! Assumes 8 byte destp (at least) THERE IS NO FORMAT CHECKING OF FLOATING POINT NUMBER. BUGGER. (Crippled PalmOS routines).

```
Int16 textEncodeFloat (Char * destp, Char * srcp, Int16 srclen)
{
  Char * dbuf;
  dbuf = xNew(srclen+1);                          /* hideous asciiz again */
  xCopy(dbuf,srcp,srclen);
  * (dbuf+srclen) = 0x0;
  w_FlpBufferAToF( (void *)destp, dbuf);      /* returns null  */
  Delete(dbuf);
  return 1;
}
```

1.4.4 TIMESTAMP routine

Related in some ways to FLOAT and INTEGER. Given a Julian day as an integer or Julian date as a float, we turn these into a timestamp. A DATE is also converted to a timestamp (with time 00:00:00)[4] but submitting a TIME is meaningless and gives NULL. A string is read and translated into a TIMESTAMP. The string format is fairly rigorous: 'YYYY-MM-DD HH:MM:SS', but some leeway is allowed — single digits are permitted for MM, DD, HH, MM and SS. First, subsidiary routines. ReadTextTs reads a complete timestamp into a 14 digit timestamp.

We also need to be able to read either one or two digits based on a separator, so we have a minor routine to do this, OneOr2. In the following routine the destination is desti, the string is S, separator sep, and maximum string length max. We always write two characters to desti, unless we failed.

```
Int16 OneOr2 (Char * desti, Int16 max, Char * S, Char sep)
{
  Char c;
  if (max < 1)
     { return 0; // fail
     }; // might also check S, desti not null [??]
```

[4] Soo convenient!

```
  c = *S;
  if (  (c < '0')
      ||(c > '9')
     )
     { return 0;
     };
  * desti = '0';
  * (desti + 1) = c;
  if (max < 2)
     { return 1;
     };

  S++;
  c = *S;
  if (c == sep)
     { return 2; // 2 chars
     };
  if (  (c < '0')
      ||(c > '9')
     )
     { return 0;
     };
  * desti = *(desti+1);
  * (desti+1) = c;
  if (max < 3)
     { return 2;
     };

  S++;
  c = *S;
  if (c != sep)
     { return 0; // fail!
     };
  return 3; // 3 chars read
}
```

We have a little problem in the above in that we might be reading say the seconds at the end of a time, and there could conceivably only be one digit, with no ASCIIZ terminator. This is why we also submit a maximum length. On failure, zero is put into the destination integer.

We return the value in the byref integer desti, and 0 on failure, or the number of characters read, *including* the separator. Let's read a text timestamp, which returns 1 on sucess, 0 on failure. Reading a timestamp comprises two components, reading a date, and reading a time. We have routines for both.

In reading a date, we submit a source and destination, and the length of the source (We assume that the destination is big enough). Normally slen will con-

strain our final testing, otherwise a space is assumed to follow the date! We return the total number of characters read. Eight characters are written to the destination.

```
Int16 ReadTextDate(Char * dst, Char * src, Int16 slen)
{ Int16 i;
  Int16 tot;

  if (slen < 8)
     { return 0;
     };

  // YYYY: FIRST we simply move over 4 characters:
  i = OneOr2 (dst, 2, src, 0x0);
  if (!i)
     { return 0; // fail
     };
  src += 2;
  slen -= 2;
  dst += 2;

  i = OneOr2 (dst, slen, src, '-');
  if (!i)
     { return 0; // fail
     };
  src += i;
  slen -= i;
  dst += 2;
  tot = 2+i;

  // MM or M:
  i = OneOr2(dst, slen, src, '-');
  if (!i)
     { return 0; // fail
     };
  src += i;
  slen -= i;
  dst += 2;
  tot += i;

  i = OneOr2 (dst, slen, src, ' ');
  if (!i)
     { return 0; // fail
     };
  return (tot+i);
}
```

Reading a time is similar. The length should force termination, but we allow for an ASCIIZ terminator. Six characters are written to the destination.

```
Int16 ReadTextTime(Char * dst, Char * src, Int16 slen)
{ Int16 i;
  Int16 tot;

  i = OneOr2 (dst, slen, src, ':');
  if (!i)
     { return 0; // fail
     };
  src += i;
  slen -= i;
  tot = i;
  dst += 2;

  i = OneOr2 (dst, slen, src, ':');
  if (!i)
     { return 0; // fail
     };
  src += i;
  slen -= i;
  tot += i;
  dst += 2;

  i = OneOr2 (dst, slen, src, 0x0);
  if (!i)
     { return 0; // fail
     };
  return (tot+i);
}
```

We amalgamate the two previous routines to read a timestamp:

```
Int16 ReadTextTs(Char * dst, Char * src, Int16 slen)
{ Int16 i;
  Int16 tot;

  i = ReadTextDate (dst, src, slen);
  if (!i)
     { return 0; // fail
     };
  dst += 8;
  src += i;
  slen -= i;
  tot = i;

  // a wrinkle: if no more, write 000000 as time!
  if (! slen)
     { xFill(dst, 6, '0');
       return tot; // helpful hint
```

```
     };

  i = ReadTextTime (dst, src, slen);
  if (!i)
     { return 0; // fail
     };
  return (tot+i);
}
```

Here's a minor copying routine for scaling of numerics:

```
Int16 ScaleNumeric(Char * P, Char * S, Int16 ilen, Int16 iscale)
{
  if (iscale == 0)
     { xCopy(P,S,ilen);
     } else
     { if (iscale < ilen)
          { ilen -= iscale;
            xCopy(P,S,ilen);
            *(P+ilen) = '.';
            xCopy(P+ilen+1,S+ilen,iscale);
            ilen += iscale+1;
          } else
          { iscale -= ilen;
            xCopy(P,"0.",2);
            xFill(P+2, iscale, '0');        // pad with zeroes
            xCopy(P+2+iscale,S,ilen);
            ilen += 2+iscale;
          };
       *(P+ilen) = 0x0; // ASCIIZ for FlpBuffer Fx.
     };
  return 1; //ok
}
```

In addition, we want to convert numbers (integers, fixed point and floats) to dates. Numbers will represent Julian dates. Our conversion routine ('Gregorian') is preceded by a few minor functions:

```
void Write2Digits(Char * P, Int16 i)
{ *(P) = '0' + i/10;
  *(P+1) = '0' + i%10;
};

void Write4Digits(Char * P, Int16 i)
{ Write2Digits (P, i/100);
  Write2Digits (P+2, i%100);
}
```

Here's our converter (after our Perl version). It makes use of the *floor* function from Rick Huebner's Mathlib library, which we invoke in the following minor routine. Note that *our* floor returns an integer!

```
Int32 ifloor(UInt16 refnum, double x)
{
   double result;
  UInt16 MathLibRef;
  Int32 i;
  SysLibTblEntryPtr entryP = SysLibTblEntry (refnum);
  ScriptingLib_globals *gl = entryP->globalsP;
  if (!gl)
  {  return 0; // hmm must write error
              // (need integer NaN)!
  };
  MathLibRef = gl->MATHLIB;
  if (! MathLibRef)
  {  return 0; //hmm must write error
     // return ((x-x)/(x-x)); //not [NaN]
  };
   MathLibFloor(MathLibRef, x, &result);
  i = (result+EPSILON);
   return i;
}
```

We add epsilon to fix the case where the result is say 2.9999999 and truncation gives 2.

```
Int16 Gregorian(UInt16 refnum, Char * dst, double * Pd, Int16 flags)
{
  Int32 z, a, b, year, month, hours, min, sec, q;
  double JD, R, G, C, DAY, S;
  Int16 localtime, daylightsave;
  localtime = +12;
  daylightsave = 0;
  // as for Julian, we STILL need to fix up this routine to
  // accommodate other locations and daylight saving by location.
  // a significant task! [FIX ME]

  JD = * Pd; // Julian date
  JD += ((double)(localtime+daylightsave))/24;
  // the time returned is *local*!

  // [here might check range of JD]
  z = ifloor(refnum, (JD - 1721118.5));
  R = JD - 1721118.5 - z;
  G = z - 0.25;
  a = ifloor(refnum, (G / 36524.25));
```

```
  b = a - ifloor(refnum, (a / 4));
  year = ifloor(refnum,((b+G) / 365.25));
  C = 365.25 * (double)year;
  q = ifloor(refnum, C);
  C = b + z - q;
  month = ((5 * C + 456) / 153); // int
  q = ((153.0 * (double)month - 457.0) / 5.0); // int
  DAY = C - q + R;
  if (month > 12)
     { year ++;
       month -= 12;
     };

  // calculate hours, min, sec:
  q = DAY; // integer
  S = DAY;
  S -= q;
  S *= 24;
  hours = S; // int
  S -= hours;
  S *= 60;
  min = S; // int
  S -= min;
  S *= 60;
  sec = (S+EPSILON); // int
  // we will disregard fractional seconds, for now.

 // next, write to dst:
 if (flags & 0x01)
    { Write4Digits(dst, year);
      Write2Digits(dst+4, month);
      Write2Digits(dst+6, DAY);
      dst += 8;
    };

 if (flags & 0x02)
    { Write2Digits(dst, hours);
      Write2Digits(dst+2, min);
      Write2Digits(dst+4, sec);
    };
  return 1;
}
```

We assume (eugh) in the above that dest is of sufficient length (caller must ensure). If bit 0 of flags is set we write a date; if bit 1 is set, a time. We do not return the length of the string written as the caller must know it! We've moved the main TIMESTAMP routine down below so it can make use of eXtract:

1.4.5 Time encoding

We do not at present validate the time content. This routine is similar to, and based on TimeStamp above. We accept a time, integer float or numeric (as seconds), and at present don't allow fractional seconds. Submitting a timestamp or date results in error. For another similar routine, see *Sql3lib.tex*.

```
Int16 EncodeTime (UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int32 i;
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 iscale;
  Int16 itype;
  Int16 ssoff;
  Char * P;
  Char * S;
  Char c;
  double d;

  bottom = *((Int16 *)(STACK+oSTART));
  top = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { PushNull (STACK); // 'fix'
       SayErr (refnum,ScErEncodeTimeEmpty);
       return 0;
     };
  top -= XVI;
  itype = *(STACK+top+15);
  iscale = *(STACK+top+13);
  ilen = 0x0F & *(STACK+top+14);

  if (itype == 'T')
     { return 1;
     }; // success.

  c=0x0;
  if (ilen > 14)  // long string
     { ilen = beReadInt16(STACK+top);
       ssoff = beReadInt16(STACK+top+2);
       S = STACKSTRING+ssoff;
       w_DmWrite(STACKSTRING, oTOP, &ssoff, 2, MAXSS);
        // string used so trash it
     } else
     { S = STACK+top;
     };
  if (ilen < 10) // might make smaller (time alone)
```

```
    { P = xNew(10);  // more than enough
    } else
    { P = xNew(ilen+1+2); // see fixed pt usage
    }; // watch the numerics. they bite.

  switch(itype)
    {
      case 'V':
        i = ReadTextTime(P, S, ilen);
        if (i < 1)
           { WriteNull (STACK);
             SayErr (refnum,ScErTimeLength);
             return 0;
           };
      break;

      case 'N':
      // similar ugly hack to that for timestamp
      // must rewrite using fixed point.
        ScaleNumeric(P, S, ilen,iscale);
        // convert number to float: [ugh]
        w_FlpBufferAToF((void *)&d,P);
        // need error check in the above!

        // convert float to time:
          d /= 86400; // seconds -> fraction of a day!
          Gregorian (refnum, P, &d, 0x02); // write time
      break;

      case 'I':
          i = *((Int32 *)S); // watch out: ARM processor?
          d = i; // ugly float
          d /= 86400; // seconds -> fraction of a day!
        Gregorian (refnum, P, &d, 0x02); // write time
      break;

      case 'F':  // float:
        d= *((double *)S);
        d /= 86400; // seconds -> fraction of a day!
        Gregorian(refnum, P, &d, 0x02); // time only
      break;

      default:
        Delete(P);
        WriteNull (STACK);
        SayErr (refnum,ScErEncodeTimeType);
        return 0;
    };
```

```
  c=6; // no partial seconds for now.
  w_DmWrite(STACK, top, P, c, SMAX);  // write stamp
  Delete(P);
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  c='S';
  w_DmWrite(STACK, top+15, &c, 1, SMAX); // nasty
  return 1;  // ok
}
```

1.4.6 Tick count

The internal tick counter is accurate to about 10ms. Here we provide this raw 'number of ticks since the system started', putting it as an integer on the stack. We limit the integer value to 1 billion using a logical AND mask.

```
Int16 GetTicks(UInt16 refnum, Char * STACK)
{
  UInt32 t;
  Int16 top;
  Char c;

  top = beReadInt16(STACK + oTOP);
  t = TimGetTicks();
  t &= 0x3FFFFFFF; // limit to 1 Gigaticks

  if (! w_DmWrite(STACK, top, (Char *)&t, 4, SMAX))
     { return 0; // fail
     };
  // 1. ARM swop req'd; 2. (might 0-fill bytes 4--13)

  c = 'I';       // integer type
  w_DmWrite(STACK, top+15, &c, 1, SMAX);
  c = 4;         // length always 4
  w_DmWrite(STACK, top+14, &c, 1, SMAX);

  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);

  return 1;      // ok
}
```

1.4.7 Date encoding

Similar to the above, with no content validation, and rejection of TIME arguments. Numerics represent Julian dates. We initially thought it might be an idea to allow submission of a timestamp (retrieving the date) but this is silly (split and discard)!

```
Int16 EncodeDate (UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int32 i;
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 iscale;
  Int16 itype;
  Int16 ssoff;
  Char * P;
  Char * S;
  Char c;
  double d;

  bottom = *((Int16 *)(STACK+oSTART));
  top = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { PushNull (STACK); // 'fix'
       SayErr (refnum,ScErEncodeDateEmpty);
       return 0;
     };
  top -= XVI;
  itype = *(STACK+top+15);
  iscale = *(STACK+top+13);
  ilen = 0x0F & *(STACK+top+14);

  if (itype == 'D')
     { return 1;
     }; // success.

  c=0x0;
  if (ilen > 14)  // long string
     { ilen = beReadInt16(STACK+top);
       ssoff = beReadInt16(STACK+top+2);
       S = STACKSTRING+ssoff;
       w_DmWrite(STACKSTRING, oTOP, &ssoff, 2, MAXSS);
        // string used so trash it
     } else
     { S = STACK+top;
     };
  if (ilen < 12) // might make smaller (time alone)
     { P = xNew(12);  // more than enough
     } else
     { P = xNew(ilen+1+2); // see fixed pt usage
     };

  switch(itype)
    {
```

```
      case 'V':
        i = ReadTextDate(P, S, ilen);
        if (i < 1)
           { WriteNull (STACK);
             SayErr (refnum,ScErDateLength);
             return 0;
           };
      break;

      case 'N':
      // similar ugly hack to that for timestamp
      // must rewrite using fixed point.
        ScaleNumeric(P, S, ilen,iscale);
        // convert number to float: [ugh]
        w_FlpBufferAToF((void *)&d,P);
        // need error check in the above!
        // convert float to date:
          Gregorian (refnum, P, &d, 0x01); // write date
      break;

      case 'I':
          i = *((Int32 *)S); // watch out: ARM processor?
          d = i; // ugly float
          Gregorian (refnum, P, &d, 0x01); // write date
      break;

      case 'F':  // float:
        d= *((double *)S);
          Gregorian (refnum, P, &d, 0x01); // write date
      break;


      default:
        Delete(P);
        WriteNull (STACK);
        SayErr (refnum,ScErEncodeDateType);
        return 0;
    };

  c=8; // YYYYMMDD
  w_DmWrite(STACK, top, P, c, SMAX);  // write stamp
  Delete(P);
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  c='S';
  w_DmWrite(STACK, top+15, &c, 1, SMAX); // nasty
  return 1;  // ok
}
```

```
/* ========================================================================
/* F. ENCODING OF DATA */

/* EncodeAll: Has been moved in its entirety to Sql3Lib.tex. A stub. */

Int16 EncodeAll (UInt16 refnum, Char * destin, Int16 destsize,
                 Char * datum, Int16 datlen,
                 Char typeofdatum, Int16 scale)
{ return -ScErEncode;
}
```

1.4.8 PushNumber

An ugly stub at present. Needs an *enormous* amount of work!

```
/* ========================================================================
/* M. PushNumber:  [moved up because of invocation]
 when script interpreter encounters numeric, this is the routine invoked.
 sgn is a stub for the sign (0=positive, 1=negative) ???
*/

Int16 PushNumber (Char * STACK, Char * STACKSTRING, Char * SCRIPT, Int16 SLEN, Int
{
  Int16 dotpos;
  Int16 scale;
  Int16 top;
  Int16 max;
  Int16 mylen;
  Int16 sstop;
  Int16 ssmax;
  Char c;
  Int16 i;

  Char * P;
  Int16 offp;

  /* MUST CHECK THAT NOTHING OTHER THAN NUMERICS AND SINGLE "." */
  i = 0;
  while (i < SLEN)
    { c = *(SCRIPT+i);
      if ((c != '.')&&( (c < '0') || (c > '9')  ))
         { return -ErScPushNumberBad;
         };
      i ++;
    };

  top = beReadInt16(STACK+oTOP);
```

```
max = beReadInt16(STACK+oMAX);
if (top + XVI > max)
   { return -ErScPushNumberFull;                              /* stack full */
   };
dotpos = Advance(SCRIPT, SLEN, '.');
if (dotpos < 1)
   { scale = 0;
   } else
   { scale = SLEN - dotpos;
     SLEN --;                                  /* actual length one less d/t dot
   };
mylen = SLEN;

if (SLEN > 13)              /* only allow 13 for numerics, as need 1 byte for S
   { mylen = 15;                                /* signal a long string */
     sstop = beReadInt16(STACKSTRING+oTOP);
     ssmax = beReadInt16(STACKSTRING+oMAX);
     if (sstop + SLEN > ssmax)
        { return -ErScPushNumberFullLong;
        };
     P = STACKSTRING;
     offp = sstop;
     w_DmWrite(STACK, top, &SLEN, 2, SMAX);   /* keep record of length */
     w_DmWrite(STACK, top+2, &sstop, 2, SMAX);/* and maximum */
     sstop += SLEN;
     w_DmWrite(STACKSTRING, oTOP, &sstop, 2, MAXSS); /* new stackstring top*/
   } else                                       /* SHORT ITEM < 14 CHARS */
   { P = STACK;
     offp = top;
   };

if (dotpos < 1)
   { w_DmWrite(P, offp, SCRIPT, SLEN, 32000);
   } else
   { SLEN --;                                /* compensate for "." */
     w_DmWrite(P, offp, SCRIPT, dotpos-1, 32000);
     w_DmWrite(P, offp+dotpos-1, SCRIPT+dotpos, scale, 32000);
   };

c = scale;
w_DmWrite(STACK, top+13, &c, 1, SMAX);        /* scale */
c = mylen;
w_DmWrite(STACK, top+14, &c, 1, SMAX);        /* length of string */
c = 'N';                                       /* numeric datum */
w_DmWrite(STACK, top+15, &c, 1, SMAX);        /* type */
top += XVI;
w_DmWrite(STACK, oTOP, &top, 2, SMAX);        /* new stack top */
return 1;                                      /* success */
```

```
}
```

1.4.9 PushInteger

A revision to cater for use of a hash before a number to uniquely identify an integer. We read the submitted integer string (no prior hash as already identified and skipped over) and push it.

The ReadWriteInt routine can also be used elsewhere [FIX ME]!

```
Int16 ReadWriteInt (Char * STACK, Int16 top, Char * P, Int16 ilen)
{
  Int32 i;
  Int16 ng=0; // clumsy signal for negative
  Char c;

  if (*P == '-')
     { P ++;
       ilen --;
       ng = 1;
     };
  if (ilen > 9)
     { return -ScErEncodeIntegerMax;   // max is 999999999 = 9 digits
     };
  i = readint32(P,ilen);                    // ugggly
  if (ng)
     { i = -i;
     };

  w_DmWrite(STACK, top, (Char *)&i, 4, SMAX);
  // above byte order needs to be reversed on ARM microprocessor
  // we could check that the above succeeded
  // we might even fill bytes 4..13 with zeroes
  c = 'I';                                          // integer type
  w_DmWrite(STACK, top+15, &c, 1, SMAX);
  c = 4;                                            // length always 4
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  // DOES NOT ADJUST STACK
  return 1; // ok
}
```

Here's the actual PushInteger:

```
Int16 PushInteger (Char * STACK, Char * STACKSTRING, Char * SCRIPT, Int16 SLEN)
{ Int16 top;
  Int16 ok;
```

```
  top = beReadInt16(STACK+oTOP);
  ok = ReadWriteInt (STACK, top, SCRIPT, SLEN);
  if (ok < 0)
     { return PushNull(STACK); // fail BUT keep the faith by pushing a null!!
     };
  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        // new stack top
  return 1;                                     // ok
}
```

1.4.10 PushFloat

In a similar fashion to PushInteger, we read in a floating point number and push it to the stack!

```
Int16 PushFloat (Char * STACK, Char * SCRIPT, Int16 SLEN)
{ Int16 top;
  double d;
  Char c;

  if (textEncodeFloat((Char *)&d, SCRIPT, SLEN) < 0)
     { PushNull(STACK); // fail BUT keep the faith by pushing a null!!
       return -ErBaadFloat; // fail
     };
  top = beReadInt16(STACK+oTOP);

  w_DmWrite(STACK, top, (Char *)&d, 8, SMAX);
  // write standard float to stack (8 bytes)
  c = 'F';  // signal float
  w_DmWrite(STACK, top+15, &c, 1, SMAX);
  c = 8; // length always 8
  w_DmWrite(STACK, top+14, &c, 1, SMAX);

  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // new stack top
  return 1;                              // ok
}


/* ============================================================================
/* G. COMPLEX FUNCTIONALITY: STRING INTERCALATION AND DECODING.
*/

Int16 UnDate(Char * dst, Int16 dlen, char * srcp)
{  /* given SCRIPTINGYMMDD, convert to SCRIPTINGY-MM-DD */
   if (dlen < 10)
      { return 0;
```

```
      };
   xCopy (dst, srcp, 4);
   * (dst+4) = '-';
   xCopy (dst+5, srcp+4, 2);
   * (dst+7) = '-';
   xCopy (dst+8, srcp+6,2);
   return 10;
}
```

1.4.11 UnTime

Given HHMMSS convert to HH:MM:SS.

```
Int16 UnTime(Char * dst, Int16 dlen, char * srcp, Int16 ilen)
{
   if ((dlen < 8) || (ilen < 6))
      { return -ScErUnTime;        /* fail */
      };
   xCopy (dst, srcp, 2);
   * (dst+2) = ':';
   xCopy (dst+3, srcp+2, 2);
   * (dst+5) = ':';
   xCopy (dst+6, srcp+4,2);

   if (ilen == 6)        /* HHMMSS only */
      { return 8;
      };

/* // for now we suppress this component. Cf Timestamp too.
   * (dst+8) = '.';
   xCopy (dst+9, srcp+6,ilen-6);
   return (ilen+3);
*/
  return 8;
}
```

Return a -ve value on failure.

1.5 Unfloat

The following uses clumsy PalmOS routines. It could withstand a rewrite.

```
Int16 UnFloat (Char * dest, Int16 dlen, Char * srcp)
{
   FlpDouble f;
   Int16 flen;
   xCopy ( (Char *)&f, srcp, 8);
```

```
   if ( FlpFToA (f, dest) == errNone )
      { flen = (UInt16) StrLen(dest);
        return flen;
      };
   return -ScErFloat0;

}



/* UnNumeric
   given MMMMNNN with length of say 7 and scale of 3, we want "MMMM.NNN".
   in other words, copy over "length-scale" chars, then "." then the rest.
*/

Int16 UnNumeric(Char * dest, Int16 destlen, Char * ITM, Int16 ilen, Int16 scale)
{     /* I suppose could test for trivial ilen=dud, -ve scale ?? */
  if ((scale < 0) || (scale > MAXIMUMSCALE))
     { return -ScErNumSillyScale;
     };
  if (scale == 0)
     { if (destlen < ilen)
          { return -ScErNumericSpace;           /* no space */
          };
       xCopy(dest, ITM, ilen);
       return ilen;
     };
  if (ilen < scale)                             /* leading zeroes */
     { if (destlen < ilen+2)
          { return -ScErNumericSpace;           /* no space */
          };
       scale -= ilen;
       xCopy(dest, "0.", 2);
       dest += 2;
       xFill(dest, scale, '0');                 // put them in
       xCopy(dest+scale, ITM, ilen);            // and the rest
       return (2+scale+ilen);
     };

  if (destlen < ilen+1 )
     { return -ScErNumericSpace;                /* no space */
     };
  ilen -= scale;
  xCopy(dest, ITM, ilen);   /* what if error and -ve ?? */
  *(dest+ilen) = '.';
  xCopy(dest+ilen+1, ITM+ilen, scale);
  return (ilen+scale+1);
}
```

1.5.1 UnTimeStamp

Convert an encoded timestamp yyyyMMddHHmmSS into a timestamp “yyyy-MM-dd HH:mm:SS”.

```
Int16 UnTimeStamp(Char * dest, Int16 destlen, Char * ITM, Int16 ilen)
{ Int16 ok;
  ok = UnDate(dest, destlen, ITM);
  if (ok != 10)
     { return -ScErTimestamp1;
     };
  dest += 10;
  *(dest) = ' ';   // space between date and time
  dest++;
  destlen-=11;
  ok = UnTime(dest, destlen-11, ITM+8, ilen-8);
  if (ok != 8) // HH:MM:SS
     { return -ScErTimestamp2;
     };
  return 19; // anyway!
}
```

1.5.2 Decode

Given a pointer (ITM) to a stack-style item, which gives all the necessary formatting information, we translate the item into an ASCII string, returning the length of the translated string (after putting the ASCII into dest).

Normal return is with the length of the returned item. If a reasonable correction for an error is NULL, then Decode returns 0, otherwise -1.

```
Int16 Decode(UInt16 refnum, Char * dest, Int16 destlen, Char * STACKSTRING, Char *
{
  Char itype;
  Int16 ilen;
  Int16 soffset;
  Char * S;
  Int16 scale;  // [!!!!!!!! FIX ME]
  Int16 r; // actual return length

  if ((destlen < 0) || (! dest))
      { SayErr (refnum, ScBadDecode);
        return -1;
      };
```

```
  itype = *(ITM+15);         // item type
  ilen = 0x0F & * (ITM+14);  // pull out length
  scale = *(ITM+13);         // only relevance = numeric

  if (ilen > 14)  // is it a long string, referenced on STACKSTRING ?
     { soffset = beReadInt16(ITM+2);
       ilen = beReadInt16(ITM);
       S = STACKSTRING + soffset;

       if (ilen > destlen)
          {
            SayErr (refnum, ScErDecodeNoSpace);
            //WriteConsoleText(refnum, "{", 1, 0);
            WriteConsoleInteger(refnum, destlen, 0);
            WriteConsoleText(refnum, ";", 1, 0);
            WriteConsoleInteger(refnum, ilen, 0);
            WriteConsoleText(refnum, "=", 1, 0);
            WriteConsoleText(refnum, S, ilen, 0);
            //WriteConsoleText(refnum, "}", 1, 0);
            return 0;
          };

+OPTIONAL
   // debug type and length:
   WriteConsoleText(refnum, "%", 1,    fDEBUG_DUMP);
   WriteConsoleText(refnum, &itype, 1, fDEBUG_DUMP);
   WriteConsoleText(refnum, "(", 1,    fDEBUG_DUMP);
   WriteConsoleInteger(refnum, ilen,   fDEBUG_DUMP);
   WriteConsoleText(refnum, ")", 1,    fDEBUG_DUMP);
-OPTIONAL

       switch (itype)
         { case 'V':
           xCopy(dest, S, ilen);
           return ilen;

           case 'N':
             r = UnNumeric(dest, destlen, S, ilen, scale);
             if (r < 0)
                { SayErr (refnum,ScErDecodeNumeric);
                  return 0;
                };
             return r;

           case 'S':
             r = UnTimeStamp(dest, destlen, S, ilen);
             if (r < 0)
```

```
               { SayErr (refnum,ScErDecodeTimestamp);
                 return 0;
               };
             return r;

           case 'X':
             SayErr (refnum, ScErDecodeCompound);
             return 0; // NO recursion!

           // default: // note: types I,F,D,T cannot be long
           //  SayErr (refnum, ScErDecodeType);
           //  return 0;
         };
       SayErr (refnum, ScErDecodeType);
       return 0;
     };

  if (! ilen)  // permissible to have length of zero, return NULL
     { if (destlen < 4)
          { SayErr (refnum, ScErDecodeNoSpace);
            return 0;
          };
       xCopy(dest, "NULL", 4);
       return 4; // clumsy
     };
```

In all of the following, the item is contained in ITM (no STACKSTRING extension, length is under 14 bytes). The coding is clumsy, might be amalgamated with the above.

```
  if (ilen > destlen)
     {
       SayErr (refnum, ScErDecodeNoSpace);
       //WriteConsoleText(refnum, "{", 1, 0);
       WriteConsoleInteger(refnum, destlen, 0);
       WriteConsoleText(refnum, ";", 1, 0);
       WriteConsoleInteger(refnum, ilen, 0);
       WriteConsoleText(refnum, "=", 1, 0);
       WriteConsoleText(refnum, ITM, ilen, 0);
       //WriteConsoleText(refnum, "}", 1, 0);
       return 0;
     };

+OPTIONAL
   // debug type and length:
   WriteConsoleText(refnum, "%", 1,    fDEBUG_DUMP);
   WriteConsoleText(refnum, &itype, 1, fDEBUG_DUMP);
   WriteConsoleText(refnum, "(", 1,    fDEBUG_DUMP);
```

```
   WriteConsoleInteger(refnum, ilen,   fDEBUG_DUMP);
   WriteConsoleText(refnum, ")", 1,    fDEBUG_DUMP);
-OPTIONAL

  switch (itype)
    { case 'V':
        xCopy(dest, ITM, ilen);
        return ilen;

      case 'I':
        r = UnInteger(dest, destlen, ITM);
        if (r < 0)
           { SayErr (refnum,ScErDecodeInteger);
             return 0;
           };
        return r;

      case 'N':
        r = UnNumeric(dest, destlen, ITM, ilen, scale);  // [check me]
        if (r < 0)
           { SayErr (refnum,ScErDecodeNumeric);
             return 0;
           };
        return r;

      case 'T':
        r = UnTime(dest, destlen, ITM, ilen);
        if (r < 0)
           { SayErr (refnum,ScErDecodeTime);
             return 0;
           };
        return r;

      case 'D':
        if (ilen != 8)
           { SayErr (refnum, ScErDecodeDate);
             return 0;
           };
        r = UnDate(dest, destlen, ITM);
        if (r != 10)
           { SayErr (refnum, ScErDecodeDate);
             return 0;
           };
        return r;

      case 'S': // timestamp must be yyyyMMDDHHMMSS with no fraction
        r = UnTimeStamp(dest, destlen, ITM, ilen);
        if (r < 0)
```

```
           { SayErr (refnum,ScErDecodeTimestamp);
             return 0;
           };
         return r;

       case 'F':
         if (ilen != 8)
           { SayErr (refnum, ScErDecodeFloat);
             return 0;
           };
         r = UnFloat(dest, destlen, ITM);
         if (r < 0)
           { SayErr (refnum, ScErDecodeFloat);
             return 0;
           };
         return r;

       case 'X':
         SayErr (refnum, ScErDecodeCompound);
         return 0; // NO recursion!

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

1.5.3 eXtract

A rewrite of **RESOLVE**. Without altering STACK or STACKSTRING, take type X item and rewrite it to the *same memory space* (on stackstring). Return appropriate code (1=ok, -ve = errors) also convert the item to type V! This function is required to avoid including compound items *within* compound items!

The type X item has the usual format i.e:

+0 = total length;

+2 = offset of start of stackstring representation;

+4 = number of stack items (including insertion string);

+6 = offset of start of block of stored items ON STACKSTRING;

```
Int16 eXtract(UInt16 refnum, Char * STACK, Char * STACKSTRING, Int16 stkitm)
{  /* check item on stack is of type X */
```

```
  Char * TEMPBUFFER;  // will use to store extracted string, temporarily
  Int16 LENTOT;       // maximum length of tempbuffer.
  Int16 Xdest;        // where we will write resulting string onto stack!
  Char * dest;        // used to move around within TEMPBUFFER
  Int16 destlen;      // keep track of usage of TEMPBUFFER

  // IN ALL OF THE FOLLOWING WE MAKE THE UNFORTUNATE ASSUMPTION THAT
  //   THE SIZE OF THE RESULTING STRING IS <= THE SUM OF THE SIZES OF THE
  //   ITEMS TO BE RESOLVED. THIS IS ACTUALLY QUITE REASONABLE EXCEPT WHERE
  //   WE ARE USING FLOATING POINT NUMBERS [need to check up on this approach;
  //   perhaps pre-allocate extra space to a compound item which contains floating
  //   point numbers. [check me!]]

  Char * MAINSRC;
  Int16 mainitem;
  Int16 nextitem;
  Int16 insertions;
  Int16 mainlen;
  Int16 partlen;
  Char c;

  if ( *(STACK+stkitm+15) != 'X')
     { return -ScERnotX; // fail if not type X
     };
  LENTOT = 10+ *((Int16 *)(STACK+stkitm+0));  // length of compound (type X) item
  // arbitrary lengthening by 10 [??? check me]
  Xdest = *((Int16 *)(STACK+stkitm+2));   // start of X item on STACKSTRING
  insertions = *((Int16 *)(STACK+stkitm+4));
    // insertion count does NOT include the topmost item which is the insertion st
    //   containing the $[]'s to replace with resolved items.
  nextitem = beReadInt16 (STACK+stkitm+6);
  TEMPBUFFER = xNew(LENTOT); // hmm? perhaps add a smidgeon??
  dest = TEMPBUFFER;
  destlen = LENTOT;

  mainitem = nextitem + XVI*(insertions);  /* get offset in ss of main string ref
  mainlen = 0x0F & (* (STACKSTRING + mainitem + 14)); /* get main length */
  if (mainlen < 15)
     { MAINSRC = STACKSTRING + mainitem;             /* literal string is stored
     } else
     { mainlen = beReadInt16(STACKSTRING+mainitem); /* get true length*/
       MAINSRC = STACKSTRING + beReadInt16(STACKSTRING+mainitem+2); /* and source
     };

partlen = Advance(MAINSRC, mainlen, '$');

while ( (mainlen > 0) && (partlen > 0) )
  {
```

```
   if (destlen < partlen)
      { Delete(TEMPBUFFER);
        return -ScErNoSpace;
      };
   xCopy(dest, MAINSRC, partlen);                  /* for now, include the $ */
   dest += partlen;
   destlen -= partlen;                             /* track */
   MAINSRC += partlen;
   mainlen -= partlen;
   if (  (mainlen > 1)                             /* don't check if eg $ @ end of
       &&(* MAINSRC == '[')                        /* if $[] */
       &&(*(MAINSRC+1) == ']')
      )
      { dest --;                                   /* get rid of $ */
        destlen ++;
        MAINSRC += 2;
        mainlen -= 2;                              /* go past $[] */
        partlen = Decode(refnum, dest, destlen, STACKSTRING, STACKSTRING+nextitem
        if (partlen < 1)
           { Delete(TEMPBUFFER);
             return partlen;
           };
        dest += partlen;
        destlen -= partlen;
        if (destlen < 0)                          // is this necessary?? [check me]
           { Delete(TEMPBUFFER);
             return -ScErNoSpace;
           };
        nextitem += XVI;                           /* this item has been used */
        insertions --;                             /* REALITY CHECK */
        if (insertions < 0)
           { Delete(TEMPBUFFER);
             return -ScErBadInsertionCount;
           };
      };
   partlen = Advance(MAINSRC, mainlen, '$');
 };

if (insertions)                                    /* if nonzero, imbalance! */
   { Delete(TEMPBUFFER);
     return -ScErInsertImbalance;
   };
if (mainlen > 0)                                   /* if still stuff at end */
   { if (destlen < mainlen)
        { Delete(TEMPBUFFER);
          return -ScErNoSpace;
        };
     xCopy(dest, MAINSRC, mainlen);                /* copy the rest over */
```

```
      destlen -= mainlen;
    };

  // now change type to 'V' and copy back to stackstring!
  c = 'V';
  w_DmWrite(STACK, stkitm+15, &c, 1, SMAX);  // write type V
  c = 15;
  w_DmWrite(STACK, stkitm+14, &c, 1, SMAX);  // signal long item [always?!]

  // offset of start at +2 is unchanged, but length may well be altered
  // first, copy back:
  LENTOT -= destlen;  // get true length of new string:
  w_DmWrite(STACKSTRING, Xdest, TEMPBUFFER, LENTOT, MAXSS);
  w_DmWrite(STACK, stkitm, &LENTOT, 2, SMAX); // write new length
  // note: there may be some free space above end of new string on stackstring, bu
  // we must NOT try to recover this, or we will stuff up other compound pointers
  // waiting to be resolved.
  Delete(TEMPBUFFER);
  return 1; // ok.
}
```

1.5.4 TIMESTAMP

Here's the main TIMESTAMP routine.

```
Int16 EncodeTimestamp(UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int32 i;
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 iscale;
  Int16 itype;
  Int16 ssoff;
  Char * P;
  Char * S;
  Char c;
  double d;

  bottom = *((Int16 *)(STACK+oSTART));
  top = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { PushNull (STACK); // 'fix'
       SayErr (refnum,ScErEncodeTsEmpty);
       return 0;
```

```
   };
top -= XVI;
itype = *(STACK+top+15);
iscale = *(STACK+top+13);
ilen = 0x0F & *(STACK+top+14);

if (itype == 'S')
   { return 1;
   }; // success.

if (itype == 'X') // [nasty]
   { if ( eXtract(refnum, STACK, STACKSTRING, top) < 1)
          { WriteNull (STACK);
            WriteConsoleText(refnum, "?Ts", 3, 0);
            SayErr (refnum,ScErEncodeTsType);
            return 0;
          };
     itype = 'V';
   };

c=0x0;
if (ilen > 14)  // long string
   { ilen = beReadInt16(STACK+top);
     ssoff = beReadInt16(STACK+top+2);
     S = STACKSTRING+ssoff;
     w_DmWrite(STACKSTRING, oTOP, &ssoff, 2, MAXSS);
      // string used so trash it
   } else
   { S = STACK+top;
   };
if (ilen < 14)
   { P = xNew(14);  // more than enough
   } else
   { P = xNew(ilen+1+2); // see fixed pt usage
   };
// we will NOT at present allow fractional seconds
// in timestamp.


switch(itype)
  {
    case 'V':
      i = ReadTextTs(P, S, ilen);
      // also allows a simple date [assumes 00:00:00]!
      if (i < 1)
         { WriteNull (STACK);
           Delete(P);
           // SayErr (refnum,ScErTsLength);
```

```
            return 1; // don't force error msg!
          };
      break;


      case 'N':
      // at present an ugly hack: instead of using fixed point
      // arithmetic, we read the float and use our float rtns!
      // We first write the number to P (it is big enough, see above)
          ScaleNumeric(P, S, ilen,iscale);
          // convert number to float: [ugh]
          w_FlpBufferAToF((void *)&d,P);
          // really need error check in the above!

          // convert float to timestamp:
          Gregorian (refnum, P, &d, 0x03); // write timestamp
      break;

      case 'I':
          i = *((Int32 *)S); // watch out: ARM processor?
          d = i; // ugly float
          Gregorian (refnum, P, &d, 0x03); // write timestamp
      break;

      case 'T':  // time:
         WriteNull (STACK);
         Delete(P);
         return -1; // fail ? error msg.

      case 'D':  // date: [hmm??]
         xCopy(P, S, 8);
         xFill(P+8, '0', 6); // zero time
      break;

      case 'F':  // float:
         d= *((double *)S);
         Gregorian(refnum, P, &d, 0x03);
      break;

      default:
        Delete(P);
        WriteNull (STACK);
        SayErr (refnum,ScErEncodeTsType);
        WriteConsoleText(refnum, "?", 1, 0);
        WriteConsoleInteger(refnum, itype, 0);
        return 0;
    };
```

```
  c=14; // no partial seconds for now.
  w_DmWrite(STACK, top, P, c, SMAX);  // write stamp
  Delete(P);
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  c='S';
  w_DmWrite(STACK, top+15, &c, 1, SMAX); // nasty
  return 1;  // ok
}
```

1.6 ParseAndPush

This routine takes a string and turns it into a compound (type-X) data item. The idea is that we ultimately substitute in real values off the stack for each occurrence of `$[]` within the string.

```
/*
  The order is important. If we have:
  "flopsy"->"mopsy"->"cottontail"->"We are $[], $[] and $[]"

  then the order in the final string will be the same as reading from left to righ

  One major issue is that it's crazy to translate from a stack data type to ASCII,
  we're simply then going to have to translate back for the purposes of SQL

  My revised strategy (after much deliberation) is:
  0. Push the new string to the stack
  1. Find the number of insertions --- if none, we're done;
     Call this number of insertions "K-1"
  2. Determine Q, the first insertion with a length >14 i.e. with a presence on th
     stringstack. In your search, also look at the new string!
     2a. Record this stringstack offset, if present, otherwise
     2b. record offset of current top of stringstack.
  3. Copy all of the STACK items as a block onto the STRINGSTACK!
  4. Create a new stack item which signals that it's a compound of items:
     The type is 'X' to signal that it's a compound!
     at +0 : store length
     at +2 : Q, as in (2) above
     at +4 : NUMBER of stack items=K, [??? check me]->INCLUDING the string into wh
     at +6 : offset of start of block of stack items on STRINGSTACK!!
     (THE Kth item is the $[]-string into which the K-1 items will be inserted)

  THERE IS A PROBLEM if the storage of the K stack items on STRINGSTACK is not on
  an integral boundary divisible by 4, so we must ensure this!

  17/3/2005:
  WE HAVE A PROBLEM where one of the items being compounded into a type X item
```

```
  is itself type X. We cannot just ignore this. Should we resolve the item NOW
  or at the time of unwrapping with 'Resolve'?
  AWFULLY TEMPTING TO DO THE LATTER buuut it means multiple levels of resolution.
  We thus resolve at time of p&p.

*/

Int16 ParseAndPush (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * strg,
{
  Int16 insertions;
  Int16 scanlen;
  Char * S;
  Char c;
  Int16 ok;

  Int16 stackscan;
  Int16 sstop;
  Int16 newsstop;

  Int16 i;
  Int16 ilen;
  Int16 mystart;
  Int16 mylen;

  Int16 bottom;
  Int16 Xcheck;

  Int16 sgn = 0; //sign of number!
```

The problem of numerics

A real issue is numerics. We do not want to violate our general convention of using strings, but we *do* want to be able to say, for example, `RETURN(0)`. Ideally we should even be able to distinguish between a float, a fixed point number, and an integer. We will do so as follows [TENTATIVE]:

1. Integers will begin with a hash (pound) sign #
2. Floats will begin with a % percentage sign
3. Ordinary fixed point numerics will begin with + or -

We will set things up so that we can still submit strings which conform to the above specifications if we encase the string in quotes and submit it as e.g. `"#strg"->` that is, in line rather than as a parenthetic argument.

The following is similar to section 1.9.4.

```
c = * strg;
if (c == '#')
   {
     return PushInteger(STACK, STACKSTRING, strg+1, stlen-1);
   };
// [thought! should we only evaluate after checking for $[]'s ?]

if (c == '%')
   {
     return PushFloat(STACK, strg+1, stlen-1);
     // similar for floating point.
   };

if ((c == '-') || (c == '+'))
   { if (c == '-')
        { sgn = 1;
        }; // ugly
     c = * (strg+1);
     if ((c <= '9') && (c >= '0')) // hmm?
        { return PushNumber (STACK, STACKSTRING, strg+1, stlen-1, sgn);
        };
   };

/* 1. Count the number of $[]s  */
insertions = 0;
S=strg;
scanlen = stlen-2;                              /* no point in checking last 2 ch
while (scanlen > 0)
  { c= *S++;
    scanlen --;
    if (  (c == '$')                            /* scan along for "$[]" */
        &&(*S == '[')
        &&(*(S+1) == ']')
       )
       { S+=2;
         scanlen -= 2;
         insertions ++;
       };
  };

ok = PushItem (0, STACK, STACKSTRING, strg, stlen, 'V', 0);
if (ok < 0) { return ok; };              /* push failed */
if (! insertions)
   { return 1;                                  /* success */
   };
```

```
insertions ++;                                  /* +1 for the pushed $[] string *
stackscan = *((Int16 *)(STACK+oTOP));           /* get current top : after last i
stackscan -= XVI * insertions;                   /* move to first item */

bottom = *((Int16 *)(STACK+oSTART));            // ensure enough items!
if (stackscan < bottom)
   {
     return -ScErNotEnoughData;
   };

/* ensure that sstop is on an integral boundary divisible by 4 ! */
sstop = *((Int16 *)(STACKSTRING+oTOP));
sstop += 3;
sstop &= 0xFFFC;
mystart = sstop; // default, if no items!

// ------------------here make sure we aren't pushing type X items! ------------
// as a bonus, we also find FIRST long item (if present) and retain this as 'mys
// what if there's none???

Xcheck = stackscan;
i = insertions;
while (i > 0)
  {
    ilen = 0xF & (*(STACK+Xcheck+14));      // get recorded length
    if (   (ilen > 14)
        && (mystart == sstop)  // ONLY USE IF 1st NOT ALREADY IDENTIFIED!
       )
       { mystart = *((Int16*)(STACK+Xcheck+2)); // get stackstring offset
       };
    if (*(STACK+Xcheck+15) == 'X') // if contained type X item:
       { ok = eXtract(refnum, STACK, STACKSTRING, Xcheck); // convert it to type
         if (ok < 1)
            { return ok;
            };
       };
    i --;
    Xcheck += XVI;
  };

// ----------------------------end check for type X --------------------------


/* NEXT, COPY OVER WHOLE CHUNK FROM STACK TO STACKSTRING!!! */
if (! w_DmWrite(STACKSTRING, sstop, STACK+stackscan, insertions*XVI, MAXSS))
   { return -ScErCompoundNoFit;                                  /* failed to writ
   };
newsstop = sstop + insertions*XVI;
```

```
if (! w_DmWrite(STACKSTRING, oTOP, &newsstop, 2, MAXSS))
   { return -ScErCompoundBadSsTop;                                  /* failed to w
   };

/* FINALLY, ESTABLISH NEW, COMPOUND ITEM ON STACK ! */
/* We must record:
   0. type of item, and 'size>14'
   a. at +0 : length (total) of compound item
   b. at +2 : offset of very first item on stringstack (mystart)
   c. at +4 : insertions (includes final item)
   d. at +6 : sstop, which records start of chunk of items!
   THE FOLLOWING IS VERY SIMILAR TO PushItem, but note that as we have dumped
   all relevant items on stack, the new stack item goes in at stackscan!

   [fixme: !!! must check that we didn't pop too many items off stack !!!]
*/

c = 'X';                                        /* item type */
if (! w_DmWrite(STACK, stackscan+15, &c, 1, SMAX))
   { return -ScErCompoundStackWrite;                                 /* stack wri
   };
c = 15;                                         /* as usual, signal large size */
if (! w_DmWrite(STACK, stackscan+14, &c, 1, SMAX))
   { return -ScErCompoundStackWriteBig;                                 /* stack
   };
/* the above can be written far more succinctly */

mylen = newsstop - mystart;                     /* size of compound ss item */
if (! w_DmWrite(STACK, stackscan+0, &mylen, 2, SMAX)) /* [CHECK ME??? WHAT IF AR
   { return -ScErCompoundBadLen;                                    /* stack write c
   };
if (! w_DmWrite(STACK, stackscan+2, &mystart, 2, SMAX))
   { return -ScErCompoundBadOffset;                                   /* failed to
   };
insertions --;   /* don't count 'insertee' in final (stored) count */
if (! w_DmWrite(STACK, stackscan+4, &insertions, 2, SMAX))
   { return -ScErCompoundInsertions;                                   /* failed to
   };
if (! w_DmWrite(STACK, stackscan+6, &sstop, 2, SMAX))
   { return -ScErCompoundStart;                                  /* failed to writ
   };

/*  ******************************** NB ********************************
    We really run into trouble if we SWOP around type 'X' compound items!
      [ check me; fix me]
    ********************************************************************
*/
```

```
  stackscan += XVI;                                /* new STACK top */
  if (! w_DmWrite(STACK, oTOP, &stackscan, 2, SMAX))
     { return -ScErCompoundTop;                                /* failed write of n
     };
  return 1;
}
```

1.6.1 JOIN/LIST

Take all items off stack (down to start) and turn them into a single string. AT PRESENT no item may be unresolved type X.

LIST is almost identical to JOIN. The only difference is that LIST places a copy of the separator at the end. [Nasty]

We submit a delimiting string on the top of the stack. When the items are concatenated from the stack, the topmost item is used as 'glue' to join all of the others!

We will simply:

1. count number of stack items
2. create a string `"$[] $[] .. $[] "` that will ultimately contain the list
3. Invoke ParseAndPush to encapsulate the lot into a type-X stack item!
4. clean up.

NB. If nothing on stack, then return NULL string!! We must also *not* put the separator string at the end, it really is only a separator!

```
Int16  Stack2String(UInt16 refnum, Char * STACK, Char *  STACKSTRING, Int16 islist
{
  Int16 bottom;
  Int16 top;
  Int16 items;
  Int16 i;
  Char * topstring;
  Char * tP;
  Int16 ilen;
  Int16 ok;
  Char c;
  Char * sep;   //separator
  Int16 seplen; // and its length

  bottom = *((Int16 *)(STACK+oSTART));
```

```
top    = *((Int16 *)(STACK+oTOP));

// 0. first, get separator!
top -= XVI;
if (top < bottom)
   { return -ScListEmpty;
   };
if (*(STACK+top+15) != 'V')
   { // [here might still allow any NULL!]
     return -ScListSeparator;  // bad list separator, it MUST be a string!
                               // hmm what if unresolved type X ???[fix me]
   };
seplen = 0x0F & *(STACK+top+14); // get length of separator
if (seplen > 14)
   { seplen = *((Int16*)(STACK+top));  // might check this isn't completely sill
     sep = STACKSTRING + *((Int16*)(STACK+top+2));
   } else
   { sep = STACK+top;
   };
w_DmWrite(STACK, oTOP, &top, 2, SMAX); // clean up stack BEFORE ParseAndPush!

/* 1. get number of stack items */
items = (top - bottom)/XVI;
if (items < 0)
   { return -ScListEmpty;                              /* fail: bad stack */
   };
if (! items)                                    /* if no items, return a null str
   { c=0;
     w_DmWrite(STACK, top+14, &c, 1, SMAX);   /* write null length item to stack
     c='V';
     w_DmWrite(STACK, top+15, &c, 1, SMAX);   /* clumsy */
     top += XVI;
     w_DmWrite(STACK, oTOP, &top, 2, SMAX);   /* push to stack */
     return 1;
   }; // [shorten the above to a call to writing of NULL]

/* 2. create intercalation string, items* size of "$[]<separator>"  */
ilen = items*(3+seplen);  // 3 characters for "$[]" and the rest for the separat
if (! islist)
   { ilen -= seplen;
   }; // clumsy hack.

topstring = xNew(ilen);   // there is *no* TERMINAL separator!
if (! topstring)
   { return -ScErListFull;
   };
tP = topstring;
i = items-1;
```

```
  while (i > 0)
    { xCopy(tP, "$[]", 3);                          /* clumsy */
      xCopy(tP+3, sep, seplen);
      tP += 3+seplen;
      i --;
    };
  xCopy(tP, "$[]", 3);
  tP += 3;
  if (islist)
     { xCopy(tP, sep, seplen); // terminal item if LIST (not for JOIN)
     };

  /* 3. ParseAndPush (this is far from elegant)
        better would be to rewrite P&P to call a subrtn which we
        then call, supplying number, rather than re-counting [BMJ hah: recounting]
  */
  ok = ParseAndPush(refnum, STACK, STACKSTRING, topstring, ilen);
  Delete(topstring);
  return ok;
}
```

1.6.2 Cleanup

We must examine the following routine with care, and also look at similar functionality elsewhere.

We submit the ‘itm’ offset on the STACK, and trim both the stack and STACKSTRING down appropriate to the removal of this item. The offset is to the *start* of the item on our stack.

```
Int16 Cleanup (Char * STACK, Char * STACKSTRING, Int16 itm)
{ Int16 sstop;
  if ((0x0F & *(STACK+itm+14)) > 14)
     { sstop = beReadInt16(STACK+itm+2);
       // if we were obsessive, we might check that sstop+length=current top
       w_DmWrite(STACKSTRING, oTOP, &sstop, 2, MAXSS);
     };
  w_DmWrite(STACK, oTOP, &itm, 2, SMAX);   // revise top down
  return 1;
}
```

1.7 Resolving a string

In a similar fashion to Perl, we permit an inline placeholder (at present only $[]) to be replaced by a string taken off the stack. The Resolve function is the guts of this functionality. In other words we ‘unfold’ ParseAndPush.

1. If a simple item, decode it to destination and exit;
2. Identify the main ($[]-containing) string, and start writing it to output;
3. Locate the first of the compound items on stackstring;
4. For each $[] in the main string, substitute in an ASCII TRANSLATION of the item; move up on the stackstring to the next item;

Resolve should return -1 for 'utter failure' and 0 if it is reasonable to 'try to fix things' by interpreting the returned value as NULL.

We first check for a type-X item, and if not present, simply Decode the value.

```
Int16 Resolve(UInt16 refnum, Char * dest, Int16 destlen, Char * STACK, Char * STAC
{
  Char * MAINSRC;
  Int16 stkitem;
  Int16 mainitem;
  Int16 nextitem;
  Int16 insertions;
  Int16 mainlen;
  Int16 partlen;
  Int16 keeplen;
  Int16 stkbot;
  Int16 ok;

  keeplen = destlen;

  // pop stack and obtain item
  stkbot = *((Int16*)(STACK+oSTART));
  stkitem = -XVI + *((Int16*)(STACK+oTOP));
  if (stkitem < stkbot)
     { SayErr(refnum, ScErResolveEmptyStack); // stack is empty
       return 0;
     };
  if ( *(STACK+stkitem+15) != 'X')
     {
       ok = Decode(refnum, dest, destlen, STACKSTRING, STACK+stkitem);
       Cleanup(STACK, STACKSTRING, stkitem);   // takes 1 item off stack +- stacks
       if (ok < 0)
          { return 0; // fail [NULL?]
          };
       return ok; // return length of string written.
     };
```

Cleanup *only* alters the stack pointers, doesn't affect actual data, although we should avoid using this fact.

The following processes a compound type:

```
  WriteConsoleText(refnum, "#X", 2, fDEBUG_DUMP); // debug

  insertions = beReadInt16 (STACK+stkitem+4);
  nextitem = beReadInt16 (STACK+stkitem+6);
  mainitem = nextitem + XVI*(insertions);   // get offset in ss of main string ref

  // might check that *(STACKSTRING+mainitem+15==’V’) //
  mainlen = 0x0F & (* (STACKSTRING + mainitem + 14));  // get main length
  if (mainlen < 15)
     { MAINSRC = STACKSTRING + mainitem;              // literal string is stored
     } else
     { mainlen = beReadInt16(STACKSTRING+mainitem);  // get true length
       MAINSRC = STACKSTRING + beReadInt16(STACKSTRING+mainitem+2);  // and source
     };

partlen = Advance(MAINSRC, mainlen, ’$’);

while ( (mainlen > 0) && (partlen > 0) )
  {
    if (destlen < partlen)
       {
         Cleanup(STACK, STACKSTRING, stkitem);
         SayErr(refnum, ScErResolveSpace); // no space
         return -1;
       };
    xCopy(dest, MAINSRC, partlen); // for now, include the $
    dest += partlen;
    destlen -= partlen;            // track
    MAINSRC += partlen;
    mainlen -= partlen;
    if (  (mainlen > 1)            // don’t check if eg $ @ end of string
        &&(* MAINSRC == ’[’)       // if $[]
        &&(*(MAINSRC+1) == ’]’)
       )
       { dest --;                  // get rid of $
         destlen ++;
         MAINSRC += 2;
         mainlen -= 2;                                 // go past $[]
         partlen = Decode(refnum, dest, destlen, STACKSTRING, STACKSTRING+nextitem
         if (partlen < 1)
            { return (partlen);              // fail
            };
         dest += partlen;
         destlen -= partlen;
         nextitem += XVI;                               // this item has been used
         insertions --;                                // REALITY CHECK
         if (insertions < 0)
            {
```

```
              Cleanup(STACK, STACKSTRING, stkitem); ///
              SayErr(refnum, ScErResolveCount); // bad insertn count??
              return -1;
            };
       };
    partlen = Advance(MAINSRC, mainlen, '$');
  };

 if (insertions)
    {
      Cleanup(STACK, STACKSTRING, stkitem); ///
      SayErr(refnum, ScErResolveInsertions); // imbalance!
      return -1;
    };
 if (mainlen > 0)  // if still stuff at end
    { if (destlen < mainlen)
         {
           Cleanup(STACK, STACKSTRING, stkitem); ///
           SayErr(refnum, ScErResolveFull); // no space
           return -1;
         };
      xCopy(dest, MAINSRC, mainlen); // copy rest over
      destlen -= mainlen;
    };
 Cleanup(STACK, STACKSTRING, stkitem);
 // as compound item references whole length, Cleanup should work fine
 return (keeplen - destlen);  // return No of bytes written
}


/* ============================================================================
/* I. LOGICAL OPERATIONS */

/*----------------------------------------------------------------------------
/* BOOLEAN:
   The original perl accepted "null OR 0 OR 'F' or 'false'" as zero, otherwise 1.
*/

Int16 testboolean (Char * item)
{ Char * myfalse = "FALSE";                    /* could make this const */
  Char t;
  Int16 ilen;
  Int32 i;
  ilen = 0x0F & (*(item+14));                  /* length */
  if (ilen == 0)
     { return 0;                               /* result is zero for any NULL */
     };
  t = *(item+15);                              /* get item type */
  switch (t)
```

```
    {
      case 'I':                                   /* integer */
        i = *((Int32 *)(item));
        if (!i)
           { return 0;                            /* return 0 if i=0 */
           };
        return 1;

      case 'N':
        if (ilen > 1)
           { return 1;                            /* cannot be 0 if length > 1 */
           };
        if (*item == '0')
           { return 0;                            /* zero --> return 0 */
           };
        return 1;

      case 'V':                                   /* character */
        if (ilen == 1)
           { if (  (UPPERCASE(*item) == 'F')
                 ||(*item == '0')
                )
                {return 0;
                };
             return 1;
           };
        if (ilen == 5)
           { if (LUPSAME (item, 5, myfalse, 5))   /* is it text "false" or "FALSE"
                { return 0;
                };
           };
        return 1;

      default:
        return 1;                                 /* D,T,S all non-0 */
        /* hmm; What about a float of "zero"? NO! Same old equality(epsilon) argum
    };
  return 1;                                       /* not taken */
};



Int16 MyBoolean (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char c;
  Int32 i;
```

```
  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErBooleanEmpty;                              /* fail: nothing o
     };
  top -= XVI;

   /* examine item */
  i = testboolean (STACK+top);

  /* reformat top stack item as type I [fix me: use writeinteger] */
  c = 'I';
  w_DmWrite(STACK, top+15, &c, 1, SMAX);        /* write type as I */
  c = 4;
  w_DmWrite(STACK, top+14, &c, 1, SMAX);        /* write length as 4*/
  w_DmWrite(STACK,top, &i, 4, SMAX);            /* write answer */
  return 1;
}




/*-------------------------------------------------------------------------------
/* AND:

  Logical AND between two integers.
  Only returns 1 if both are 1.
  If either is not an integer, FORCE ERROR. No sloppy logic!
  Does NOT perform logical bitwise AND between numbers!
  [fix me: create separate fx for this]
*/

Int16  AndLogic (Char * STACK)
{
  Int16 bottom;
  Int16 top;
  Int32 i;
  Int32 i2;
  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     { return -ScErAndEmpty;                    /* fail: insufficient stack */
     };

  if (  (*(STACK+top+15) != 'I')
```

```
      ||(*(STACK+top-XVI+15) != 'I')
     )
     { return -ScErAndArgs; // should also force NULL to stack!
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        /* pop just one item */
  i = *((Int32 *)(STACK+top));
  i2= *((Int32 *)(STACK+top-XVI));
  if (i == 1)
     { if (i2 == 1)
          { return 1;                            /* success; leave 1 on stack */
          };
       if (i2 != 0)
          { return -ScErAndArgs2;                /* FAIL!! must be 1/0 arg! */
          };
     } else
     { if (i != 0)
          { return -ScErAndArgs2;
          };
       if ((i2 != 0)&&(i2 != 1))
          { return -ScErAndArgs2;
          };
     };
  i = 0;
  w_DmWrite(STACK, top-XVI, &i, 4, SMAX);       /* success; 0 on stack */
  return 1;
}



/*-------------------------------------------------------------------------------
/* OR:

 Very similar to And (above)
 Same insistence on integer 1/0 being present!
*/

Int16  OrLogic (Char * STACK)
{
  Int16 bottom;
  Int16 top;
  Int32 i;
  Int32 i2;
  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     { return -ScErOrEmpty;                     /* fail: insufficient stack */
     };
```

```
  if (  (*(STACK+top+15) != 'I')
      ||(*(STACK+top-XVI+15) != 'I')
     )
     { return -ScErOrArgs;
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        /* pop just one item */
  i = *((Int32 *)(STACK+top));
  i2= *((Int32 *)(STACK+top-XVI));               /* i2 is the deeper one */

  if (i2 == 1)
     { if ((i == 0) || (i == 1))
          { return 1;                            /* success, leave 1 on stack */
          };
       return -ScErOrArgs2;
     };
  if (i2 != 0)
     { return -ScErOrArgs2;
     };
  if (i == 0)
     { return 1;                                /* success; leave 0 on stack */
     };
  if (i != 1)
     { return -ScErOrArgs2;
     };
  w_DmWrite(STACK, top-XVI, &i, 4, SMAX);       /* success; 1 on stack */
  return 1;
}



/*-------------------------------------------------------------------------------
/* NOT:

  Similar to And, Or. one argument.
*/

Int16  NotLogic (Char * STACK)
{
  Int16 bottom;
  Int16 top;
  Int32 i;
  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErNotEmpty;                     /* fail: insufficient stack */
     };
  top -= XVI;
```

```
  if (*(STACK+top+15) != 'I')
     { return -ScErNotArgs;
     };
  i = *((Int32 *)(STACK+top));

  if ((i != 0) && (i != 1))
     { return -ScErNotArgs2;
     };
  i ^= 1;                                          /* exclusive or (xor) to toggle b
  w_DmWrite(STACK, top, &i, 4, SMAX);             /* success; 1 on stack */
  return 1;
}
```

1.7.1 Greater

```
/*-------------------------------------------------------------------------------
/* Greater:
   // AMENDED 11/3/2005:
   In keeping with SUB, GREATER must compare the lower item with the topmost stack
   and return true if the lower item is greater!
   at present limited to integer, float
   we will insert fixed point arithmetic comparison too
   WE must consider string comparisons (YES [!!! FIX ME !!!]
*/

Int16  DoGreater (UInt16 refnum, Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Char c;
  Int32 i=0;

  double d;
  double d2;

  Char * P1;
  Char * P2;
  Int16 L1;
  Int16 L2;

  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
```

```
    { return -ScErGtEmpty;                    /* fail: insufficient stack */
    };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);      /* pop just one item */

  t = *(STACK+top+15);                         /* get type of topmost number */
  switch (t)
    { case 'I':
        if (*(STACK+top-XVI+15) != 'I')
           { return -ScErGtArgs;
           };
        i = *((Int32 *)(STACK+top-XVI));       /* get deeper value */
        i -= *((Int32 *)(STACK+top));          /* subtract top stack item */
        if (i > 0)
           { i = 1;                            /* 1 signals YES, greater */
           } else
           { i = 0;
           };
        break;

      case 'F':
        if (*(STACK+top-XVI+15) != 'F')
           { return -ScErGtArgs;
           };
        xCopy ((Char *)&d, STACK+top-XVI, 8);        /* deeper item */
        xCopy ((Char *)&d2, STACK+top, 8);    /* upper item */
        if (d > d2)  // this is clumsy. Can simply use byte-by-byte comparison as
           { i = 1;
           } else
           { i = 0;
           };
        c = 'I';                               /* result must be integer */
        w_DmWrite(STACK, top-XVI+15, &c, 1, SMAX);
        c = 4;                                 /* length is 4 */
        w_DmWrite(STACK, top-XVI+14, &c, 1, SMAX);
        break;

      case 'N':
        if (*(STACK+top-XVI+15) != 'N')
           { return -ScErGtArgs;
           };
           /* WILL INCLUDE APPROPRIATE ROUTINE HERE */
           return -ScErGtArgs;
        /* break; */

      case 'V':  // string ???
         // default i=0 (above)
         L1 = 0x0F & *(STACK+top+14); // length of top item
         L2 = 0x0F & *(STACK+top-XVI+14); // bottom
```

```
         if (L1 > 14)
            { P1 = STACKSTRING + * ((Int16 *)(STACK+top+2));
              L1 = * ((Int16 *)(STACK+top));
            } else
            { P1 = STACK+top;
            };
         if (L2 > 14)
            { P2 = STACKSTRING + * ((Int16 *)(STACK+top+2-XVI));
              L2 =  * ((Int16 *)(STACK+top-XVI));
            } else
            { P2 = STACK+top-XVI;
            };
  // debug console write:
  +OPTIONAL
   WriteConsoleText(refnum, "[", 1, fDEBUG_DUMP);  // debug only
   WriteConsoleText(refnum, P2, L2, fDEBUG_DUMP);
   WriteConsoleText(refnum, ":", 1, fDEBUG_DUMP);  //
   WriteConsoleText(refnum, P1, L1, fDEBUG_DUMP);
   WriteConsoleText(refnum, "]", 1, fDEBUG_DUMP);  //
   -OPTIONAL
         if ( BetterCompare(P2, L2, P1, L1) > 0 )
            { i = 1;
            };
   +OPTIONAL
    WriteConsoleText(refnum, "->", 2, 0);
    WriteConsoleInteger(refnum, i, 0);
   -OPTIONAL
        c = 'I';    // result must be integer
        w_DmWrite(STACK, top-XVI+15, &c, 1, SMAX);
        break;


      // case 'X': // ??? what about this?? (or 'resolve' above)



      default:
        return -ScErGtArgs;
    };

  w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
  return 1;
}
```

1.7.2 Less

```
/*-------------------------------------------------------------------------
/* Less
   very similar to DoGreater.
*/
Int16  TestLess (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Char c;
  Int32 i=0;

  double d;
  double d2;

  Char * P1;
  Char * P2;
  Int16 L1;
  Int16 L2;

  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     { return -ScErGtEmpty;                    /* fail: insufficient stack */
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);       /* pop just one item */

  t = *(STACK+top+15);                          /* get type of topmost number */
  switch (t)
    { case 'I':
        if (*(STACK+top-XVI+15) != 'I')
           { return -ScErGtArgs;
           };
        i = *((Int32 *)(STACK+top-XVI));        /* get deeper value */
        i -= *((Int32 *)(STACK+top));           /* subtract top stack item */
        if (i < 0)
           { i = 1;                             /* 1 signals YES, greater */
           } else
           { i = 0;
           };
        break;

      case 'F':
        if (*(STACK+top-XVI+15) != 'F')
           { return -ScErGtArgs;
           };
```

```
      xCopy ((Char *)&d, STACK+top-XVI, 8);        /* deeper item */
      xCopy ((Char *)&d2, STACK+top, 8);    /* upper item */
      if (d < d2)
         { i = 1;
         } else
         { i = 0;
         };
      c = 'I';                                  /* result must be integer */
      w_DmWrite(STACK, top-XVI+15, &c, 1, SMAX);
      c = 4;                                    /* length is 4 */
      w_DmWrite(STACK, top-XVI+14, &c, 1, SMAX);
      break;

    case 'N':
      if (*(STACK+top-XVI+15) != 'N')
         { return -ScErGtArgs;
         };
         /* WILL INCLUDE APPROPRIATE ROUTINE HERE */
         return -ScErGtArgs;
      /* break; */

    case 'V':  // string ???
       // default i=0 (above)
       L1 = 0x0F & *(STACK+top+14); // length of top item
       L2 = 0x0F & *(STACK+top-XVI+14); // bottom
       if (L1 > 14)
          { P1 = STACKSTRING + * ((Int16 *)(STACK+top+2));
            L1 = * ((Int16 *)(STACK+top));
          } else
          { P1 = STACK+top;
          };
       if (L2 > 14)
          { P2 = STACKSTRING + * ((Int16 *)(STACK+top+2-XVI));
            L2 =  * ((Int16 *)(STACK+top-XVI));
          } else
          { P2 = STACK+top-XVI;
          };
       if ( BetterCompare(P2, L2, P1, L1) < 0 )
          { i = 1;
          };
      c = 'I';    // result must be integer
      w_DmWrite(STACK, top-XVI+15, &c, 1, SMAX);
       break;

    default:
      return -ScErGtArgs;
  };
```

```
  w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
  return 1;
}



/*-------------------------------------------------------------------------------
/* ISNUMBER
   This is tricky : do we also 'testnumber' if the argument is a string ???

*/

Int16 cleanstackstring (Char * STACK, Char * STACKSTRING, Int16 itm)
{ Int16 ilen;
  Int16 ioff;
  ilen = 0x0F & *(STACK+itm+14);
  if (ilen < 15)
     { return 1;
     };
  ioff = *((Int16 *)(STACK+itm+2));
  /* here might check that length+offset = current stackstring top */
  w_DmWrite(STACKSTRING, oTOP, &ioff, 2, MAXSS); /* remove string from top */
  return 1;                                      /* clumsy: 'success' */
}

Int16  TestNumber (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char c;
  Int16 i;

  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErTestnumEmpty;                    /* fail: insufficient stack *
     };
  top -= XVI;
  c = *(STACK+top+15);
  if (  (c == 'I')
      ||(c == 'N')
      ||(c == 'F')
     )
     { i = 1;
     } else
     { i = 0;
     };
  cleanstackstring(STACK, STACKSTRING, top);
```

```
  writeinteger(STACK, top, i);
  return 1;
}



/*-------------------------------------------------------------------------------
/* ISNULL:
*/

Int16  TestNull (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 i;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErTestNullEmpty;               /* fail: insufficient stack */
     };
  top -= XVI;
  ilen = 0x0f & *(STACK+top+14);
  if (ilen == 0)
     { i = 1;
     } else
     { cleanstackstring(STACK, STACKSTRING, top); /* could .. if ilen > 14 */
       i = 0;
     };
  writeinteger(STACK, top, i);  // need not alter oTOP as are reading and writing
  return 1;
}
```

1.7.3 Same

```
Int16 sameness (Char * STACK, Char * STACKSTRING, Int16 itmA, Int16 itmB)
{ /* return 1 iff identical, else 0  */
  Char t;
  Int16 ilen;
  Char * Pa;
  Char * Pb;

  t = *(STACK+itmA+15);
  if (t != *(STACK+itmB+15))
     { return 0;                                /* different types */
```

```
     };
  ilen = 0x0F & *(STACK+itmA+14);
  if (ilen != (0x0F & *(STACK+itmB+14)))
     { return 0;                                  /* different lengths */
     };
  if (ilen < 15)
     { return xSame (STACK+itmA, ilen, STACK+itmB, ilen);
     };
  ilen = *((Int16 *)(STACK+itmA+0));
  if (ilen != *((Int16 *)(STACK+itmB+0)))
     { return 0;                                  /* different lengths, again */
     };
  Pa = STACKSTRING + *((Int16 *)(STACK+itmA+2));
  Pb = STACKSTRING + *((Int16 *)(STACK+itmB+2));
  return (xSame(Pa, ilen, Pb, ilen));
}

Int16  TestSame (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 i;

  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     { return -ScErGtEmpty;                     /* fail: insufficient stack */
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        /* pop just one item */

  i = sameness (STACK, STACKSTRING, top, top-XVI);

  top -= XVI;
  writeinteger(STACK, top, i);
  return 1;
}
```

1.7.4 Any

This routine checks the stack — if there's anything at all above the last mark, then it returns true (1), otherwise false (0).

```
Int16 AnyTest (Char * STACK)
{
  Int16 bottom;
  Int16 top;
```

```
  Int16 i=0;

  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top > bottom)
     { i = 1;
     };
  return writepushinteger(STACK, i);
}



/* ================================================================================
/* J. ARITHMETIC OPERATIONS: */
// [moved to NUMERIC library]



/* ================================================================================
/* K. FLOW CONTROL AND STACK MODIFICATION*/


/*--------------------------------------------------------------------------------
/* SKIP:
   Will skip iff boolean (integer) 1 is on stack. Anything else --> no skip.
*/

Int16 MySkip(Char * STACK, Char * STACKSTRING)
{ /* only skip if INTEGER 1 is on stack ?! */
  /* WHAT ABOUT numeric 1 ? See BOOLEAN*/
  /* hmm being a bit anal ? */

  Int16 bottom;
  Int16 top;
  Int32 i;
  Int16 Stop;

  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErSkipEmpty;                   /* fail: nil on stack */
     };

  top -= XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);       /* pop item */

  if (*(STACK+top+15) != 'I')
     { if ( (0x0F & *(STACK+top+14)) > 14)
          { Stop = *((Int16 *)(STACK+top+2));  // pop item off stackstring
```

```
            w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS); // new stack top!
            // might even check whether value makes sense!
          };
        return 1;                                   /* just continue, don't skip */
        /* might even warn about bad type ? */
      };
  i = *((Int32 *)(STACK+top));                      /* pull out actual integer */
  if (i != 1)
     { return 1;
     };
  return iSKIP;                                     /* only skip if integer 1 ! */
}
```

1.7.5 MyCopy

The following 'simply' copies the top item on the stack, *in toto*.[5] If sayerr is set, then a null stack results in an error message being written.[6]

```
Int16  MyCopy (UInt16 refnum, Char * STACK, Char *  STACKSTRING, Int16 sayerr)
{ Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 ioffset;
  Int16 ssmax;
  Int16 max;

  bottom = *((Int16 *)(STACK+oSTART)); // ? beReadInt16
  top    = *((Int16 *)(STACK+oTOP)); // [watch out: ?ARM processor]
  max    = *((Int16 *)(STACK+oMAX));
  if (top <= bottom)
     { if (sayerr || (top < bottom))
          { SayErr (refnum,ScErCopyEmpty);
          };
       return -1;
     };
  if (top+XVI > max)
     { SayErr (refnum,ScErCopyOverflow);
       return -1;
     };

  w_DmWrite(STACK, top, STACK+top-XVI, XVI, SMAX); // 16 bytes
  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // stack top
```

[5] CHECK ME: what if a compound item i.e. type X? Seems OK.

[6] Clearing sayerr is our way of preventing misleading error messages being written during debugging.

```
  // If long item, must copy stackstring too:
  ilen = 0x0F & *(STACK+top-XVI+14); // length
  if (ilen > 14)
     { ilen = *((Int16 *)(STACK+top-XVI)); // actual length
       ioffset = *((Int16 *)(STACK+top-XVI+2));
       ssmax = *((Int16 *)(STACKSTRING+oMAX));
       if (ioffset+2*ilen > ssmax) // generous
          { SayErr (refnum,ScErCopyFull);
            return -1;
          };
       w_DmWrite(STACKSTRING, ioffset+ilen, STACKSTRING+ioffset, ilen, MAXSS);
       ioffset += ilen;
       w_DmWrite(STACK, top-XVI+2, &ioffset, 2, SMAX);
         // update copied item.
       ioffset += ilen;
       w_DmWrite(STACKSTRING, oTOP, &ioffset, 2, MAXSS);
         // new stackstring top!
     };
  return ilen; // success
}
```

In the above we return the actual length of the item! On failure, MyCopy returns -1, as zero can indicate a null item (length zero)!

1.7.6 Discard

We discard a single item from the stack.

```
/*-------------------------------------------------------------------------------
Int16  Discard (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Int16 ioff;

  /* standard pop: */
  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top < bottom)
     { return -ScErDiscardEmpty;                /* fail: insufficient stack */
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);       /* pop just one item */
  ilen = 0x0F & *(STACK+top+14);
  if (ilen > 14)                                /* if long string, trash this too
```

```
     { ioff = *((Int16 *)(STACK+top+2));         /* get offset of start of long st
       /* could here also check that ioff+ilen = current stackstring top */
       w_DmWrite(STACKSTRING, oTOP, &ioff, 2, MAXSS);
     };
  return 1;
}



/*-------------------------------------------------------------------------------
/* BURY:
 Should be used with caution, but a logical extension to having a stack.
 See our perl implementation, which is trivial;
 The catch: if we bury an item with a STACKSTRING, we have a major problem, becaus
 if we leave the stackstring, subsequent operations will stuff this up/cause an
 error.
 A possible solution:
   a. have ancillary burialground stackstring area, and move stackstrings there;
   b.
   ABCDE --bury-->
   ABCD             burialground: E
   ABCD --bury-->
   ABC              burialground  ED (D is now deepest, first to be dug up)
   etc.
   Digup reverses the bury command.
   NOW WHAT BETTER PLACE FOR A BURIAL GROUND THAN THE TOP OF THE STACKS, GROWING D

[19-3-2005]
now there is a further problem: if we bury a type X, we *will* stuff it up d/t int
references, so we must resolve it!

*/

/* Bury:
take single item off stack, and move it to far end of stack, adjusting oMAX value
for stack downwards by 16 bytes.
If component on stackstring, then do the same for this, AND adjust pointer appropr
one item 'vanishes' off the stack.
NOTE that this implementation differs from the Perl one, in that we can still 'pop
buried item (unmatched by a digup) off the Perl stack, whereas we cannot make the
same mistake here!
*/

Int16  Bury(UInt16 refnum, Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 max;
  Int16 ilen;
```

```
Int16 Stop;
Int16 Smax;
Int16 ioff;
Int16 ok;

/* standard pop: */
bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
top    = *((Int16 *)(STACK+oTOP));
max    = *((Int16 *)(STACK+oMAX));
if (top <= bottom)
   { return -ScErBuryEmpty;                   /* fail: nothing to bury */
   };
max -= XVI;
if (max < top)
   { return -ScErBuryFull;
   };
top -= XVI;

if (* (STACK+top+15) == 'X')
   { ok = eXtract(refnum, STACK, STACKSTRING, top); // effectively convert type
     // it would here be feasible to adjust stackstring top, as (see eXtract)
     // there is usually some space above the top of the resolved string before
     // we encounter the oTOP of STACKSTRING!
     if (ok < 1)
        {
          return ok;
        };
   };

w_DmWrite(STACK, max, STACK+top, XVI, SMAX);   /* copy over top item */
w_DmWrite(STACK, oTOP, &top, 2, SMAX);        /* write new top, 16 bytes down */
w_DmWrite(STACK, oMAX, &max, 2, SMAX);        /* AND new max*/


ilen = 0x0F & *(STACK+top+14);                 /* get item length */
if (ilen > 14)                                 /* if stackstring exists */
   { ilen = *((Int16 *)(STACK+top));           /* get (original) length */
     // debugging:
     if (ilen > 1000)
        { return -999; // ???: force failure
        };
     ioff = *((Int16 *)(STACK+top+2));         /* get stackstring offset */
     Stop = *((Int16 *)(STACKSTRING+oTOP));
     Smax = *((Int16 *)(STACKSTRING+oMAX));
     Smax -= ilen;
     if (Smax < Stop)
        { /* do NOT assume that PalmOS will handle data transfer intelligently.
```

```
              rather fail [? check me]
            */
            return -ScErBurySpace;
          };
       // might also check that ioff+ilen = Stop BUT don't: see eXtract for reason

       w_DmWrite(STACKSTRING, Smax, STACKSTRING+ioff, ilen, MAXSS); /* ??? the MAX
       w_DmWrite(STACKSTRING, oMAX, &Smax, 2, MAXSS);  /* revised Smax down */
       w_DmWrite(STACKSTRING, oTOP, &ioff, 2, MAXSS); /* likewise for top */
       w_DmWrite(STACK, max+2, &Smax, 2, SMAX);       /* new offset */
     };
  return 1;
}



/*-------------------------------------------------------------------------------
// Digup:
//
// reverses the Bury process
// Note that with long strings, the +2 offset pointer now points to the upper regi
// of the stackstring, ie at [oMAX], and this needs to be reversed.

Int16  Digup (Char * STACK, Char *  STACKSTRING)
{
  Int16 top;
  Int16 max;
  Int16 ilen;

  Int16 Stop;
  Int16 Smax;
  Int16 ioff;
  top    = *((Int16 *)(STACK+oTOP));
  max    = *((Int16 *)(STACK+oMAX));

  if (max >= SMAX)
     { return -ScErDigEmpty;
     };
  /* the following is a hack */
  if (max - top < XVI)
     { return -ScrErDigFull;                    /* don't assume intelligent trans
     };

  w_DmWrite(STACK, top, STACK+max, XVI, SMAX);  /* transfer item back */
  ilen = 0x0F & *(STACK+max+14);                 // might otherwise get from [STAC
    // major stuffup because we had 0xF0 not 0x0F!
  if (ilen > 14)
     { ilen = *((Int16 *)(STACK+max));
       ioff = *((Int16 *)(STACK+max+2));
```

```
      Stop = *((Int16 *)(STACKSTRING+oTOP));
      Smax = *((Int16 *)(STACKSTRING+oMAX));
      if (Smax - Stop < ilen)
         { return -ScrErDigSpace;
         };
      /* here might check that ioff = Smax ? [hmm?] */
      w_DmWrite(STACKSTRING, Stop, STACKSTRING+ioff, ilen, MAXSS); // copy over s
      w_DmWrite(STACK, top+2, &Stop, 2, SMAX);       /* new offset */
      Stop += ilen;
      Smax += ilen;
      w_DmWrite(STACKSTRING, oMAX, &Smax, 2, MAXSS);  /* revised Smax up */
      w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS); /* likewise for top */
    };
  max += XVI;
  top += XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);      /* write new top, 16 bytes UP */
  w_DmWrite(STACK, oMAX, &max, 2, SMAX);        /* AND new max*/
  return 1;
 }
```

1.7.7 Swop

Swop around two items on the stack, the topmost and the next one down Unfortunately this necessitates our swopping stackstring items too, if they exist for *both* stack items. (Or writing extensive garbage collection routines).

```
Int16  MySwop (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 max;
  Int16 ilen;
  Int16 i2;

  Int16 Stop;
  Int16 Smax;
  Int16 Olow;
  Int16 Ohi;

  /* 1. check for two items on stack, and space */
  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  max    = *((Int16 *)(STACK+oMAX));
  if (top+XVI > max)
     { return -ScErSwopFull;
     };
```

```
  top -= XVI;
  if (top <= bottom)
     { return -ScErSwopEmpty;  /* fail: not enough on stack */
     };

  /* 2. swop stack items */
  w_DmWrite(STACK, top+XVI, STACK+top, XVI, SMAX); /* take top item out */
  w_DmWrite(STACK, top, STACK+top-XVI, XVI, SMAX); /* move lower item in */
  w_DmWrite(STACK, top-XVI, STACK+top+XVI, XVI, SMAX); /* and move top back into l

  ilen = 0x0F & *(STACK+top+14);
  if (ilen > 14)
     { i2 = 0x0F & *(STACK+top-XVI+14); /* get length of lower item */
       if (i2 > 14)                    /* if both long, must swop on stackstring t
          { ilen = *((Int16 *)(STACK+top));
            i2 = *((Int16 *)(STACK+top-XVI));
            Stop = *((Int16 *)(STACKSTRING+oTOP));
            Smax = *((Int16 *)(STACKSTRING+oMAX));
            if (Smax - Stop < ilen)
               { return -ScErSwopLong;   /* not enough stackstring space */
               };
            Olow = *((Int16 *)(STACK+top+2)); /* Olow matches ilen */
            Ohi  = *((Int16 *)(STACK+top-XVI+2)); /* Ohi corresponds to i2 */
            w_DmWrite(STACKSTRING, Stop, STACKSTRING+Olow, ilen, MAXSS); /* move l
            w_DmWrite(STACKSTRING, Olow, STACKSTRING+Ohi, i2, MAXSS); /* move uppe
              /* in the above stmt we trust the o/s not to behave stupidly */
            w_DmWrite(STACKSTRING, Olow+i2, STACKSTRING+Stop, ilen, MAXSS); /* low
            w_DmWrite(STACK, top-XVI+2, &Olow, 2, SMAX); /* lower now points low *
            Olow += i2;
            w_DmWrite(STACK, top+2, &Olow, 2, SMAX); /* higher now points to top i
          };
     };
  return 1;
}
```

1.7.8 Replace

Similar to Swop, but simply remove the deeper item on the stack. Clumsily written. A sometimes-very-convenient utility.[7]

```
Int16  MyReplace (UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 top;
```

[7] Previously we conceived of a 'Replace' which would replace matching items within a string. This routine does *not* do so, replacing an item on the stack with the topmost item.

```
Int16 xlen;
Int16 xoff=0;
Int16 oldlen;
Int16 oldoff=0;

Char * xbuf;

/* 1. check for two items on stack, and space */
bottom = *((Int16 *)(STACK+oSTART));
top    = *((Int16 *)(STACK+oTOP));

top -= XVI;
if (top <= bottom)
   { SayErr (refnum,ScErReplace);
     return -1;
   };

// find length and offset of overwritten string:
xlen = 0x0F & *(STACK+top-XVI+14);
if (xlen > 14)
   { xlen = *((Int16 *)(STACK+top-XVI));
     xoff = *((Int16 *)(STACK+top-XVI+2));
   };
// and that of retained string:
oldlen = 0x0F & *(STACK+top+14);
if (oldlen > 14)
   { oldlen = *((Int16 *)(STACK+top));
     oldoff = *((Int16 *)(STACK+top+2));
   };

// replace lower item with top
w_DmWrite(STACK, top-XVI, STACK+top, XVI, SMAX);

// fix up STACKSTRING, if needed:
if (xlen > 14) // if need to cut out of stackstring:
   { if (oldlen > 14) // [could err if oldlen on stackstring < 15?! NO!]
        { // rather than assuming rational OS behaviour, use buffer:
          xbuf = xNew(oldlen+1);
          xCopy(xbuf, STACKSTRING+oldoff, oldlen); // to buffer
          w_DmWrite(STACKSTRING, xoff, xbuf, oldlen, MAXSS); // move down
          w_DmWrite(STACK, top-XVI+2, &xoff, 2, SMAX);
          xoff += oldlen;
          Delete(xbuf);
        };
     w_DmWrite(STACKSTRING, oTOP, &xoff, 2, MAXSS);
   };
```

```
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // new top, 1 item down.
  return 1;
}
```

1.8 String-handling functions

1.8.1 IN

InString corresponds to the **IN** command. It tests whether the string on TOP of the stack is contained within the string deep to it (next down), and only applies to a text string (type 'V'). It is CaSe SENsitIvE! We updated this routine on 2007-12-08 to permit compound (X) types on the stack by using eXtract.

```
Int16  InString (UInt16 refnum, Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char * Pi;
  Char * MP;
  Int16 pilen;
  Int16 mplen;
  Char c;
  Int32 i;
  Int16 soffset;
  Char itype;

  // standard pop:
  bottom = *((Int16 *)(STACK+oSTART));  // or beReadInt16 ??
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     { return -ScErInEmpty;             // fail: insufficient stack
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // pop just one item

  itype = *(STACK+top+15);
  if (itype != 'V')            // if not a string
     { if ( (itype != 'X')
          ||(eXtract(refnum, STACK, STACKSTRING, top) < 1)
          )
          { return -ScErInType;
          };
     };

  pilen = 0x0F & *(STACK+top+14);
  if (pilen > 14)
```

```
    { pilen = *((Int16 *)(STACK+top+0));
      soffset = *((Int16 *)(STACK+top+2));
      Pi = STACKSTRING + soffset;
      w_DmWrite(STACKSTRING, oTOP, &soffset, 2, MAXSS); // clean up stackstring
    } else
    { Pi = STACK+top;
    };

 top -= XVI;
 itype = *(STACK+top+15);
 if (itype != 'V')  // likewise
    { if ( (itype != 'X')
         ||(eXtract(refnum, STACK, STACKSTRING, top) < 1)
         )
         { return -ScErInType;
         };
    };

 mplen = 0x0F & *(STACK+top+14);
 if (mplen > 14)
    { mplen = *((Int16 *)(STACK+top+0));
      soffset = *((Int16 *)(STACK+top+2));
      MP = STACKSTRING + soffset;
      w_DmWrite(STACKSTRING, oTOP, &soffset, 2, MAXSS); // clean up stackstring
    } else
    { MP = STACK+top;
    };

 if (pilen > 0)               // if not NULL string
    {
// We really should use Boyer-Moore, but here we laboriously just ...
      c = *Pi;
      Pi ++;
      pilen --;
      while (mplen > pilen)
        { if (*MP++ == c)
             { if (xSame(MP, pilen, Pi, pilen))
                  { mplen = 0;  // force exit and SIGNAL SUCCESS
                  };
             };
          mplen --;
        };
      if (mplen < 0) // if found !
         { i = 1;    // signal YES
         } else
         { i = 0;    // NO
         };
    } else
```

```
     { i = 1; // any string trivially contains NULL
     };

  c = 'I';    // integer type
  w_DmWrite(STACK, top+15, &c, 1, SMAX);
  c = 4;      // length always 4
  w_DmWrite(STACK, top+14, &c, 1, SMAX);
  w_DmWrite(STACK, top, &i, 4, SMAX);
    // write integer answer boolean 1/0
  return 1;
}
```

1.8.2 Lowercase

```
Int16  DoLowercase (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Char * P;
  Int16 off;
  Char c;

  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErLenEmpty;                    /* fail: insufficient stack */
     };
  top -= XVI;
  ilen = 0x0F & *(STACK+top+14);
  if (ilen > 14)
     { ilen = *((Int16 *)(STACK+top+0));
       off = *((Int16 *)(STACK+top+2));
       P = STACKSTRING;
     } else
     { P = STACK;
       off = top;
     };
  while (ilen > 0)
    { /* cumbersome */
      c = lowercase(*(P+off));
      w_DmWrite(P, off, &c, 1, 32000);
      off ++;
      ilen --;
    };
  return 1;
}
```

1.8.3 Uppercase

```
Int16  DoUppercase (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Char * P;
  Int16 off;
  Char c;

  bottom = *((Int16 *)(STACK+oSTART));          /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErLenEmpty;                    /* fail: insufficient stack */
     };
  top -= XVI;
  ilen = 0x0F & *(STACK+top+14);
  if (ilen > 14)
     { ilen = *((Int16 *)(STACK+top+0));
       off = *((Int16 *)(STACK+top+2));
       P = STACKSTRING;
     } else
     { P = STACK;
       off = top;
     };
  while (ilen > 0)
    { /* cumbersome */
      c = UPPERCASE(*(P+off));
      w_DmWrite(P, off, &c, 1, 32000);
      off ++;
      ilen --;
    };
  return 1;
}
```

1.8.4 SPLIT

We split a single string into many strings, based on a particular character[8] On top of stack is “what to split on”, and below this is “what to act on”. The tricky bit is

[8] We might even make this a sequence of characters — why not?

fixing up the stack. In contrast, Perl is seductively convenient.

```
Int16  SplitString(UInt16 refnum, Char * STACK, Char *  STACKSTRING)
{

  Int16 bottom;
  Int16 top;
  Char * divider;
  Int16 divlen;
  Char * strg;
  Int16 stlen;
  Int16 SStop;
  Char c;
  Char * P;
  Char * kept;
  Int16 j;  // split length

  bottom = *((Int16 *)(STACK+oSTART));  /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if ((top - bottom) < XVI) // if not enough arguments
     {
       SayErr(refnum, ScErSplitEmpty);
       return -1; // fail
     };

  strg = STACK + top;
  if( ! ( 0x0f & (*(UInt8 *)(strg-XVI+14))) ) // if null splittee
    { w_DmWrite(STACK, oTOP, &top, 2, SMAX); // return null!
      return 1;
    };


  // 1. get string to split on:

  // 1a. check length:
  divlen = 0x0F & *(strg+14);  // CAN be zero (null)

    // might check that it's type 'V':
  if (  (divlen > 0)
      &&( *(strg+15) != 'V' )
     )
     {
       SayErr(refnum, ScErSplitSeparator);
       return -1; // fail
     };
  if (divlen > 14)
     { divlen = *((Int16 *)(strg+0)); // extended string length
       SStop = *((Int16*)(strg+2)); // offset in stackstring
```

```
       strg = STACKSTRING + SStop;  // => new ptr
       w_DmWrite(STACKSTRING, oTOP, &SStop, 2, MAXSS); // keep track on STACKSTRIN
     };
  divider = xNew(divlen);       // keep copy of divider
  xCopy(divider, strg, divlen); //
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // keep track of actual stack top!

  // debug console write:
   WriteConsoleText(refnum, "[", 1, fDEBUG_DUMP);  // debug only
   WriteConsoleText(refnum, divider, divlen, fDEBUG_DUMP);
   WriteConsoleText(refnum, ":", 1, fDEBUG_DUMP);  // debug only

+OPTIONAL
// THE FOLLOWING IS STRICTLY DEBUGGING:
/*   WriteConsoleText(refnum, "<", 1, 0);
   P = STACK + top - XVI;
   stlen = 0x0F & *(P+14);
   if (stlen > 14)
      { stlen = *((Int16 *)(P+0));
        P = STACKSTRING + *((Int16 *)(P+2));
      };
   WriteConsoleText(refnum, P, stlen, 0);
   WriteConsoleText(refnum, ">", 1, 0); //fDEBUG_DUMP
*/
// END DEBUGGING
-OPTIONAL


  // 2. get 'splittee':
  stlen = MAXSTRINGLENGTH; // must set this before invoke StackPeek
  StackPeek (0, STACK, STACKSTRING, 0, &stlen); // get length of item on stack
  if (stlen < 1)
     { Delete(divider);
       SayErr(refnum, ScErSplitLen);
       return -1; // fail
     };

+OPTIONAL
   WriteConsoleText(refnum, "(l=", 3,  fDEBUG_DUMP);  // debug only
   WriteConsoleInteger(refnum, stlen,  fDEBUG_DUMP);           //
   WriteConsoleText(refnum, ")", 1,  fDEBUG_DUMP);
-OPTIONAL


  strg = xNew(stlen+0x10); // hmm add a few bytes just in case [check me]
  stlen = Resolve(refnum, strg, stlen+0x10, STACK, STACKSTRING);
  // [stlen instead of stlen+0x10 caused enormous pain]
  if (stlen < 1)
```

```
   { Delete(divider);
     SayErr(refnum, ScErSplitResolve);
     return -1; // fail [perhaps put NULL?]
   };
+OPTIONAL
WriteConsoleText(refnum, strg, stlen, fDEBUG_DUMP); // debug only
WriteConsoleText(refnum, "->", 2, fDEBUG_DUMP);
-OPTIONAL

// 3. perform the split
// what about a terminal item (ignore terminal divider?)
if (!divlen) // split on NULL ie explode string into characters:
   { j = 0;
     while (j < stlen)
       { PushItem (0, STACK, STACKSTRING, strg+j, 1, 'V', 0);
         j++;
       };
   } else
   { c = *(divider);
     P = strg;
     j = Advance(P, stlen, c);
     while (j > 0)
       { kept = P;
         P += j;
         stlen -= j;
         if ( xSame(P, divlen-1, divider+1, divlen-1) ) // if rest is also the s
              // [this will also work if divlen is just 1]
            { PushItem (0, STACK, STACKSTRING, kept, j-1, 'V', 0);
              // we initially erred by specifying "j" and not "j-1".
              /// debug:
              +OPTIONAL
               WriteConsoleText(refnum, kept, j-1, fDEBUG_DUMP); // debug only
               WriteConsoleText(refnum, ";", 1, fDEBUG_DUMP);
              -OPTIONAL
              P += divlen-1;
              stlen -= divlen-1;
            };
         j = Advance(P, stlen, c);
       };
     // 3a. terminal item:
     if (stlen > 0)
        { PushItem(0, STACK, STACKSTRING, P, stlen, 'V', 0);
          +OPTIONAL
           WriteConsoleText(refnum, P, stlen, fDEBUG_DUMP); // debug only
          -OPTIONAL
        };
   };
```

```
  // 4. clean up:
  Delete(divider);
  Delete(strg); // might check for success

 +OPTIONAL
 WriteConsoleText(refnum, "]", 1, fDEBUG_DUMP); // debug only
 -OPTIONAL

return 1;  // ok
}
```

1.8.5 LENGTH

The length returned is really only relevant for numeric and varchar types. Others return a standard length which is not related to the displayed length once converted to ascii. Length of numeric does NOT include sign or decimal point — the obvious solution is to turn the numeric into a string.

```
Int16  GetLength (UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Int16 ilen = -1;
  Char t; // type of item

  bottom = *((Int16 *)(STACK+oSTART));           /* or beReadInt16 ?? */
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErLenEmpty;                     /* fail: insufficient stack */
     };

  top -= XVI;
  t = *(STACK+top+15);
  if (t == 'X')
     { if (eXtract(refnum, STACK, STACKSTRING, top) < 1)
          { ilen = 0; // failed
          };
     };  // "Resolve" first!

  if (ilen)  // nonzero (see above)
     { ilen = 0x0F & *(STACK+top+14);
       if (ilen > 14)
          { ilen = *((Int16 *)(STACK+top));
          };
     };

  cleanstackstring(STACK, STACKSTRING, top);
```

```
  writeinteger(STACK, top, ilen);
  return 1;
}
```

1.8.6 Cut

Cut a string into two, based on an integer offset specified on the stack.
[WOOPS. MUST RESOLVE IN THE FOLLOWING]!

```
Int16  DoCut(UInt16 refnum, Char * STACK, Char *  STACKSTRING)
{ Int16 bottom;
  Int16 top;
  Char * strg;
  Int16 stlen;
  Int16 j;  // split length

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if ((top - bottom) < XVI) // if not enough arguments
     { SayErr(refnum, ScErCutEmpty);
       return -1; // fail
     };

  // 1. get cut length:
  if (*(STACK+top+15) != 'I')
     { SayErr(refnum, ScErCut);
       return -2;
     };
  j = (Int16) (*((Int32 *)(STACK+top))); // get cut length
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // track stack top

  // 2. resolve string:
  stlen = MAXSTRINGLENGTH;
  StackPeek (0, STACK, STACKSTRING, 0, &stlen); // item length?
  if (stlen < 1) // hmm [what if submit NULL?]
     { SayErr(refnum, ScErCut);
       return -3; // fail
     };
  strg = xNew(stlen+0x10); // add a few bytes [ugh]
  stlen = Resolve(refnum, strg, stlen+0x10, STACK, STACKSTRING);
  if (stlen < 1)
     { SayErr(refnum, ScErCut);
       return -4; // fail [perhaps put NULL?]
     };

  // 3. perform the split
  if (stlen < j)
```

```
     { PushItem (0, STACK, STACKSTRING, strg, stlen, 'V', 0);
       PushNull(STACK);
     } else
     { PushItem (0, STACK, STACKSTRING, strg, j, 'V', 0);
       PushItem (0, STACK, STACKSTRING, strg+j, stlen-j, 'V', 0);
     };

  // 4. clean up:
  Delete(strg); // might check for success
  return 1;  // ok
}


/* ============================================================================
/* N. MISCELLANEOUS FUNCTIONS INCLUDING TIME. */

/*----------------------------------------------------------------------------
/* NOW:
 return timestamp on stack. format is usual "SCRIPTINGYMMDDHHMMSS" ie 14 chars lon

 Internal PalmOS functions provide:
 typedef struct{
Int16 second;
Int16 minute;
Int16 hour;
Int16 day;
Int16 month;
Int16 year;
Int16 weekDay;
} DateTimeType
*/


Int16  TimestampNow (Char * STACK)
{
  UInt32 secs;
  DateTimePtr dtp;
  Char * bufr;
  Int16 top;
  Int16 max;

  top    = *((Int16 *)(STACK+oTOP));
  max    = *((Int16 *)(STACK+oMAX));
  top += XVI;
  if (top > max)
     { return -ScErNowFull;
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);
  top -= XVI;
```

```
  secs = TimGetSeconds();                /* hmm. 2026 will be a problem */
  dtp = (DateTimePtr) xNew( sizeof(DateTimeType) );
  TimSecondsToDateTime (secs, dtp);
  bufr = xNew(XVI);
  Write4Digits(bufr,   dtp->year);
  Write2Digits(bufr+4, dtp->month);
  Write2Digits(bufr+6, dtp->day);
  Write2Digits(bufr+8, dtp->hour);
  Write2Digits(bufr+10,dtp->minute);
  Write2Digits(bufr+12,dtp->second);
  *(bufr+14) = 14;                                  /* length */
  *(bufr+15) = 'S';                                 /* signal timestamp */
  w_DmWrite(STACK, top, bufr, XVI, SMAX);        /* copy over whole schmeer */

  Delete(bufr);
  Delete((Char *)dtp);
  return 1;
}
```

1.8.7 SetTime

Given a timestamp set the PDA system time to the new time if valid. First, a few trivial subroutines.

```
Int16 read2digits (Char * P)
{ Int16 i;
  i  = 10 * ( *(P) - '0');
  i += * (P+1) - '0';
  return i;
}

Int16 read4digits (Char * P)
{
  Int16 i;
  i = 100 * read2digits(P);
  i += read2digits(P+2);
  return i;
}

Int16  SetTime (Char * STACK)
{ DateTimePtr dtp;
  Int16 bottom;
  Int16 top;
  Char * P;
  UInt32 ui;
  Int16 ok = 1;
```

```
  bottom = beReadInt16(STACK+oSTART);
  top = beReadInt16(STACK+oTOP);
  if (top <= bottom)
     { ok = -ScErSecEmpty;  // fail: nothing on stack
       top = bottom;
       writeinteger(STACK, top, 0);
       top += XVI;
       w_DmWrite(STACK, oTOP, &top, 2, SMAX); // adjust stack
       return ok;
     };

  top -= XVI;
  if ( *(STACK+top+15) != 'S')
     { ok = -ScErSecType;   // must be a timestamp
       writeinteger(STACK, top, 0);
       return ok;
     };

  P = STACK + top;
  dtp = (DateTimePtr) xNew( sizeof(DateTimeType) );
  dtp->year = read4digits(P);   // NO ERROR CHECKING
  dtp->month= read2digits(P+4);
  dtp->day  = read2digits(P+6);
  dtp->hour = read2digits(P+8);
  dtp->minute=read2digits(P+10);
  dtp->second=read2digits(P+12);
  ui=TimDateTimeToSeconds(dtp);

  if (ui < 3281990400) // fail if under about 2008 (can check/fix this)
     { ok = -ScErSecEarly;
       writeinteger(STACK, top, 0);
       Delete((Char *)dtp);
       return ok;
     };
  TimSetSeconds(ui); // doesn't validate

  writeinteger(STACK, top, 1); // success
  Delete((Char *)dtp);
  return ok;
}


/*-------------------------------------------------------------------------------
/* SECONDS:
 Convert a TIMESTAMP into a system-local number of seconds after a particular date
 This is a local convenience for date calculation. Should be abandoned in favour o
 conversion to a Julian date ???
```

```
  [FIX ME: AVOID THIS SHIT] !!!
*/

Int16  LocalTimeConvert (Char * STACK, Char *  STACKSTRING)
{
  DateTimePtr dtp;
  Int16 bottom;
  Int16 top;
  Int16 ilen;
  Char * P;
  UInt32 ui;

  bottom = beReadInt16(STACK+oSTART);
  top = beReadInt16(STACK+oTOP);
  if (top <= bottom)
     { return -ScErSecEmpty;  /* fail: nothing on stack */
     };
  top -= XVI;
  if ( *(STACK+top+15) != 'S')
     { return -ScErSecType;                    /* must be a timestamp */
     };
  ilen = 0x0F & *(STACK+top+14);
  if (ilen > 14)
     { P = STACKSTRING + *((Int16 *)(STACK+top+2));
     } else
     { P = STACK + top;
     };
  /* might check that ilen is at least 14 chars long ? */

  dtp = (DateTimePtr) xNew( sizeof(DateTimeType) );
  dtp->year = read4digits(P);                   /* NO ERROR CHECKING */
  dtp->month= read2digits(P+4);
  dtp->day  = read2digits(P+6);
  dtp->hour = read2digits(P+8);
  dtp->minute=read2digits(P+10);
  dtp->second=read2digits(P+12);
  ui=TimDateTimeToSeconds(dtp);
  if (ui < 3000000000)                         /* PalmOS dates from 1904 ???*/
     { Delete((Char *)dtp);
       return -ScErSecEarly;
     };
  ui -= 3000000000;                            /* adjust to fit our narrow frame
  Delete((Char *)dtp);

  writeinteger(STACK, top, (Int32)ui);
  return 1;
}
```

```
/*-------------------------------------------------------------------------
/* MAP:
  another complex function. [removed]
  format is "option1:result1|option2:result2" etc. Return one option based on matc

  THIS IS LITTLE UTILISED AT PRESENT. RATHER REWRITE THE SINGLE USE, IMPLEMENT
  PROPERLY LATER, AND FOR NOW, FAIL
*/

Int16  Map (Char * STACK, Char *  STACKSTRING)
{ return -ScErMap;
}
```

1.8.8 Regular expressions: removed

```
/*-------------------------------------------------------------------------
/* REGEX:
  Primitive regex. For now we will probably just take 2 items off stack and return
  [LITTLE USED, DIFFICULT IMPLEMENTATION. NOT A PRIORITY, BUT WILL
  LATER IMPLEMENT. fix me!!]
*/

Int16  Regex (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     { return -ScErRegexEmpty;                  /* fail: insufficient stack */
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);       /* pop just one item */
  top -= XVI;
  writeinteger(STACK, top, 1);                  /* simply succeed, for now */
  return 1;
}
```

1.8.9 Compare Stack Items: removed

```
/*   ============================================================================
// MAX, MIN similar things? Most of these have been moved to ../sql3/SQL3.c

Int16 CompareStackItems (Char * STACK, Char * STACKSTRING, Int16 iA, Int16 iB)
{ // iA, iB are integer offsets into STACK
  // return -1 if A <B, 0 if A = B, +1 if A > B
  // hmm what about floats ??
  // 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 -ScErBadTypeCompare;
     };
  // do not need to switch on type, as all types are stored appropriately even flo
  // (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);
}
```

1.8.10 GetItemStyle

```
//-------------------------------------------------------------------------------
// Another import. GetItemStyle = convert our code into PalmOS code. Trivial, apar
//    from the fact that we ADD 1 to the return code, as buttonCtl code is zero,
//    and we wish to reserve zero to signal 'failed'!

Int16 GetItemStyle(UInt16 refnum, Int16 icode)
{ Int16 answr;
  switch (icode)
     { case BUTTONCTL:
         answr =  (Int16) buttonCtl;
         break;
       case PUSHBUTTONCTL:
         answr =  (Int16) pushButtonCtl;
         break;
       case CHECKBOXCTL:
         answr =  (Int16) checkboxCtl;
         break;
       case POPUPTRIGGERCTL:
         answr =  (Int16) popupTriggerCtl;
         break;
//       case SELECTORTRIGGERCTL: //?
//         answr =  (Int16) selectorTriggerCtl;
//         break;
//       case REPEATINGBUTTONCTL: //?
//         answr =  (Int16) repeatingButtonCtl;
//         break;
//       case SLIDERCTL:
//         answr =  (Int16) sliderCtl;
//         break;
//       case FEEDBACKSLIDERCTL:  //?
//         answr =  (Int16) feedbackSliderCtl;
//         break;
       default:
         return 0; // fail; UNFORTUNATELY buttonCtl code is zero! this is dreadful
     }
 return (answr+1);     // bump by 1 as buttonCtl (PalmOS) is zero!
}




/* ============================================================================
// complex sorting routines:

//-------------------------------------------------------------------------------
```

```
// InsertItem:
  // COMMON FUNCTION used by SORT: and SortFx.
  // Given A an array of 2-byte integers, insert itm at word offset posn. Size of
   // 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 -ScErInsortFailed; // ERRDEBUG("ERROR failed insertion",0);
      };
   if (posn < 0)
      { return -ScErInsortFailed; // ERRDEBUG("Can't insert at -ve posn",posn);
      };
   P = A + alen*2;                          // 2 bytes per item
   i = alen-posn;
   while (i > 0)              // the following might be done a lot more elegantly!
     { *((Int16*)(P)) = *((Int16*)(P-2)); // ugly. [fix me, with due diligence!]
        P-=2;
        i --;
     };
   *((Int16 *)P) = itm;
   return 1;                                      // success
}
```

1.8.11 Find Stack Position: removed

```
/*-------------------------------------------------------------------------
// FindStackPosition:
//
// given a buffer (sortarea) of i two byte indexes into data items on STACK (start
// bottom), get the i-th item off STACK and find its position within the buffer,
// returning this position. Insertion is to the LEFT of the specified position (e.
// 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, in
// above the last equal item (assuming caller still also moves from left to right
// 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 ridic

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

  Int16 refitem = bottom + itm*XVI; // item we are comparing 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] should be able to similarly shorten Finc
    { 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

}



/*-------------------------------------------------------------------------------
// URNZ:
//
// moderately useful function to test topmost item on stack.
// [has not lived up to its promise]
// If the value is ZERO or NULL, or indeed if there is nothing whatsoever on the s
// then it:
// (a) unmarks the stack (discarding everything above the mark)
// (b) sends iRETURN, FORCING a return.
// OTHERWISE nothing is done, and the top item on the stack is NOT discarded/alter


Int16 UnmarkReturnNorZ(Char * STACK, Char * STACKSTRING)  // [???]
```

```
{
  Int16 bottom;
  Int16 top;
  Int16 ok;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  if (  (top <= bottom)                          // if nothing on stack
      ||( (0x0F & *(STACK+top-XVI+14)) == 0)       // or value is NULL (zero lengt
      ||(  ( *(STACK+top-XVI+15) == 'I' )
         &&( *((Int32*)(STACK+top-XVI)) == 0)    // or integer with value of zero
        )
     )
     { ok = ClearMark(STACK,STACKSTRING);             // clear mark and clean stac
       if (ok < 0)
          { return ok;
          };
       return iSTOP;                           // and force return+stop!
     };
  return 1; // simply continue
}
```

1.8.12 Print to Console

This is a debugging function which allows us to print to our PDA console. If the debug print fails we still don't return zero, as this would force an ugly ALERT, which we generally don't want in a debugging function!

```
Int16 PrintConsole(UInt16 refnum, Char * STACK, Char * STACKSTRING)
{
  Int16 top;
  Int16 bottom;
  Int16 ilen;
  Int16 SStop;
  Char c;
  Char * strg;

  bottom = beReadInt16(STACK + oSTART);
  top = beReadInt16(STACK + oTOP);
  SStop = beReadInt16(STACKSTRING + oTOP);

  if (bottom >= top)
     { WriteConsoleText(refnum, "\n??", 3, 0); //
     };
  top -= XVI;
  c = *(STACK+top+15); // get type
```

```
  switch (c)
    {
      case 'V':  // text
      break;

      case 'X':
         if ( eXtract(refnum, STACK, STACKSTRING, top) < 1)
            {
              WriteConsoleText(refnum, "\n?X-prn", 7, 0);
              w_DmWrite(STACK, oTOP, &top, 2, SMAX); // remove item!
              return 1; // [not 0 as this will force alert(ugh)!]
            }; // [or flow onto default ?!]
         break;

      // at present ONLY accept string!
      //case 'N':  // numeric
      //case 'I':  // integer
      //case 'F':  // float
      //case 'D':  // date
      //case 'T':  // time
      //case 'S':  // timeStamp

      default:
        WriteConsoleText(refnum, "\n?Print", 7, 0);
        WriteConsoleText(refnum, &c, 1, 0); // type?!
        w_DmWrite(STACK, oTOP, &top, 2, SMAX); // remove item!
        return 1;  // [despite fail, do NOT give 0]
    };

  ilen = 0x0F & (*(STACK+top+14));
  if (ilen < 15)
     { strg = STACK+top;
     } else
     { ilen = beReadInt16(STACK+top); // length
       SStop = beReadInt16(STACK+top+2); // start of string on STACKSTRING
       strg = STACKSTRING + SStop;
     };

  WriteConsoleText(refnum, strg, ilen, 0); // force write
  w_DmWrite(STACK, oTOP, &top, 2, SMAX); // remove written item
  w_DmWrite(STACKSTRING, oTOP, &SStop, 2, MAXSS); // fix STACKSTRING too!
  return 1;      // ok
}
```

1.9 Script interpretation

First we have a subsidiary routine, which is solely a debugging routine and slows things down enormously.

1.9.1 DumpScript

The following debugging code allows us to write the next script item in full to the console (if fDEBUG_STMT), and (if fDEBUG_DUMP) to then dump the topmost stack item to the console (by making a copy and then dumping this)!

```
+OPTIONAL

Int16 DumpScript (UInt16 refnum, Char * STACK, Char * STACKSTRING,
                  Char * SCRIPT, Int16 SLEN)
{ Char * frd;
  Int16 stkbot;
  Int16 stkitem;
  UInt16 CONSOLE;
  SysLibTblEntryPtr entryP;
  ScriptingLib_globals *gl;
  Int16 ok = 0;

  // to minimise the delay, we here test for relevant flags:
  entryP = SysLibTblEntry (refnum);
  gl = entryP->globalsP;
  CONSOLE = gl->CONSOLE;
  if (! (gl->DEBUGFLAGS & (fDEBUG_STMT | fDEBUG_DUMP))  )
     { return 0;
     };

  WriteConsoleText(refnum, "\n\x95", 2, fDEBUG_STMT);
  WriteConsoleText(refnum, SCRIPT, SLEN, fDEBUG_STMT);
 // the character \x95 is ""

  // in the following, if fDEBUG_DUMP, then before EVERY instruction
  // display [the current stack top item] on the console!
  WriteConsoleText(refnum, "[", 1, fDEBUG_DUMP);

  stkbot = *((Int16*)(STACK+oSTART));
  stkitem = *((Int16*)(STACK+oTOP));
  stkitem -= stkbot;
  WriteConsoleInteger(refnum, (stkitem/XVI), fDEBUG_DUMP);
  WriteConsoleText(refnum, ":", 1, fDEBUG_DUMP);

  // there was a problem in the following if MyCopy fails,
  // as an 'error' is then generated. This is inappropriate,
```

```
  // as it's OK for the stack to be empty! 2006-12-14.
  // Our fix is to pass the final 'sayerr' parameter to MyCopy

  ok = MyCopy(refnum, STACK, STACKSTRING, 0); // make copy,
  if (ok >= 0) // if succeeded
     { frd = xNew(0x10+ok);
       if (frd)
          { ok = Resolve(refnum, frd, ok+0x10, STACK, STACKSTRING);
            if (ok > 0)
               { WriteConsoleText(refnum, frd, ok, fDEBUG_DUMP);
               }; // ???
            Delete(frd);
          } else
          { ok = 0;
          };
     };
  WriteConsoleText(refnum, "]", 1, fDEBUG_DUMP);
  return ok;
}

-OPTIONAL
```

Something interesting we discovered while debugging MyCopy is how if we fail to delete the 'second' backup copy of the PRC file stored by PalmOS in the PDA backup directory, then this old copy is re-installed in preference to the new version, causing endless confusion.

The console write between the square brackets is interesting. We simply invoke *MyCopy*, and if this succeeds, resolve the copied item, writing the buffer contents to the console. The code should clearly ONLY be used for debugging purposes as it slows things down a lot!

1.9.2 Actual script interpretation

`DoScript` actually interprets a script. The can be no leading or trailing `->`. We check for an argument in parenthesis before other evaluation by looking for the terminal right parenthesis! This routine sends a leading dollar sign (for a variable) back as this is managed externally. Function names preceded by either an ampersand or equals sign are also returned back to the caller for external evaluation.

```
Int16 DoScript (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * SCRIPT, In
{  /* assumes: valid STACK, STACKSTRING and SCRIPT pointers. No checks of these.
   */
  Char c;
  Int16 o;
  Int16 ok;
```

```
  Int16 ilen;

Int16 sgn=0;
```

1.9.3 Some debugging

First, we optionally dump the script and stack to the console.

```
+OPTIONAL
   DumpScript( refnum, STACK, STACKSTRING, SCRIPT, SLEN);
-OPTIONAL
```

Next, we send back to caller if a dollar sign is present (local variable usage):

```
  c = * SCRIPT;
  if (c == '$')
     { return iVARNAME;                                  /* 2 signals $[whatever] *
     };
```

1.9.4 Numerics

What about numbers? At present our routines are rudimentary. We aim to implement the following:

1. Integers — preceded by a hash ('pound') sign thus #123
2. Floats — preceded by a percentage sign %3.1415926
3. Fixed point numbers — always preceded by a + or a minus. This approach is unconventional but rigorous. We would allow strings beginning with numeric characters, but favour encasing such strings in quotes![9]

```
  if (c == '#')   // signals an integer
     { return PushInteger(STACK, STACKSTRING, SCRIPT+1, SLEN-1);
     };

  // next, floating point:
  if (c == '%')
     { return PushFloat(STACK, SCRIPT+1, SLEN-1);
     };

  if ((c == '-') || (c == '+'))
```

[9] Even perhaps when they are in parenthesis, to avoid confusion. Perhaps have a 'strict' warning mode as default?

```
    { if (c == '-')
         { sgn = 1;
         }; // ugly, like (sgn = (c == '-'))
      c = * (SCRIPT+1);
      if ((c <= '9') && (c >= '0')) // hmm?
         { return PushNumber (STACK, STACKSTRING, SCRIPT+1, SLEN-1, sgn);
         };
    };
```

1.9.5 A “quoted string”

```
  /* 0. check for "quoted string" */

  if (c == '"')
     { SCRIPT ++;
       SLEN --;
       c = * (SCRIPT+SLEN-1);                    /* get terminal character */
       if (c != '"')
          { return -ScErTerminalQuote;                          /* fail if no term
          };
       SLEN --;                                  /* clip off terminal */

       ok = ParseAndPush(refnum, STACK, STACKSTRING, SCRIPT, SLEN);
       if (ok < 1)
          { return (ok);                  /* fail with relevant code */
          };
       return 1;                                 /* success */
     };
```

1.9.6 Parenthesis

```
  /* A. check for parenthetic argument */
  c = * (SCRIPT + SLEN -1);                      /* get final character */
  if (c == ')')
     { o = Advance(SCRIPT, SLEN, '(');           /* locate left parenthesis */
       if (! o)
          { return -ScErLeftParenthesis;
          };
       ilen = (SLEN-o)-1;
       if (ilen > 0)                             /* permit fx() with no args */
                                                 /* or could just look for "(" lef
          {
            /*   ok = PushItem (refnum, STACK, STACKSTRING, SCRIPT+o, ilen, 'V', 0
            ok = ParseAndPush(refnum, STACK, STACKSTRING, SCRIPT+o, ilen);
            if (ok < 1)
               { return ok;             /* fail with relevant code */
               };
```

```
        };
      SLEN = o-1;                                  /* chop L parenthesis etc */
   };
```

1.9.7 User &function and function

```
/* B. if &fname, return code = 3  )  also allows ->&fx(arg)->   */
c = UPPERCASE(* SCRIPT);
SCRIPT ++;                                         /* discard first character */
SLEN --;                                           /* track */
if (c == '&')
   { return iFUNCTION;                             // implied: parenthetic stuff is
   };

if (c == '=')
   { return eFUNCTION;  // similar to iFUNCTION but pops return before call!
   };
```

2 Alphabetic routine listing

A hash would be a smarter idea for the following, but we laboriously trudge through a case statement examining the first letter of each command, to decide what to do.

Here's a list of the various functions, before we move on to the code. This is alphabetical, in contrast to the functional grouping described in *PerlPgm.tex*. Examine the source .TEX file for more obsolete and experimental functions, several of which have been remmed out.

ADD Add two numbers, placing the result on the stack. There is *no* automatic type conversion!

ALERT Display a message onscreen, with an OK button;[10]

ASK Ask for user input, taking default text off the top of the stack, with a title below this; [?? check me]

AND Logical AND between two items on stack;

BOOLEAN Is value on stack Boolean?

BURY Take the top item off the stack and 'bury' it on the second stack, the opposite of DIGUP;

[10] Formerly this was called SAY, which shouldn't be used.

COMMIT Fake a COMMIT. At present COMMIT is disabled on the PDA;

CONFIRM Ask a question, giving the options OK or Cancel, and returning 1 or 0 in response (Cf ALERT);

COPY Copy the top item on the stack;

CACHE Cache SQL statements on the PDA, optimising performance. See UNCACHE too;

DATE Given a Julian date number, create a Gregorian date;

DEBUG Set current debugging mode, given an integer value. Default value is zero which disables all debugging. Various flags can be used to e.g. debug SQL, variables and so forth.

DIGUP The opposite of bury — retrieve a buried item back onto the stack;

DISCARD Discard the top item on the stack;

DIV Divide the deeper number on the stack by the topmost number on the stack. The same strictures apply as for ADD;

DOSQL Execute SQL statement (*not* a QUERY).

DEPTH How deep is the stack (down to the topmost MARK);

DISTINCT ((Only used within the context of SQL, unique to the PDA))

ENABLED Enable or disable the current item, depending on whether 1 or zero is on the stack;

EXIT Exit the program completely. That's it!

FAIL Current process fails completely (abort);

FLOAT convert to floating point. Converts timestamp to a Julian day number.

GREATER Is deeper number on stack greater than more superficial (topmost) one?

INK Set the foreground ('ink') value for a widget. See PAPER. Not yet active on the PDA;

INTEGER Coerce a number into an integer.

ISNUMBER Is this item a number?

IN Check for a string within a string. Check for the topmost string within the string which is next down (deeper) on the stack;

ISNULL Is the value on the stack NULL?

JOIN Remove topmost stack item, then join all remaining stack items to form a single string, inserting the given sequence (topmost stack item) between each of the component strings.

KEY Given a table name, generate the next sequential key for that table;

LENGTH How long is a string?

LESS Is deeper number less than more superficial (top) one?

LOWERCASE See uppercase. Use sparingly if at all.

LINESLEFT How many lines to the bottom of the current table, as will be used in ROLLMENU.

MARK Mark the stack, making items below the mark inaccessible. Cf UNMARK. It's as if the stack ends down at the mark;

ME Return the unique (database) ID of the current user;

MENU Given the text name of a menu, open that menu storing the current menu name on the menu stack; if supplied with a number, then go back *that number* of menus (popping them off the menu stack); a value of zero should reload the current menu! See also POPMENU;

MOD Calculate the deeper number modulo the topmost number on the stack;

MUL Multiply two numbers on the stack, replacing them with the result;

NAME Create a named variable ('name'), the convention $[name] subsequently being used to refer to that variable within scripts. See also SET;

NEG Negate the number on the stack;

NOT Apply a logical NOT to the value on the stack. Not gives 1 only if the value on the stack is one of zero or NULL.

NOW A timestamp obtained when the script started executing.

NULL Put null value on stack;

OR Logical OR of two items on stack;

PAPER Unused on the PDA as yet, but set the background attribute of a widget. See also INK;

POPMENU Pop a menu and the associated X value off the stack. Which menu we pop is determined by the number currently on the stack. The menu specified (and X value) are clipped out of the stack! See also PUSHMENU.

PRINT Print to console.

PUSHMENU The opposite of POPMENU. May need a little work, so use with fear and trembling?!

QUERY perform SQL query and fetch result(s) of one row only;

QMANY Like QUERY, but retrieve as many rows as there are;

QOK Did the most recent query (or QMANY) succeed?

REFRESH

REPEAT Repeatedly invoke the specified &function, until that function specifies STOP.

RETURN Return from ‘function’ call;

RUN Take a string off the stack and run it as a script! Powerful and dangerous;

ROLLBACK Fake a rollback. At present the rollback facility is disabled on the PDA;

ROLLMENU A quirky command. Force a reload of the current menu *but* scroll the current table(s) contained in the menu down by one screen! See also LINESLEFT;

SAME Are two strings or other items identical;

SETME Set the unique database ID of the current user (cf. ME);

SETX Set the value of the transfer variable X. The value can only be an *integer*!

SET Set the value of a NAME;

SKIP Skip over next statement;

SPLIT Split a string into several strings, using the given string to identify where to split.

STOP Terminate current process (function). See also REPEAT;

SUB Subtract the topmost number from the next deepest number on the stack;

SWOP Swop (not SWAP!) the top two items on the stack;

TICKS Get timer ticks (PalmOS system ticks, about 10 ms);

TIMESTAMP Given a Julian day number, convert to a Gregorian timestamp.

TIME Convert a number (in seconds) into a time HH:MM:SS; If a timestamp is submitted we simply pull out the time portion and discard the rest!

TITLE Set title of current menu!

TEST Used to test PDA code (debugging only);

UNMARK Turn off the most recent mark, making items below that mark accessible again;

UPPERCASE A cumbersome function which should best be replaced by a more generic conversion routine, perhaps along the lines of Perl regex substitutions. Use with caution if at all. Likewise for lowercase.

URZN Obsolete test, which unmarks the stack and returns if top item tested is zero or non-existent (must write this out of all code, do *not* use);

UNCACHE Turn off PDA caching of a data set for a given table (See CACHE);

V Determine the Value associated with a particular element within a table, and put this value on the stack.

X Retrieve the single transfer variable for this menu. The transfer variable X is the only direct link between menus; all other communication is via the database! See also SETX;

```
/* C. process the relevant function!  */
/* AFAIK the default C++ switch implementation is a little slow, but
   this is the least of our problems. [If we really wanted speed we could
   have some sort of binary set of nested elses, or even a table lookup]
*/
switch (c)
  {
    case 'A':                                   /* ADD ALERT AND ASK */
      if (LUPSAME(SCRIPT,SLEN, "DD",2))
         { return iA_ADD;
           // return DoAdd(STACK, STACKSTRING);
         };
```

```
  if (LUPSAME(SCRIPT,SLEN, "LERT",4))
     { return iALERT; /* oops. cf SAY */
     };
  if (LUPSAME(SCRIPT,SLEN, "ND",2))
     { return AndLogic(STACK);
     };
  if (LUPSAME(SCRIPT,SLEN, "SK",2))
     { return iASK;                          /* 4 signals YOU POP STACK, as
     };
  if (LUPSAME(SCRIPT,SLEN, "NY",2))
     { return AnyTest(STACK);
     };
  return -ScErAx;                             /* bad A-fx */


case 'B':                                   /* BOOLEAN BURY */
  if (LUPSAME(SCRIPT,SLEN, "OOLEAN",6))
     { return MyBoolean(STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "URY",3))
     { return Bury(refnum, STACK, STACKSTRING);
     };
  return -ScErBx;

case 'C':                                   /* COMMIT CONFIRM COPY */
  if (LUPSAME(SCRIPT,SLEN, "OMMIT",5))
     { return iCOMMIT; /* signal: you do this */
     };
  if (LUPSAME(SCRIPT,SLEN, "ONFIRM",6))
     { return iCONFIRM;   /* for now at least, signals 'You ask for confirma
     };
  if (LUPSAME(SCRIPT,SLEN, "ONSOLE",6))
     { return iCONSOLE; // will eventually open the console!
     };
  if (LUPSAME(SCRIPT,SLEN, "OPY",3))
     { if (MyCopy(refnum, STACK, STACKSTRING, 1) >= 0)
          { return 1;
          };
       return 0;
     };
  if (LUPSAME(SCRIPT,SLEN, "ACHE", 4)) // caching
     { return iCACHE;
     };
  if (LUPSAME(SCRIPT,SLEN, "UT", 2)) // cut
     { return DoCut(refnum, STACK, STACKSTRING);
     };
  return -ScErCx;
```

```
      case 'D':                                  /* DEBUG DIGUP DISCARD DIV DOSQL
        if (LUPSAME(SCRIPT,SLEN, "ATE",3))
           { return EncodeDate(refnum, STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "EBUG",4))
           { return iDEBUG;
           };
        if (LUPSAME(SCRIPT,SLEN, "IGUP",4))
           { return Digup(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "ISCARD",6))
           { return Discard(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "IV",2))
           { return iA_DIV;
             // return DoDiv(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "OSQL",4))
           { return iDOSQL;  /* signal: YOU get SQL statement off the stack, for r
           };
        if (LUPSAME(SCRIPT,SLEN, "EPTH",4))
           { return MarkDepth(STACK);
           };
//        if (LUPSAME(SCRIPT,SLEN, "ISTINCT",7))
//           { return DoDistinct(STACK, STACKSTRING);
//           };
        if (LUPSAME(SCRIPT,SLEN, "UMP",3))
           { return iDUMP;
           };
        return -ScErDx;


      case 'E':                                  /* ENABLED */
        if (LUPSAME(SCRIPT,SLEN, "NABLED",6))
           { ok = popboolean(STACK);
             if (! ok)
                { return iDISABLED;
                };
             if (ok == 1)
                { return iENABLED;
                };
             return ok;                          /* fail */
           };
        if (LUPSAME(SCRIPT,SLEN, "XIT",3))
           { return iQUIT; // signal: well then, quit!
           };
        return -ScErEx;
```

```
      case 'F':                                  /* FAIL FLOAT */
        if (LUPSAME(SCRIPT,SLEN, "AIL",3))
           { return iFAIL;
           };
        if (LUPSAME(SCRIPT,SLEN, "LOAT",4))
           { return EncodeFloat(refnum, STACK, STACKSTRING);
           };
//        if (LUPSAME(SCRIPT,SLEN, "ORCEX",5))
//           { return iFORCEX; /* signal: YOU force X ??? LATER TO MOVE IT HERE !
//           };
      return -ScErFx;



      case 'G':                                  /* GREATER */
        if (LUPSAME(SCRIPT,SLEN, "REATER",6))
           { return DoGreater(refnum, STACK, STACKSTRING);
           };
      return -ScErGx;


      case 'H':
        return -ScErHx;                               /* bad H-fx */

      case 'I':                                  /* IAM IN INK INTEGER ISNUMBER */
        if (LUPSAME(SCRIPT,SLEN, "AM",2))
           { return iIAM; /* signal: YOU handle this one, for now */
           };
        if (LUPSAME(SCRIPT,SLEN, "NK",2))
           { return iINK; /* signal: YOU set the ink colour */
           };
        if (LUPSAME(SCRIPT,SLEN, "NTEGER",6))
           { return EncodeInteger(refnum, STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "SNUMBER",7))
           { return TestNumber(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "N",1))
           { return InString(refnum, STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "SNULL",5))     // ISNULL [2005-2-20]
           { return TestNull(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "SNAME",5))     //
           { return iISNAME;
           };
      return -ScErIx;
```

```
case 'J':
if (LUPSAME(SCRIPT, SLEN, "OIN", 3))  // JOIN
     { return Stack2String(refnum, STACK, STACKSTRING, 0); // not LIST
     };
return -ScErJx;


case 'K':                                   /* KEY */
  if (LUPSAME(SCRIPT,SLEN, "EY",2))
     { return iKEY;  /* signal: you generate the SQL key */
     };
return -ScErKx;


case 'L':                                   /* LABEL LENGTH LESS LIST LOWERCA
  if (LUPSAME(SCRIPT,SLEN, "ABEL",4))
     { return iLABEL; /* signal: you do the label */
     };
  if (LUPSAME(SCRIPT,SLEN, "ENGTH",5))
     { return GetLength(refnum, STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "ESS",3))
     { return TestLess(STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "IST",3))
     { return Stack2String(refnum, STACK, STACKSTRING, 1); // is LIST
     }; // fx is only used in internal processing.

  if (LUPSAME(SCRIPT,SLEN, "OWERCASE",8))
     { return DoLowercase(STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "INESLEFT",8))
     { return iLINESLEFT;
     };
return -ScErLx;


case 'M':                                   /* MAP MARK ME MENU MOD MUL */
  if (LUPSAME(SCRIPT,SLEN, "AP",2))
     { return Map(STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "ARK",3))
     { return SetMark(STACK);
     };
  if (LUPSAME(SCRIPT,SLEN, "E",1))
     { return iME; /* signal: get 'me' value, put on stack */
     };
  if (LUPSAME(SCRIPT,SLEN, "ENU",3))
```

```
            { return iMENU; /* signal: go to menu */
            };
          if (LUPSAME(SCRIPT,SLEN, "OD",2))
            { return iA_MOD;
              // return Mod(STACK, STACKSTRING);
            };
          if (LUPSAME(SCRIPT,SLEN, "UL",2))
            { return iA_MUL;
              // return DoMul(STACK, STACKSTRING);
            };
//          if (LUPSAME(SCRIPT,SLEN, "AX",2))
//            { return DoMax(STACK, STACKSTRING);
//            };
//          if (LUPSAME(SCRIPT,SLEN, "IN",2))
//            { return DoMin(STACK, STACKSTRING);
//            };
        return -ScErMx;


        case 'N':                                   /* NAME NEG NOT NOW NULL */
          if (LUPSAME(SCRIPT,SLEN, "AME",3))
            { return iNAME; /* signal: set name */
            };
          if (LUPSAME(SCRIPT,SLEN, "EG",2))
            { return iA_NEG;
              // return DoNegate(STACK, STACKSTRING);
            };
          if (LUPSAME(SCRIPT,SLEN, "OT",2))
            { return NotLogic(STACK);
            };
          if (LUPSAME(SCRIPT,SLEN, "OW",2))
            { return TimestampNow(STACK);
            };
          if (LUPSAME(SCRIPT,SLEN, "ULL",3))
            { return PushNull(STACK);
            };
        return -ScErNx;


        case 'O':                                   /* OR */
          if (LUPSAME(SCRIPT,SLEN, "R",1))
            { return OrLogic(STACK);
            };
        return -ScErOx;


        case 'P':                                   /* PAD?? PAPER POPMENU  */
```

```
        if (LUPSAME(SCRIPT,SLEN, "APER",4))
           { return iPAPER; /* signal: set the paper yourself */
           };
        if (LUPSAME(SCRIPT,SLEN, "OPMENU",6))
           { return iPOPMENU; /* signal: pop the menu(s) */
           };
        if (LUPSAME(SCRIPT,SLEN, "USHMENU",7))
           { return iPUSHMENU; // similar to PopMenu
           };
        if (LUPSAME(SCRIPT, SLEN, "RINT", 4))
           { return PrintConsole(refnum, STACK, STACKSTRING);
           }; // print string (only) to console [??]
      return -ScErPx;


      case 'Q':                                  /* QUERY */
        if (LUPSAME(SCRIPT,SLEN, "UERY",4))
           { return iQUERY; /* signal: do your own query */
           };
        if (LUPSAME(SCRIPT,SLEN, "MANY",4))
           { return iSQLMANY; /* signal: you perform SQL many */
           };
        if (LUPSAME(SCRIPT,SLEN, "OK",2))
           { return iOKSQL;  //
           };
      return -ScErQx;


      case 'R':                                  /* RECIPROCAL?? REFRESH REGEX REP
                                                    RETURN ROLLBACK RUN?? */
        if (LUPSAME(SCRIPT,SLEN, "EFRESH",6))
           { return iREFRESH; /* signal: refresh ??!! */
           };
//        if (LUPSAME(SCRIPT,SLEN, "EGEX",4))
//           { return Regex(STACK, STACKSTRING);
//           };
        if (LUPSAME(SCRIPT,SLEN, "EPEAT",5))
           { // assume format was REPEAT(&fxname)
             // main pgm will re-read &fxname, so for now we discard it!! thus:
             Discard(STACK, STACKSTRING);  // a hack.
             return iREPEAT;
           };
        if (LUPSAME(SCRIPT,SLEN, "ETURN",5))
           { return iRETURN;
           };
        if (LUPSAME(SCRIPT,SLEN, "UN",2))
           { return iRUN;
           };
```

```
  if (LUPSAME(SCRIPT,SLEN, "OLLBACK",7))
     { return iROLLBACK; /* signal: rollback SQL */
     };
  if (LUPSAME(SCRIPT,SLEN, "OLLMENU",7))
     { return iROLLMENU; // signal: roll on to copy of this menu!
     };
  if (LUPSAME(SCRIPT,SLEN, "EDRAW",5))
     { return iREDRAW;
     };
  if (LUPSAME(SCRIPT,SLEN, "EPLACE",6))
     { return MyReplace(refnum, STACK, STACKSTRING);
     };
return -ScErRx;


case 'S':                                  /* SAME SAY->ALERT SEND SET SECON
                                              SETX SETZ SKIP SPLIT
                                              STOP SUB SWOP */
  if (LUPSAME(SCRIPT,SLEN, "AME",3))
     { return TestSame(STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "END",3))
     { return iSEND; /* signal: you handle this comm */
     };
  if (LUPSAME(SCRIPT,SLEN, "ECONDS",6))
     { return LocalTimeConvert(STACK, STACKSTRING);
     };
  if (LUPSAME(SCRIPT,SLEN, "ETME",4))
     { return iSETME; /* signal: YOU set this variable */
     };
  if (LUPSAME(SCRIPT,SLEN, "ETTIME",6))
     { return SetTime(STACK);
     };
  if (LUPSAME(SCRIPT,SLEN, "ETX",3))
     { return iSETX; /* signal: you set X value */
     };
  if (LUPSAME(SCRIPT,SLEN, "ETZ",3))
     { return iSETZ; /* signal: you set Z */
     };
  if (LUPSAME(SCRIPT,SLEN, "ET",2))
     { return iSET; /* signal: YOU set this variable */
     };
  if (LUPSAME(SCRIPT,SLEN, "KIP",3))
     { return MySkip(STACK, STACKSTRING); /* signal: skip flow control */
     };
  if (LUPSAME(SCRIPT,SLEN, "PLIT",4))
     { return SplitString(refnum, STACK, STACKSTRING);
     };
```

```
        if (LUPSAME(SCRIPT,SLEN, "TOP",3))
           { return iSTOP; /* signal: stop */
           };
        if (LUPSAME(SCRIPT,SLEN, "UB",2))
           { return iA_SUB;
             // return DoSub(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "WOP",3))
           { return MySwop(STACK, STACKSTRING);
           };
//        if (LUPSAME(SCRIPT,SLEN, "ORT",3))
//           { return DoSort(STACK, STACKSTRING);
//           };
      return -ScErSx;


      case 'T':                                  /* TEXTAFTER??? TEXTBEFORE??? */
        if (LUPSAME(SCRIPT,SLEN, "IMESTAMP",8))
           { return EncodeTimestamp(refnum, STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "IME",3))
           { return EncodeTime(refnum, STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "ITLE",4))
           { return iTITLE;
           };
        if (LUPSAME(SCRIPT,SLEN, "EST",3))
           { return iTEST;
           };
        if (LUPSAME(SCRIPT, SLEN, "ICKS", 4))
           { return GetTicks(refnum, STACK);
           };
        if (LUPSAME(SCRIPT, SLEN, "OGGLE", 5))
           { return iTOGGLE;
           };

      return -ScErTx;


      case 'U':                                  /* UNMARK UPPERCASE */
        if (LUPSAME(SCRIPT,SLEN, "NMARK",5))
           { return ClearMark(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "PPERCASE",8))
           { return DoUppercase(STACK, STACKSTRING);
           };
        if (LUPSAME(SCRIPT,SLEN, "RZN",3))
           { return UnmarkReturnNorZ(STACK, STACKSTRING);
```

```
          };
        if (LUPSAME(SCRIPT,SLEN, "NCACHE", 6)) // uncaching
          { return iUNCACHE;
          };
      return -ScErUx;



      case 'V':                                   /* V */
        if (! SLEN)
          { return iV;  /* signal: V value */
          };
      return -ScErVx;


      case 'W':
      return -ScErWx;


      case 'X':                                   /* X */
        if (! SLEN)
          { return iX;  /* signal: X */
          };
      return -ScErXx;


      case 'Y':
      return -ScErYx;


      case 'Z':
      return -ScErZx;

    //  default:
    //  return -ScErOther;
    };

  return -ScErOther;
}
```

2.1 Various routines

```
/* ============================================================================

/* notes:
  For type conversion we need the following fx:
```

```
    TIMESTAMP() accepts text string timestamp, returns a true timestamp; could als
       a numeric, but we then need a convention. The only one that makes sense is
    (timestamp of time undefined; float is also Julian, integer is ??? undefined ?
    FLOAT accepts text string, or numeric, or integer. Also timestamp which must c
      to Julian. float of time gives seconds after midnight.
    TIME takes text time string, or numeric (seconds after midnight) or even int.
      (submitting timestamp/date is silly).
    DATE similar to timestamp (submitting time is silly; submit datestamp might cl
    VARCHAR should turn others into ASCII
    [already have BOOLEAN which gives integer 0/1]
    NUMERIC performs similar transformations to FLOAT but what SCALE should we use
      --- NUMERIC of FLOAT especially tricky do we truncate / what??? [perhaps mar
      specification of scale]
    INTEGER ??? do we need this. Yep. convert in similar fashion to others, but or
    part, and value must be under 1,000,000,000 AND *POSITIVE*.
*/


/* ========================================================================
/* Rather arbitrary placement here of double to pixel conversion routines:
  (Our reason is that we then avoid importing bulky floating point routines into
  the main program; ok, there are other ways..)
*/

/* PixelX:
   given a decimal portion of window width, convert it to pixels by simply
   multiplying by pixel width provided.
*/

// FlpFloat     _f_mul(FlpFloat, FlpFloat)          FLOAT_EM_TRAP(sysFloatEm_f_mul


// PLAYING AROUND IN AN ATTEMPT TO FIX THE PROBLEM WITH DOUBLES.

/*
Int16 PixelX (UInt16 refnum, double * d, Int16 menuwidth)
{
  Int16 x;

  Char * Ma;
  Char * Mb;
  double * a;
  double * b;

  Ma = xNew(8);
  Mb = xNew(8);
  xCopy(Ma, (Char *)d, 8);
  xCopy(Mb, (Char *)d, 8);
```

```
  a = (double *) Ma;
  b = (double *) Mb;

//  *a = (*(a)) * (*(b));

  Delete(Ma);
  Delete(Mb);


  x = 2;

  if (x < 2)
     { x = 2;
     };
  if (x > 156)
     { x = 156;
     };
  return x;
}
*/


Int16 PixelX (UInt16 refnum, double * d, Int16 menuwidth)
{
  Int16 x;
  x = (Int16)(*d * (double)menuwidth);
  if (x > MenuScrWidth)
     { x = MenuScrWidth;
     };
  return x+MenuLMargin;
}
```

In the following, we return a pixel offset along the Y axis, scaling according to the supplied pixel height of the menu. The submitted d value is the fractional height, where 1.0 is full height.

```
Int16 PixelY (UInt16 refnum, double * d, Int16 menuheight)
{ Int16 y;

  if (menuheight > MenuMaxHt)
     { menuheight = MenuMaxHt;
     };
  y = (Int16)(*d * (double)menuheight);
  if (y < 0)
     { y = 0;
     };
  if (y > MenuMaxHt)
```

```
     { y = MenuMaxHt;
     };
  return y+MenuYOffset;
}
```

The values of MenuMaxHt and MenuYOffset ensure that the top header portion of the PalmOS screen remains unwritten on!

```
Int16 PixelW (UInt16 refnum, double * d, Int16 menuwidth)
{
  Int16 w;
  w = (Int16)(*d * (double)menuwidth);
  if (w < 10)                                    /* we limit to 10 pixels, Palmos
     { w = 10;
     };
  if (w > (MenuScrWidth))
     { w = MenuScrWidth;
     };
  return w;
}
```

In *PixelH*, as with *PixelY* above, we check for sanity, including a check that we haven't included the menu header in our calculations:

```
Int16 PixelH (UInt16 refnum, double * d, Int16 menuheight)
{
  Int16 h;

  if (menuheight > MenuMaxHt)
     { menuheight = MenuMaxHt;
     };
  h = (Int16)(*d * (double)menuheight);
  if (h < 10)
     { h = 10;
     };
  if (h > MenuMaxHt)
     { h = MenuMaxHt;
     };
  return h;
}
```

Note that because we can put an item at a peculiar offset, the above doesn't guarantee that an item will always be completely visible within the window.

```
/* ============================================================================
/* UsefulRead
```

```
   A utility function to read in and convert something.
   mode specifies what to read and what to do.
   Write to dest, returning length as Int16 answer;
   Read from srcP, slen must be appropriate.
*/

Int16 ReadHexDigit(Char * P)
{ Char c;
  c = * P;
  if ((c >= '0') && (c <= '9'))
     { return (c - '0');
     };
  if ((c >= 'a') && (c <= 'z'))
     { c-= 0x20;
     };
  if ((c >= 'A') && (c <= 'F'))
     { return (c - 'A' + 0xA);
     };
  return -ScErBadHexDigit;
};



Int16 ReadHex2Digits(Char * P)
{ // given 2 byte pointer, read a hex number, return 0..255
  Int16 hi;
  Int16 lo;
  hi = ReadHexDigit(P);
  if (hi < 0) { return hi; };
  lo = ReadHexDigit(P+1);
  if (lo < 0) { return lo; };
  return (16*hi + lo);
}



Int32 HexColour (Char * clrstring, Int16 clen)
{  // given colour in string, return 24 bit colour as 32 bit number: 0RGB
   // later on we will also translate 'standard' colour names (text) to numerics!
   // for now, only permissible format of clrstring is RRGGBB (hex digits)
   // we could export this to a lib fx
   // we ASSUME THAT THE LEADING "#" HAS BEEN STRIPPED OFF.

   Int32 r;
   Int32 g;
   Int32 b;  // clumsy use of Int32. ASM better!

   if (clen != 6)                   // RRGGBB
      { return ScErHexColour;
```

```
      };

   r = ReadHex2Digits(clrstring+1);
   if (r < 0) { return r; };
   g = ReadHex2Digits(clrstring+3);
   if (r < 0) { return r; };
   b = ReadHex2Digits(clrstring+5);
   if (r < 0) { return r; };

   return ( (r<<16) + (g<<8) + b );
}




Int16 UsefulRead (UInt16 refnum, Int16 mode, Char * dest, Int16 dlen, Char * srcP,
{
  Int32 i;

  switch(mode)
    {
      case HEXADECIMALCOLOUR:
        if (dlen < 4)
           { return -ScErBadUReadLen;
           };
        i = HexColour(srcP, slen);
        if (i < 0)
           { return ((Int16) i);                 // coerce into 16 bit -ve!
           };
        * ((Int32 *)(dest)) = i;                 // store value
      return 4;                                  // 4 byte length

      /* default:
        return -ScErBadReadMode;                 //
      */
    };

  return -ScErBadReadMode;
}



/* ============================================================================
/* DoFloat:
   Convenient (clumsy) method of doing floating point arithmetic without all the
   overheads of importing the float code into the main program!
   Rather nasty, actually.
   The first operand and destination is a double stored at d;
```

```
   The second operand depends on the mode, and is stored at sourceP.
*/


Int16 DoFloat (UInt16 refnum, Int16 mode, double * d, Char * sourceP)
{
  Int16 itgr;

  switch(mode)
    {
      case MULTIPLYINTEGER:
        itgr = *((Int16 *)(sourceP));           //
        * d *= (double) itgr;                   // clumsy but more explicit.
      return 1;

      /* default:
        return -ScErDoFloatMode;
      */
    };

  return -ScErDoFloatMode;
}



/* ============================================================================
// &Function handling.
// Current exported functions are:
//   1. IniFx
//   2. SearchFxBody

// Scripted functions begin with an '&'.
// All function definitions are contained in the PalmOS database: FUN.PDB
// As this is one of our SQL databases, we can interrogate it via an SQL query,
// but this approach is slow and clumsy.
// We will therefore do the following:
// 1. At startup, create a buffer area called FxBUFFER. It will be up to 32K in si
//    although at present we only need about 2K.
// 2. Examine each record in FUN (from 1 onwards), and keep details in FxBUFFER th
//     Offset  Details
//    a. +0    a 4-byte number : a locked pointer to the relevant record
//    b. +4    offset of fx name
//    c. +6    length of fx name
//    d. +8    offset of fx body
//    e. +A    length of fx body
//    f. +C    index of record in PalmOS database (? need me)
//    g. +E    (4 spare bytes)
///   EACH RECORD HAS 16 BYTES IN FxBUFFER.
```

```
//
//    THE RECORDS ARE SORTED BY NAME; we also keep FxBUFFER open;
// 3. When we invoke a fx, we use a binary search of FxBUFFER to locate the record
//
// NOTE that when we invoke a function, we must keep a record of the prior functio
// AS WELL AS the current evaluation offset within that function. We create FxSTAC
// which devotes 4 bytes to each fx: 2 for the index of the function into FxBUFFER
// and 2 for the offset.

// We now define functions to:
// a. Initialise FxBUFFER (partially performed in caller)
// b. [ don't need "Close FxBUFFER" as performed in caller ]
// c. Perform a binary search on FxBUFFER
// d. Invoke a function
// e. RETURN from a function


//-------------------------------------------------------------------------------
// CompareItems
// Index into given buffer, and thus compare items.
//
  // return 0 if equal, -1 if left<right, +1 if left > right
   // DBG argument is just a debugging aid to ensure that we don't transgress
   // upper or lower bounds of buffer.

Int16 CompareItems (Int16 left, Int16 right, Char * fbuffer, Int16 DBG )
{
   Char * L;
   Char * R;
   Int16 llen;
   Int16 rlen;
   Int16 loff;
   Int16 roff;

   if (left < 0)
      { return -ScErLowLeftCompare; // ERRDEBUG ("ERROR LOW left compare",left);
      };
   if (right < 0)
      { return -ScErLowRightCompare; // ERRDEBUG ("ERROR LOW right compare",right)
      };
   if (left >= DBG)
      { return -ScErHiLeftCompare; // ERRDEBUG ("ERROR HI left compare",left);
      };
   if (right >= DBG)
      { return -ScErHiRightCompare; // ERRDEBUG ("ERROR HI right compare",right);
      };

   L = fbuffer + XVI*left;
```

```
   R = fbuffer + XVI*right;
   loff = * ((Int16 *)(L+4));  // get offset of name
   llen = * ((Int16 *)(L+6));  // and its length
   roff = * ((Int16 *)(R+4));  // likewise for
   rlen = * ((Int16 *)(R+6));  //   second item.
   L = *((Char **)(L));        // get actual L pointer
   R = *((Char **)(R));        //   and for R.
   L += loff;
   R += roff;                  // actually point to the strings
   return (BetterCompare(L, llen, R, rlen));
}



//-------------------------------------------------------------------------------
// FindPosition:
  // the following assumes that NO TWO NAMES ARE THE SAME
// given a new item, find correct insertion position.
  // fbuffer contains 16 byte details of items, A is array of sorted items
  // alen is number of WORDS in A.

Int16 FindPosition( Char * A, Int16 alen, Int16 itm, Char * fbuffer, Int16 DBG)
{ Int16 i;
  Int16 c;
  Int16 left=0;
  Int16 right=alen-1;
  // idea is to progressively narrow search range:

  while (left < right)
    {
      i = (left+right)/2;                              // truncate
      // woops. leaving out the parenthesis was a disaster!
      c = CompareItems ( *((Int16*)(A+2*i)), itm, fbuffer, DBG);
      if (c < -1) // failure!! [clumsy, fix me]
         { return c;
         };
      if (c < 0)                                     // itm is above i-item
         { left = i+1;
         } else
         { right = i-1;
         };
    };
  // IF right<left, implies we must insert to immediate left of the left item;
  // IF right==left, then there is still one more comparison to make:
  //     based on this, we either insert to left or right of current (l=r) item!
  if (right < left)
     { return (left);
     };
  c = CompareItems ( *((Int16*)(A+2*left)), itm, fbuffer, DBG);
```

```
  if (c < -1) // failure!! [fix as above]
     { return c;
     };
  if (c < 0)                                    // if itm is the greater string
     { return (left+1);                         // insert to right of current ite
     };
  return left;                                  // otherwise, 'at' current (on le
}



//-------------------------------------------------------------------------------
// SortFx:
  // given an array of 16byte descriptors(fbuffer), their length, and a sort area
   // (2 bytes per item), provided sorted index in sort area.
   // For format of fbuffer, see IniFxBuffer in sql.hpp.

Int16 SortFx(Char * sortarea, Int16 fblen, Char * fbuffer)
{ Int16 pos;
  Int16 i=1;
  Int16 ok;

  *((Int16 *)sortarea) = 0;                     // kick off with first item (inde
  while (i < fblen)
    { pos=FindPosition( sortarea, i, i, fbuffer, fblen);
      if (pos < 0)
         { return pos; // fail
         };
      ok = InsertItem(sortarea, i, pos, i);
      if (ok < 0)
         { return ok;
         };
      i ++;
    };
  return 1;                                     // 'success'
}

// for in-place quicksort see:
// http://ciips.ee.uwa.edu.au/~morris/Year2/PLDS210/qsort1a.html
// also look at: http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/quick/q
// http://en.wikipedia.org/wiki/Quicksort
// Here's a cute Shell sort: http://mainline.brynmawr.edu/Courses/cs206/spring2004

// Sorting:
// We will use a binary insertion sort.
// This is technically O(n^2), but there are several reasons why it's a good sort
//    1. Comparisons are expensive relative to insertions;
//    2. We have limited memory;
```

```
//    3. We don't have large numbers of items to sort.

// The plan is:
//   a. create an empty list of length n
//   b. insert the first item
//   c. insert the next item at the appropriate point, using a binary search
//      to locate the position
//   d. continue applying (c) to each item in turn.
// See: http://www.brpreiss.com/books/opus4/html/page491.html
// check out: http://primates.ximian.com/~fejj/blog/archives/000013.html




//---------------------------------------------------------------------------
// IniFx:
//
// Go through each record in FUN.PDB (Name is simply FUN) and
//    store relevant data.

Int16 IniFx(UInt16 SCRIPTLIBCODE,
            Char * FxBUFFER, DmOpenRef FUNFILE, Int16 FxCOUNT,
            Char * FxSORTED)
{
  // EXAMINE EACH RECORD IN TURN, keep name;
  // The order of the fx and names in FUN is key|body|name.
  //

  // FINALLY CREATE INDEXED SORT OF THE NAMES:
  // ERRDEBUG("Functions loaded = ", FxCOUNT);

  Int16 ok;
  ok = SortFx(FxSORTED, FxCOUNT, FxBUFFER);
  if (ok < 0)
     { return ok;
     };
  return 1;
}




//---------------------------------------------------------------------------
// SearchFxBody:
//
// given function name, locate its body string and return a pointer to this
// AND ALSO RETURN A BYREF LENGTH!
// similar to FindPosition.
```

```
Char * SearchFxBody(UInt16 SCRIPTLIBCODE,
                    Char * name, Int16 nlen,
                    Int16 * BODYLEN,                      // note byref value!
                    Char * FxSORTED, Int16 FxCOUNT,
                    Char * FxBUFFER)
{ Int16 i;
  Int16 c;
  Int16 left=0;
  Int16 right=FxCOUNT-1;

  Char * thisitm;
  Char * thisname;
  Int16 idx;
  Int16 thislen;
  Int16 thisoff;
  Int16 bodyoff;

  while (left <= right)
    {
      i = (left+right)/2;
      idx = *((Int16*)(FxSORTED+2*i));          // get reference to i-th item
      thisitm = FxBUFFER + 16*idx;              // point to description in FXBUFF
      thisoff = *((Int16 *)(thisitm+4));        // get offset of name from +4
      thislen = *((Int16 *)(thisitm+6));        // get length of name from +6
      thisname = *((Char **)(thisitm));         // get actual locked ptr to row
      thisname += thisoff;                      // cumbersome: point to name
      c = BetterCompare(thisname, thislen, name, nlen);
      if (c < 0)                                // thisitm is less, name is to ri
         { left = i+1;
         } else
         { if (c > 0)                           // thisitm is greater, name is to
              { right = i-1;
              } else                            // the two are identical
              { bodyoff = *((Int16 *)(thisitm+8));  // get offset of body
                *(BODYLEN) = *((Int16 *)(thisitm+0xA));// get length of body
                return (thisname - thisoff + bodyoff); // return pointer to body!
              };
         };
    };
  return 0;                                    // signal failure
}




/* ========================================================================
// HANDLING OF LOCAL VARIABLES:
```

```
// we use a local buffer to store and retrieve named variables.
// Maximum name length is 32 characters
// Maximum number of variables is just 16.
// for definitions of MAXNAMELENTH MAXVARS STARTNAMELENGTHS STARTNAMES
//                    STARTVARIABLES and STARTLOCALSTACKSTRING see SCRIPTING.h

// We need space thus:
// +0:           16 bytes for some data (8 words):
//     +0           oSTART [unused, now]
//     +2           = oTOP [top of this buffer = TOP OF LOCAL *STACKSTRING* !!
//                    When clear variables, this is cleared to STARTLOCALSTACKTRIN
//     +4           oMAX = maximum buffer size (?! for now, 4096)
//     +6           oFLAGS = [unused, now]
//     +8..+F       = 0's (reserved)
// +16:          16 bytes for lengths of names (as max name length is just 32, 1 b
// +32:          512 bytes for names. Each name starts at offset divisible by 32
// +32+512:      256 bytes for the actual variables, (copied directly from stack)
// +32+512+256:  4096-(32+512+256) bytes for stackstring values (LOCALSTACKSTRING)


//-------------------------------------------------------------------------------
// ClearBufferArea:
//
// Utility function to clear a buffer area to zeroes.
//   IT IS THE RESPONSIBILITY OF THE CALLER to arrange that the length is sufficie
// The following clumsy function ASSUMES that BUF exists
// [check me: are there potential integral boundary problems]

Int16 ClearBufferArea (Char * BUF, Int16 bufoff, Int16 buflen, Int16 bufmax)
{
  Char * null8 = "\x0" "\x0" "\x0" "\x0" "\x0" "\x0" "\x0" "\x0";
    // [fix me: make the above global const]

  while (buflen > 8)                              // peel off 8 bytes at a time
    { w_DmWrite(BUF, bufoff, null8, 8, bufmax);  // might test for success
      bufoff += 8;
      buflen -= 8;
    };

  while (buflen > 0)                              // fix up remaining few bytes
    { w_DmWrite(BUF, bufoff, null8, 1, bufmax);     // might test for success
      bufoff ++;
      buflen --;
    };
  return 1;    // ok
}


//-------------------------------------------------------------------------------
```

```
//  ClearVariables
//
// Clear all variables:
//   1. clear name lengths (0=no name present)
//   2. Clear local stackstring top
//   (that's it)!

Int16 ClearVariables(UInt16 SCRIPTLIBCODE, Char * LOCALS)
{ Int16 ss = STARTLOCALSTACKTRING;
  Int16 lMAX = *((Int16 *)(LOCALS+oMAX));

  ClearBufferArea (LOCALS, STARTNAMELENGTHS, MAXVARS, lMAX);
    // by clearing name lengths, we indicate names are free!
  w_DmWrite(LOCALS, oTOP, &ss, 2, lMAX);  // clear string area!
  return 1;
}


//-------------------------------------------------------------------------
// IniVariables:
// Intialise variable buffer

Int16 IniVariables (UInt16 SCRIPTLIBCODE, Char * LOCALS, Int16 LOCALSIZE)
{
  if (! LOCALS)
     { return -ErNoLocalBuffer;
     };
  if (LOCALSIZE < STARTLOCALSTACKTRING)          // if ridiculously small
     { return -ErNoLocalBuffer;
     };

  w_DmWrite(LOCALS, oMAX, &LOCALSIZE, 2, LOCALSIZE);
  ClearVariables (0, LOCALS);

  return 0;
}


//-------------------------------------------------------------------------
// FindVariable:
//   given pointer to name, and length, and pointer to LOCALS:
//   identify name and return its index (0..15)

Int16 FindVariable (Char * varname, Int16 varlen, Char * LOCALS)
{
  Int16 ilen;
  Int16 i;

  i = 0;
```

```
  while (i < MAXVARS)
    { ilen = *(LOCALS + STARTNAMELENGTHS + i);
      if (ilen == varlen)
         { if (xSame (varname, ilen, LOCALS+STARTNAMES+ (i*MAXNAMELENGTH), ilen))
              { return i; // return index of the name
              };
         };
      i ++;
    };
  return -1;  // fail
}


//-------------------------------------------------------------------------------
// FindEmptyVariable:
/// find next free space for a variable

Int16 FindEmptyVariable (Char * LOCALS)
{
  Int16 i;

  i = 0;
  while (i < MAXVARS)
    { if ( *(LOCALS + STARTNAMELENGTHS+i) == 0 )   // or ! ...
         { return i; // return index of the empty name
         };
      i ++;
    };
  return -1;  // fail
}



//-------------------------------------------------------------------------------
// GetVariable:
//
// given string in format "[varname]"
// 1. look up the name
// 2. place value on STACK
// [FIX ME! we should rewrite this and perl code to $(varname) ipo $[varname]  ]

Int16 GetVariable (UInt16 SCRIPTLIBCODE,
                   Char * varstring, Int16 varlen, Char * LOCALS,
                   Char * STACK, Char * STACKSTRING )
{
   Int16 idx;
   Int16 top;
   Int16 Stop;
   Int16 Storestart;
```

```
   Int16 storelen;
   Int16 S;

   //0. check format
   if ((!varstring) || (varlen < 3))
      { return -ErrGetVarName;
      };
   varlen --; // for square bracket
   if (  ( *(varstring) != '[' )
       ||( *(varstring+varlen) != ']' )
      )
      { return -ErrGetVarName;
      };
   varlen --; // for square bracket
   varstring ++; // past '['

   // 1. find name (look at length; if the same, match name using length)
   idx = FindVariable(varstring, varlen, LOCALS);
   if (idx < 0)
      { return -ErrVarNotFound;
      };

   // 2. find corresponding variable, and write to stack and stackstring if requir
   S = STARTVARIABLES + idx*XVI;
   top = *((Int16 *)(STACK+oTOP));
   w_DmWrite(STACK, top, LOCALS+S, XVI, SMAX);  // copy over variable to stack
   if ( (0x0F & (*(STACK+top+14))) > 14 ) // if long variable
      { Stop = *((Int16 *)(STACKSTRING+oTOP));
        storelen = *((Int16 *)(STACK+top)); // at +0 we store length
        Storestart = *((Int16 *)(STACK+top+2)); // at +2 we store offset
        if (! w_DmWrite(STACKSTRING, Stop, LOCALS+Storestart, storelen, MAXSS))
           { return -ErrGetFullStack;
           };  // probably best to check for overrun test outcome!
        w_DmWrite(STACK, top+2, &Stop, 2, SMAX); // write new offset in STACKSTRIN
        Stop += storelen;
        w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS);
      };
   top += XVI;
   w_DmWrite(STACK, oTOP, &top, 2, SMAX);
  return 1;  // ok.
}


//-------------------------------------------------------------------------------
// CheckVariable:
//
// similar to GetVariable but DOES NOT RETURN A VALUE. Simply checks whether
// the variable exists, and returns 1=yes, 0=no.
// returns 1/0 on the stack!
```

```
// Submit the name ON THE STACK as a 'V' string

Int16 CheckVariable (UInt16 SCRIPTLIBCODE,
                     Char * LOCALS,
                     Char * STACK, Char * STACKSTRING )
{
  Int16 bottom;
  Int16 top;
  Int16 vlen;
  Int16 Stop;
  Char c;
  Char * P;
  Int32 i;

  bottom = *((Int16 *)(STACK+oSTART));
  top = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     { return -ScErVarCheckEmpty;
     };
  top -= XVI;
  if (*(STACK+top+15) != 'V')
    { return -ScErVarType;
    };
  vlen = 0x0F & *(STACK+top+14);
  if (vlen > 14)
     { vlen = *((Int16 *)(STACK+top));
       Stop = *((Int16 *)(STACK+top+2));
       P = STACKSTRING + Stop;
       w_DmWrite(STACKSTRING,oTOP, &Stop, 2, MAXSS); // update stackstring
     } else
     { P = STACK+top;
     };

   i = FindVariable(P, vlen, LOCALS);
   if (i < 0)
      { i = 0;
      } else
      { i = 1;
      };
   w_DmWrite(STACK, top, &i, 4, SMAX); // write integer result (1/0)
   c = 'I';
   w_DmWrite(STACK, top+15, &c, 1, SMAX); // new type is 'I'
   c = 4;
   w_DmWrite(STACK, top+14, &c, 1, SMAX); // new length thus must be 4
  return 1; //ok.
}


//-------------------------------------------------------------------------------
```

```
// MakeVariable:
//
// given name, create the variable!

Int16 MakeVariable (UInt16 SCRIPTLIBCODE, Char * STACK, Char * STACKSTRING, Char *
{
  Int16 idx;
  Int16 N;
  Char c;
  Int16 lMAX;
  Char * varname;
  Int16 varlen;
  Int16 top;
  Int16 bottom;
  Int16 Stop;
  Int16 D;
  Char * null8 = "\x0" "\x0" "\x0" "\x0" "\x0" "\x0" "\x0" "\x0";

  lMAX = *((Int16 *)(LOCALS+oMAX));

  // the following might be made into a common rtn cf SetVariable:
  // [fix me?]
  // -1. get varname, varlen: name of variable WILL be on the STACK!
  top = *((Int16 *)(STACK+oTOP));
  bottom = *((Int16 *)(STACK+oSTART));
  if (top <= bottom)
     { return -ScErMakePop;
     };
  top -= XVI;
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);        // pop the item
  varlen = 0x0F & *(STACK+top+14);
  if ( varlen < 15)                              // if not a long name
     { varname = STACK+top;
     } else
     { varlen = *((Int16 *)(STACK+top+0));
       Stop = *((Int16 *)(STACK+top+2));
       varname = STACKSTRING+Stop;
       w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS); // pop stackstring too!
     };

  // 0. some checks:
  if ( (varlen < 1) || (varlen > MAXNAMELENGTH) )
     { return -ErrVarBadName;
     };
     // hmm might also check name syntax, but we are lax.

  // 1. set variable name:
  //    (a) check it doesn't already exist!
```

```
   idx = FindVariable(varname, varlen, LOCALS);
   if (idx > -1)
      { return -ErrVarAlreadyExists;
      };
  //    (b) make sure there is space
   idx = FindEmptyVariable(LOCALS);
   if (idx < 0)
      { return -ErrVarsFull;
      };
  //    (c) copy over name
  N = STARTNAMES + (idx*MAXNAMELENGTH);   // start of place to insert new name
  w_DmWrite(LOCALS, N, varname, varlen, lMAX);  // copy name
  c = varlen; // must be 1 byte long
  w_DmWrite(LOCALS, STARTNAMELENGTHS+idx, &c, 1, lMAX);

  // SET INITIAL DEFAULT VALUE OF VARIABLE TO *NULL*:
  // Note there is an issue here: We haven't typed the variables. [FIX ME?????????
  // .. ok for perl, but we must be more strict????

    D = STARTVARIABLES + idx*XVI;
    w_DmWrite(LOCALS, D, null8, 8, lMAX);
    w_DmWrite(LOCALS, D+8, null8, 7, lMAX); // write null value!
    c = 'V';
    w_DmWrite(LOCALS, D+15, &c, 1, SMAX); // character type is standard null


  return 1; //ok
}


//-------------------------------------------------------------------------------
// SetVariable:
//
// Given name of variable, get value from stack to our local variable buffer.

Int16 SetVariable (UInt16 SCRIPTLIBCODE,
             Char * STACK, Char * STACKSTRING, Char * LOCALS )
{
  Int16 idx;
  Int16 D;
  Int16 top;
  Int16 bottom;
  Int16 stringoff;
  Int16 stringlen;
  Int16 localStop;
  Char * varname;
  Int16 varlen;
  Int16 Stop;
  Int16 lMAX;
```

```
lMAX = *((Int16 *)(LOCALS+oMAX));

// the following might be made into a common rtn cf MakeVariable:
// [fix me?]
// -1. get varname, varlen: name of variable WILL be on the STACK!
top = *((Int16 *)(STACK+oTOP));
bottom = *((Int16 *)(STACK+oSTART));

top -= XVI;  // NOTE THIS CHECKS FOR MINIMUM OF 2 variables on stack!
if (top <= bottom)
   { return -ScErMakePop;
   };

varlen = 0x0F & *(STACK+top+14);
if ( varlen < 15)                              // if not a long name
   { varname = STACK+top;
   } else
   { varlen = *((Int16 *)(STACK+top+0));
     Stop = *((Int16 *)(STACK+top+2));
     varname = STACKSTRING+Stop;
     w_DmWrite(STACKSTRING, oTOP, &Stop, 2, MAXSS); // pop stackstring too!
   };

// 1. find variable:
 idx = FindVariable(varname, varlen, LOCALS);
 if (idx < 0)
    { return -ErrVarNotFound;
    };

// 2. pop value off stack, setting stack top etc.
   // (a) copy over 16 bytes from stack
D = STARTVARIABLES + idx*XVI;
top -= XVI; // pop *ANOTHER* item
w_DmWrite(STACK, oTOP, &top, 2, SMAX);
w_DmWrite(LOCALS, D, STACK+top, XVI, lMAX);  // keep string
   // (b) if extended string, copy over this and fix up offsets
if (  (0x0F & *(STACK+top+14)) > 14  )
   { stringoff = *((Int16 *)(STACK+top+2));
     stringlen = *((Int16 *)(STACK+top+0));
     localStop = *((Int16 *)(LOCALS+oTOP)); // get current top
     if (! w_DmWrite(LOCALS, localStop, STACKSTRING+stringoff, stringlen, lMAX))
        { return -ErrLocalStringFull;
        };
     w_DmWrite(LOCALS, D+2, &localStop, 2, lMAX);  // write new offset
     localStop += stringlen;
     w_DmWrite(LOCALS, oTOP, &localStop, 2, lMAX); // new top (of string area)
     w_DmWrite(STACKSTRING, oTOP, &stringoff, 2, MAXSS); //string now off stacks
```

```
     };
  return 1;  // ok.
}


/*   ============================================================================
// Bypass:
//   Directly access a command (as specified in iCode) without submitting a script
//   Predominantly used by SQL routines to invoke post-processing SORT (ORDER BY),
//   and DISTINCT.
// Nature of commands is specified in modifier, and also in aux string, where requ
// alen is length of aux string.

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

  switch (iCode)
    {
      case iDISCARD:
       return Discard(STACK, STACKSTRING);

      case iINTEGER:
       return EncodeInteger (refnum, STACK, STACKSTRING);

//      case iCOPY:  // used in debugging..
//        if ( MyCopy(refnum, STACK, STACKSTRING, 0) >= 0)
//           { return 1;
//           };
//        return 0;

      case iNULL:
      return PushNull(STACK);

      case iMARK:
        i = modifier;
        PushItem (0, STACK, STACKSTRING, (Char *) &i, 4, 'I', 0); // push mark dep
        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 -ScErBypass; };
//        return DoMax(STACK, STACKSTRING);
//        // In our initial implementation of SELECT, we will only permit "SELECT
//        // without group by. This is translated into "SELECT column" *with* a po
//        // 'max' flag set. After values have been extracted, the max flag is exa
//        // 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 -ScErBypass; };
//        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 (0, STACK, STACKSTRING, (Char *) &i, 4, 'I', 0); // push intege
//        return DoDistinct(STACK, STACKSTRING);
//        // DISTINCT execution in initial SELECT is similar to MAX above.
//        // We only allow distinct on one column, at present.

//      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 EXPO
//        PushItem (0, STACK, STACKSTRING, aux, alen, 'V', 0); // push aux deeper
//        PushItem (0, STACK, STACKSTRING, (Char *) &i, 4, 'I', 0); // push intege
//        return DoSort(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 spec
//        // if we say "SORT BY tablename.columname DESC").
//        // Similar rules apply for mark and unmark.

      default:
      return -ScErBypass;

    };

}
```

```
// ============================================================================
// PRE-PROCESSING:

//----------------------------------------------------------------------------
// Intial checking function:

Int16 CHECKsame (Char * P0, Int16 maxP0, Char * p1, Int16 lgth)
{
  if (maxP0 < lgth)
     { return 0;
     };
  return LUPSAME (P0, lgth, p1, lgth);
}


//----------------------------------------------------------------------------
// Preprocess:
// Pre-processing of SQL statements.
// Slows things down a bit, but a bit softer on formatting. Translates into a form
// that subsequent parsing recognises. All of the following applies only OUTSIDE
// SQL 'quotes'
//
// NB. WE MIGHT PREPROCESS SQL STATEMENTS AS THEY EXIST IN FUNCTIONS ETC. THIS WOU

// Processing 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 "(("
//   7. [must still check out NOT very carefully ??????????????????????????]
//   8. "SELECT DISTINCT" becomes just SELECT, and the DISTINCT post-processor is
//      if distinct used, then at present only one column must be returned, althou
//      this will later be modified to allow several. At present we submit col cou
//   9. "SELECT MAX(columname) BECOMES SELECT columname, and MAX post processor is
//   10. likewise for SELECT MIN(..)
//   11. "... 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.
//   12. "... ORDER BY colname DESC" is similar to (11) but submit "D" sort config
//   13. 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 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
// POST-PROCESSOR SIGNALS:
         //   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
         // cannot have sort and max or distinct (?) and max
   //   (3) +2 Next byte is length of following sort configuration string
   //   (4) +3 up to 13 bytes devoted to sort configuration string
   //   SO actual string to interpret follows from byte 16 onwards!

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

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

   return 0;  // HAS BEEN REMOVED TO SQL3.c
}

Int16 SeekText (UInt16 refnum,
                Char * current, Int16 clen, Char * target, Int16 tlen, Char mode)
{ return 0;
}

Int16 SeekNumber (UInt16 refnum,
                Char * current, Int16 clen, Char * target, Int16 tlen, Char mode)
{ return 0;
}

Int16 SeekFloat (UInt16 refnum,
                 Char * current, Char * target, Char mode)
{ return 0;
}

Int16 PackConditions (UInt16 refnum,
                 Char * dest, Int16 destmax, Char * sqsrc, Int16 srclen)
{ return 0;
}
```

```
/*   ============================================================================
/*                                        THE END
/*   ============================================================================
```

3 Header file: SCRIPTING.h

```
#ifndef SCRIPTING_H
#define SCRIPTING_H

#include <LibTraps.h>
#include <FloatMgr.h>

// modes for reading data:
#define HEXADECIMALCOLOUR 1

// miscellaneous:

#define MAXSTRINGLENGTH 4096
  // rather arbitrary upper limit (current) on string length ??

#define MAXNAMELENGTH 32
#define MAXVARS 16
#define STARTNAMELENGTHS 16
#define STARTNAMES STARTNAMELENGTHS + MAXVARS
#define STARTVARIABLES STARTNAMES + MAXNAMELENGTH*MAXVARS
#define STARTLOCALSTACKTRING STARTVARIABLES + 16*MAXVARS



#define MAXIMUMSCALE 16
#define MAXIMUMFLOATLENGTH 32
// current maximum length of ascii float string ???

// modes for DoFloat:
#define MULTIPLYINTEGER 1
```

Function codes are of particular importance as they are used to transmit vital signals back to the calling program. Take not of iFUNCTION and eFUNCTION in particular, which are respectively used to signal that a user function is either being called (&) or *replacing* (=) the current function.

```
// function codes:

#define  iNORMAL     1

#define  iSKIP       28
#define  iFUNCTION   3
#define  iRETURN     21
#define  iVARNAME    2
#define  iREPEAT     20
```

```
#define  iKEY        10
#define  iQUERY      17
#define  iDOSQL      6
#define  iME         12
#define  iSQLMANY    29
// #define  iFORCEX     7
#define  iIAM        8
#define  iNAME       14
#define  iSEND       24
#define  iSET        25
#define  iSETX       26
#define  iSETZ       27
#define  iV          31
#define  iINK        9
#define  iPAPER      15
#define  iCOMMIT     33
#define  iROLLBACK   22
#define  iLABEL      11
#define  iMENU       13
#define  iPOPMENU    16
#define  iREFRESH    19
#define  iALERT      23
#define  iASK        4
#define  iCONFIRM    5
#define  iSTOP       30
#define  iQUIT       18
#define  iDEBUG      35
#define  iFAIL       38
#define  iENABLED    39
#define  iDISABLED   40
#define  iRUN        41
#define  iX          42
// #define  iCOPY       49
#define  iOKSQL      51
#define  iDUMP       52
#define  iINTEGER    53
#define  iDISCARD    54
#define  iPUSHMENU   55
#define  iDEPTH      56
#define  iISNAME     57
#define  iNULL       58
#define  iREDRAW     59

#define  iSORT       43
#define  iDISTINCT   44
#define  iMAX        45
#define  iMIN        46
#define  iMARK       47
```

```
#define  iUNMARK     48
#define  iUNMARKONLY 50

#define  iCONSOLE     60
#define  eFUNCTION    61
#define  iCOMPENSATED 62
#define  iROLLMENU    63
#define  iLINESLEFT   64
#define  iSETME       65
#define  iTITLE       66

#define iTEST         67
#define iCACHE        68
#define iUNCACHE      69
#define iTOGGLE       70

// iMARK and iUNMARK are no longer needed. See fx in SCRIPTING.C


/* Note that the following must be evaluated by the preprocessor.  In particular,
   they must be #defines not enum values.  In fact, it's perfectly valid
   just to write SYS_TRAP(sysLibTrapCustom+1) etc in the function
   declarations below.  */

#define SCRIPTINGString_trapno  sysLibTrapCustom+0
#define SCRIPTINGPushItem       sysLibTrapCustom+1
#define SCRIPTINGPopItem        sysLibTrapCustom+2
#define SCRIPTINGDoScript       sysLibTrapCustom+3
#define SCRIPTINGResolve        sysLibTrapCustom+4
#define SCRIPTINGPeek           sysLibTrapCustom+5
#define SCRIPTINGEncode         sysLibTrapCustom+6
#define SCRIPTINGPixelX         sysLibTrapCustom+7
#define SCRIPTINGPixelY         sysLibTrapCustom+8
#define SCRIPTINGPixelW         sysLibTrapCustom+9
#define SCRIPTINGPixelH         sysLibTrapCustom+10
#define SCRIPTINGUsefulRead     sysLibTrapCustom+11
#define SCRIPTINGDoFloat        sysLibTrapCustom+12
#define SCRIPTINGSeekText       sysLibTrapCustom+13
#define SCRIPTINGSeekNumber     sysLibTrapCustom+14
#define SCRIPTINGSeekFloat      sysLibTrapCustom+15
#define SCRIPTINGPackConditions sysLibTrapCustom+16
#define SCRIPTINGIniFx          sysLibTrapCustom+17
#define SCRIPTINGSearchFxBody   sysLibTrapCustom+18
#define SCRIPTINGIniVariables   sysLibTrapCustom+19
#define SCRIPTINGGetVariable    sysLibTrapCustom+20
#define SCRIPTINGSetVariable    sysLibTrapCustom+21
#define SCRIPTINGMakeVariable   sysLibTrapCustom+22
#define SCRIPTINGClearVariables sysLibTrapCustom+23
```

```
#define SCRIPTINGBypass         sysLibTrapCustom+24
#define SCRIPTINGpreprocess     sysLibTrapCustom+25
#define SCRIPTINGcheckvariable  sysLibTrapCustom+26
#define SCRIPTINGpeekdepth      sysLibTrapCustom+27
#define SCRIPTINGGetItemStyle   sysLibTrapCustom+28
#define SCRIPTINGPassConsole    sysLibTrapCustom+29
#define SCRIPTINGPassBug        sysLibTrapCustom+30
#define SCRIPTINGPassMathlib    sysLibTrapCustom+31


#ifndef SCRIPTING_TRAP
#define SCRIPTING_TRAP(trapno)  SYS_TRAP(trapno)
#endif

Err SCRIPTINGOpen (UInt16 refNum)
    SCRIPTING_TRAP(sysLibTrapOpen);

Err SCRIPTINGClose (UInt16 refNum, UInt16 *numappsP)
    SCRIPTING_TRAP(sysLibTrapClose);

Int16 ScriptIni (UInt16 refnum, Char * STACK, Char * STACKSTRING)
    SCRIPTING_TRAP(SCRIPTINGString_trapno);

Int16 PushItem (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * itm, Int16
 Char itype, Int16 scale)
    SCRIPTING_TRAP(SCRIPTINGPushItem);

Int16 PopItem (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * itm, Int16
    SCRIPTING_TRAP(SCRIPTINGPopItem);

Int16 DoScript (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * script, In
    SCRIPTING_TRAP(SCRIPTINGDoScript);

Int16 Resolve(UInt16 refnum, Char * dest, Int16 destlen, Char * STACK, Char * STAC
    SCRIPTING_TRAP(SCRIPTINGResolve);

Int16 StackPeek (UInt16 refnum, Char * STACK, Char * STACKSTRING, Char * dest, Int
    SCRIPTING_TRAP(SCRIPTINGPeek);

Int16 EncodeAll (UInt16 refnum, Char * destin, Int16 destsize,
                 Char * datum, Int16 datlen,
                 Char typeofdatum, Int16 scale)
    SCRIPTING_TRAP(SCRIPTINGEncode);

Int16 PixelX (UInt16 refnum, double * d, Int16 w)   SCRIPTING_TRAP(SCRIPTINGPixelX
Int16 PixelY (UInt16 refnum, double * d, Int16 w)   SCRIPTING_TRAP(SCRIPTINGPixelY
Int16 PixelW (UInt16 refnum, double * d, Int16 w)   SCRIPTING_TRAP(SCRIPTINGPixelW
Int16 PixelH (UInt16 refnum, double * d, Int16 w)   SCRIPTING_TRAP(SCRIPTINGPixelH
```

```
Int16 UsefulRead (UInt16 refnum, Int16 mode, Char * dest, Int16 dlen, Char * srcP,
    SCRIPTING_TRAP(SCRIPTINGUsefulRead);

Int16 DoFloat (UInt16 refnum, Int16 mode, double * d, Char * sourceP)
    SCRIPTING_TRAP(SCRIPTINGDoFloat);

Int16 SeekText (UInt16 refnum,
                Char * current, Int16 clen, Char * target, Int16 tlen, Char mode)
    SCRIPTING_TRAP(SCRIPTINGSeekText);

Int16 SeekNumber (UInt16 refnum,
                Char * current, Int16 clen, Char * target, Int16 tlen, Char mode)
    SCRIPTING_TRAP(SCRIPTINGSeekNumber);

Int16 SeekFloat (UInt16 refnum,
                 Char * current, Char * target, Char mode)
    SCRIPTING_TRAP(SCRIPTINGSeekFloat);

Int16 PackConditions (UInt16 refnum,
                 Char * dest, Int16 destmax, Char * sqsrc, Int16 srclen)
    SCRIPTING_TRAP(SCRIPTINGPackConditions);

Int16 IniFx(UInt16 SCRIPTLIBCODE,
            Char * FxBUFFER, DmOpenRef FUNFILE, Int16 FxCOUNT,
            Char * FxSORTED)
    SCRIPTING_TRAP(SCRIPTINGIniFx);

Char * SearchFxBody(UInt16 SCRIPTLIBCODE,
                    Char * name, Int16 nlen,
                    Int16 * BODYLEN,
                    Char * FxSORTED, Int16 FxCOUNT,
                    Char * FxBUFFER)
    SCRIPTING_TRAP(SCRIPTINGSearchFxBody);

Int16 IniVariables (UInt16 SCRIPTLIBCODE, Char * LOCALS, Int16 LOCALSIZE)
    SCRIPTING_TRAP(SCRIPTINGIniVariables);

Int16 GetVariable (UInt16 SCRIPTLIBCODE,
                   Char * varstring, Int16 varlen, Char * LOCALS,
                   Char * STACK, Char * STACKSTRING )
    SCRIPTING_TRAP(SCRIPTINGGetVariable);

Int16 SetVariable (UInt16 SCRIPTLIBCODE,
              Char * STACK, Char * STACKSTRING, Char * LOCALS )
    SCRIPTING_TRAP(SCRIPTINGSetVariable);

Int16 MakeVariable (UInt16 SCRIPTLIBCODE, Char * STACK, Char * STACKSTRING, Char *
```

```
    SCRIPTING_TRAP(SCRIPTINGMakeVariable);

Int16 ClearVariables(UInt16 SCRIPTLIBCODE, Char * LOCALS)
    SCRIPTING_TRAP(SCRIPTINGClearVariables);

Int16 Bypass (UInt16 refnum,
              Int16 iCode, Int16 modifier, Char * aux, Int16 alen,
              Char * STACK, Char * STACKSTRING)
    SCRIPTING_TRAP(SCRIPTINGBypass);

Int16 Preprocess (UInt16 refnum,
                  Char * dest, Int16 dlen,
                  Char * srcp, Int16 srclen)
    SCRIPTING_TRAP(SCRIPTINGpreprocess);

Int16 CheckVariable (UInt16 SCRIPTLIBCODE,
                     Char * LOCALS,
                     Char * STACK, Char * STACKSTRING )
    SCRIPTING_TRAP(SCRIPTINGcheckvariable);

Int16 PeekDepth(UInt16 refnum, Char * STACK)
    SCRIPTING_TRAP(SCRIPTINGpeekdepth);

Int16 GetItemStyle(UInt16 refnum, Int16 icode)
   SCRIPTING_TRAP(SCRIPTINGGetItemStyle);

Int16 PassConsoleScripting (UInt16 refnum, UInt16 cons, UInt16 errlib)
    SCRIPTING_TRAP(SCRIPTINGPassConsole);

Int16 PassBugScripting (UInt16 refnum, UInt16 bugs)
    SCRIPTING_TRAP(SCRIPTINGPassBug);

Int16 ScriptMathLib (UInt16 refnum, UInt16 MathLibCODE)
    SCRIPTING_TRAP(SCRIPTINGPassMathlib);


// a large number of error codes have been moved from here to
// ERRDEBUG.h

#define MAXCBUF 256
  // size of a condition-storing buffer used in WHILE statement processing
#define fSORT     1
#define fMAX      2
#define fMIN      4
#define fDISTINCT 8
// above are pre-processor flags

#endif
```

4 The Makefile

```
LIBPATH = c:/palmdev/sdk-4
CREATOR = JxVS
LIBPATH = c:/palmdev/sdk-4
VERSION = 1

CC = m68k-palmos-gcc -Wall -g -O2
#-I $(LIBPATH)/include \
#-I $(LIBPATH)/include/core \
#-I $(LIBPATH)/include/core/system \
#-I $(LIBPATH)/include/core/system/Unix \
#-I $(LIBPATH)/include/core/hardware \
#-I $(LIBPATH)/include/core/UI \
#-I $(LIBPATH)/include/libraries/PalmOSGlue \
#-L $(LIBPATH)/lib/m68k-palmos-coff
AS = m68k-palmos-as

all: SCRIPTING-syslib.prc

SCRIPTING-syslib.prc: SCRIPTING.def SCRIPTING
build-prc -o $@ SCRIPTING.def SCRIPTING
ls -l *.prc

SCRIPTING_objs = SCRIPTING.o SCRIPTING-dispatch.o

SCRIPTING: $(SCRIPTING_objs) Makefile
$(CC) -shared -nostartfiles -nostdlib -o $@ $(SCRIPTING_objs) -lnfm -lgcc
m68k-palmos-objdump --section-headers SCRIPTING

SCRIPTING.o: SCRIPTING.c SCRIPTING.h

SCRIPTING-dispatch.o: SCRIPTING-dispatch.s

SCRIPTING-dispatch.s: SCRIPTING.def
m68k-palmos-stubgen SCRIPTING.def

clean:
rm -f *.o *.prc *-dispatch.? SCRIPTING
```

5 The DEF file: SCRIPTING.def

```
syslib { "SCRIPTING Library" ScLi }

export {
  SCRIPTINGOpen SCRIPTINGClose nothing nothing
  ScriptIni PushItem PopItem DoScript Resolve StackPeek EncodeAll
  PixelX PixelY PixelW PixelH UsefulRead DoFloat
  SeekText SeekNumber SeekFloat PackConditions
  IniFx SearchFxBody
  IniVariables GetVariable SetVariable MakeVariable ClearVariables
  Bypass Preprocess CheckVariable PeekDepth
  GetItemStyle PassConsoleScripting PassBugScripting
  ScriptMathLib
  }
```