Analgesia Database: Numeric Library for PalmOS PDA

Version 0.95

J.M. van Schalkwyk

February 27, 2009

Contents

1 The C file: NUMERIC.c

This library contains numeric routines of general utility within our database and scripting language. Several of these were initially included in the scripting language but have now been moved here for reasons of both logic *and* space!

The library has grand aspirations, but at present is largely a stub. The big idea is ultimately:

1. To have representation of our chosen SQL data types within the scripting routines;
2. To only permit formal conversion between these types, i.e. to forbid the bane of automatic casts;
3. To make number representation IEEE754r compliant;[1]
4. To permit substitution into SQL queries (and so forth) directly from results, without intermediate conversion;
5. To have all scripting commands verify types, and only act on appropriate types.

In terms of the above, my initial implementation in Perl misled me. The weak typing of Perl (and automatic casting) may be convenient, but is fundamentally unwise, as is default use of floats, a practice which is inherently flawed.

In addition, we should be implementing calls *between* libraries, instead of our current practice of bumping things back up from scripting to the main program and then down to this numeric library.

First, the usual includes. We also add in *palmsql3A.h*, which contains constants we require in both the main program and our scripting.

```
#include <SystemMgr.h>
#include <PalmOS.h>
#include "NUMERIC.h"
#include "../palmsql3A.h"
#include "../err/ERRDEBUG.h"
```

1.1 Basic Functions

These functions include the standard opening/closing functions, `nothing` (as described in *ErrLib.tex*), and several small utility functions, mostly imported from the main program for local use.

[1]But see what we do in returning errors from Asc2Int32!

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

Err NUMERICOpen (UInt16 refnum) {
  return 0;
  }

Err NUMERICClose (UInt16 refnum, UInt16 *numappsP) {
  return 0;
  }

Err nothing (UInt16 refnum) {
  return 0;
  }
```

If we wanted globals, we could probably do this, using the GaussLib example as a prototype. But we don't. Let's look at the xCopy function, similar to that in the `wraps` class of the main program:

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

The w_DmWrite routine also corresponds to the one in 'wraps'.

```
Int16 w_DmWrite (void *recordP, UInt32 offset, const void *srcP,
                 UInt32 bytes, UInt32 max)
{ Int16 ok;
  if (recordP == 0)
   { return 0;
   };
  if (bytes == 0)
   { return 1; // ok!
   };
  if (srcP == 0)
   { return 0;
   };
  if (bytes + offset > max)
    { return 0; // no space
    };
  ok = DmWriteCheck(recordP, offset, bytes);
  if (ok != errNone)
    { return 0;
    };
  ok = DmWrite (recordP, offset, srcP, bytes);
  return (! ok);
}
```

The use of DmWriteCheck is clumsy, and will likely slow things down significantly, the cost of error anticipation!

1.1.1 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'. This function is a general-purpose 'fix-it' function stolen from *ScriptingLib.tex*. There is *no* check that the top is valid. At present we don't use this routine, so have disabled it:

```
Int16 PushNull(Char * STACK)
{ Int16 top;
  Char c;
  top    = *((Int16 *)(STACK+oTOP));
  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;
}
```

1.1.2 Write NULL to stack

WriteNull is similar to PushNull, but the item at the top of the stack is overwritten. There is no push. Lifted from the same source as PushNull but modified to 'just fix it'. Inelegant.

```
Int16 WriteNull(Char * STACK)
{ Int16 bottom;
  Int16 top;
  Char c;
  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top < bottom)
     { top = bottom+XVI; // try to fix
       w_DmWrite(STACK, oTOP, &top, 2, SMAX);
       top -= XVI;
     };
  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;
}
```

1.2 Simple ‘numeric-related’ routines

CountItems counts the number of strings contained within the submitted text (non-ASCIIZ). Each string ends with the character `c`. In order to count the last string it *must* have the same character terminator as the rest. We return the count. The submitted value `max` contains the length of the containing string.

```
Int16 CountItems (UInt16 refnum, Char * myptr, Int16 max, Char c)
{ Int16 cnt = 0;
  while (max > 0)
    { if (* myptr++ == c)
          { cnt ++;
          };
      max --;
    };
  return cnt;
}
```

We’ve avoided the ugly WChar usage, at the expense of internationalisation. Next, the hideous read of a 32 bit integer. [Rewrite me!]. On the Palm, sixteen and thirty two bit reads must be aligned on integral boundaries. Ugly but true. If we have no guarantee of this condition, we must painstakingly pull out each byte. Even MORE painful is the automated translation of characters when we cast them to eg. Int32.

CAUTION: the following is big-endian, and processor-specific, if the dword was written by the processor and not byte-by-byte.

```
Int32 ReadInt32X (UInt16 refnum, Char * myptr)
{
  Int32 i;
  i =   0xFF & ((Int32) *(myptr+3));
  i |= (0xFF & ((Int32) *(myptr+2)))<<8;
  i |= (0xFF & ((Int32) *(myptr+1)))<<16;
  i |= (0xFF & ((Int32) *(myptr+0)))<<24;
  return i;
}
```

Here's the equally ugly write. It just so happens that at present we always write 32 bit integers to 32 bit integral boundaries, but we define things more generally.

CAUTION: This too is specific for big-endian processors, and ugly to boot!

```
Int16 WriteInt32X (UInt16 refnum, Char * myptr, Int32 datum)
{ if (! myptr)
     { return 0; // null ptr!
     };
  * (myptr+3) = (Char) datum & 0xFF;
  datum = datum >> 8;
  * (myptr+2) = (Char) datum & 0xFF;
  datum = datum >> 8;
  * (myptr+1) = (Char) datum & 0xFF;
  datum = datum >> 8;
  * (myptr) = (Char) datum;
  return 1;
}
```

Similar is the 16-bit read.

CAUTION: the following is big-endian, and processor-specific, if the word was written by the processor and not byte-by-byte. Bugger, it's clumsy.

```
Int16 ReadInt16X (UInt16 refnum, Char * mP)
{
  return(  (*(mP+1) & 0xFF ) + ((*(mP))<<8 )   );
}
```

Aah. Just use inline assember? this is what PalmOS has on the topic:

"The asm keyword is used to pass information through the compiler to the assembler. The Palm OS compiler permits assembler code to be inlined using the keywords asm, _asm, and __asm. The asm keyword has its normal C99 and C++ behavior; in addition, when used as the first keyword in a function definition, the contents of the function are all taken as assembly instructions and the function is emitted "naked," without a prologue or epilogue that pushes or pops registers from the stack. (A 'bx lr' return instruction is placed after your code, in case you do not explicitly return.) An asm function is called in the same way as any function; its arguments are in registers r0-r3 and on the stack, as is defined by ATPCS:

```
    asm int func (int a, int b) {
        add r0, r0, r1 // return a+b
    }
```

The “inline” qualifier can be used with asm functions to indicate that the body of the asm function should be inserted at each call-site. (The asm function should not explicitly return or use labels. As in the above example, it should fall off the end to return execution to the caller.) Supported use of asm routines is limited to ”nop,” as an inline asm statement and relatively small asm functions that do not use labels.

`__asm` Followed by curly brackets, indicates a multi-line inline assembly block. Otherwise, indicates inline assembly until the end of the current line.

Next, the sixteen bit write. Regardless of the processor, we write a big-endian integer to memory.

```
Int16 WriteInt16X (UInt16 refnum, Char * myptr, Int16 datum)
{ if (! myptr)
     { return 0; // null ptr
     };
  * myptr = datum / 256;
  * (myptr+1) = datum % 256; // clumsy, robust
  return 1;
}
```

Our next routine reads ASCII, converting to Int32. This clumsy routine formerly limited us to values from zero to a billion minus 1, but now we permit all but -0x80000000, which we use as an error signal!

```
Int32 Asc2Int32 (UInt16 refnum, Char * myptr, Int16 slen)
{ Int16 posi = 1; // 0 signals -ve
  Int32 i = 0;
  Int16 c;

 if (slen <= 0)
     { return -0x80000000; // error
     };
 if (* myptr == ’-’)
    { myptr ++;
      slen --;
      posi = 0;
    };
  if (slen > 9)
     { // was: ERRmsg(ErLongAscInt);
       return -0x80000000;
     };
   while (slen > 0)
     { c = (Int16) * myptr;
       myptr ++;
       if ( (c > 0x39) || (c < 0x30) )
```

```
          { return -0x80000000;
          };
       c -= 0x30;
       if (i >= 100000000)
          { return -0x80000000;
          };
       i *= 10;
       i += c;
       slen --;
     };
   if (posi)
      { return i;
      };
  return (-i);
}
```

1.3 Basic arithmetic

For DoAdd and the routines which follow, our rules are quite rigorous. Only numbers of the *same type* can be operated upon. For fixed-point numerics:

- Scale of sum is biggest scale, likewise for subtraction
- Scale of product is sum of scales of numbers
- Scale of a/b is the biggest scale of a;b

It seems a good idea to use DECNUMBER-324. Perhaps turn this into a palmos library. 64-bit encoding seems adequate for our purposes.

For floating point: just do it!

We will first examine the division routine, if only because this was the first one we revised in July 2006.

1.3.1 Division

We divided the deeper stack value by the value on the *top* of the stack, and replace the former with the result.

There is one thing to consider — what do we do when an error occurs. The most important thing here is not to mess up the stack (which we have been doing up till now) as this generates *more serious* errors. The logical answer is, as we expect *something* on the stack, to write NULL there (or NaN; we will regard the two as equivalent).

```
Int16  DoDiv (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Int32 i;
  Int32 i2;
  double d;
  double d2;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
```

We move the top one down, which will accommodate the answer. However, if there are insufficient data on the stack, we will nevertheless write a NULL to the stack, to preserve stack integrity (but must still alert about the error)!

```
  if (top <= bottom)
     { WriteNull(STACK); //  write null, fail
       return -ScErDivEmpty; // WriteNull tries a fix.
       // "Divide err: no args"
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);

  t = *(STACK+top+15);
  switch (t)
    {

      case ’I’:
        if (*(STACK+top-XVI+15) != ’I’)
           { WriteNull(STACK); //  write null
             return -ScErDivArgs1;
             // "Divide err: bad integer dividend"
           };
        i  = *((Int32 *)(STACK+top-XVI)); // ib4?
        i2 = *((Int32 *)(STACK+top));     // ib4?
        if (i2 == 0)
           { WriteNull(STACK); //  write null
             return -ScErDivOver; // forestall!
             // "Divide err: divisor zero"
           };
        i /= i2;
        w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
        return 1;

      case ’F’:
        if (*(STACK+top-XVI+15) != ’F’)
           { WriteNull(STACK); //  write null
             return -ScErDivArgs2;
             // "Divide err: bad float dividend"
           };
        xCopy ((Char *)&d, STACK+top, 8);
        xCopy ((Char *)&d2, STACK+top-XVI, 8);
        d2 /= d;
        w_DmWrite(STACK, top-XVI, &d2, 8, SMAX);
        return 1;
        /* NOTE: what about trapping NaN, infinities etc */
        /* could do so here, or later when output */
        /* the above applies to all floating point */

      case 'N':
        if (*(STACK+top-XVI+15) != 'N')
           { WriteNull(STACK); //  write null
             return -ScErDivArgs3;
             // "Divide err: unsupported numeric"
           };
           /* WILL INCLUDE APPROPRIATE ROUTINE HERE */
      break;

     // default:
     //   WriteNull(STACK); // write null
     //   return -ScErDivArgs4;
    };

  // default is actually this!
  WriteNull(STACK); //  write null
  return -ScErDivArgs4;
  // "Divide err: bad numeric arg(s)"
}
```

1.3.2 Modulo

Modulo is nearly identical (Should we rename this Remainder?). Look at IEEE-854, comparing remainder and remainder-near. For now, only integer version.

```
Int16  DoMod (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Int32 i;
  Int32 i2;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     {
       WriteNull(STACK);
       return -ScErModEmpty;
       // "Modulo err: no args"
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);

  t = *(STACK+top+15);
  switch (t)
    { case ’I’:
        if (*(STACK+top-XVI+15) != ’I’)
           {
             WriteNull(STACK);
             return -ScErModArgs1;
             // "Modulo err: no args"
           };
        i  = *((Int32 *)(STACK+top-XVI));
        i2 = *((Int32 *)(STACK+top));
        if (i2 == 0)
           {
             WriteNull(STACK);
             return -ScErModOver;  // div by 0??
             // "Modulo err: divide by 0"
           };
        i %= i2;  // find mod of deeper vs top
        w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
        return 1;

      case ’F’:
      break;

      case ’N’:
        /* here will eventually insert/invoke a remainder rtn */
      break;
      // default:
      //  WriteNull(STACK);
      //  return -ScErModArgs2;
    };
  WriteNull(STACK);
  return -ScErModArgs2;
  // "Modulo err: unsupported args"
}
```

1.3.3 Addition

```
Int16  DoAdd (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Int32 i;

  double d;
  double d2;

  // standard pop:
```

```
bottom = *((Int16 *)(STACK+oSTART));
top    = *((Int16 *)(STACK+oTOP));
top -= XVI;
if (top <= bottom)
   {
     WriteNull(STACK);
     return -ScErAddEmpty; // underflow
           // "Addition err: no args"
   };
w_DmWrite(STACK, oTOP, &top, 2, SMAX); // pop 1 item

t = *(STACK+top+15); // top No.
switch (t)
  { case 'I':
      if (*(STACK+top-XVI+15) != 'I')
         {
           WriteNull(STACK);
           return -ScErAddArgs1;
           // "Addition err: bad integer"
         };
      i = *((Int32 *)(STACK+top));
      i += *((Int32 *)(STACK+top-XVI));
      if (i > 999999999)
         {
           WriteNull(STACK);
           return -ScErAddIOver;
           // "Addition err: our overflow"

           // what about underflow eg -nn + (-nn)
         };
      w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
      return 1;

    case 'F':
      if (*(STACK+top-XVI+15) != 'F')
         {
           WriteNull(STACK);
           return -ScErAddArgs2;
           // "Addition err: bad float"
         };
      xCopy ((Char *)&d, STACK+top, 8);
      xCopy ((Char *)&d2, STACK+top-XVI, 8);
      d += d2;
      w_DmWrite(STACK, top-XVI, &d, 8, SMAX);
      return 1;

    case 'N':
      if (*(STACK+top-XVI+15) != 'N')
```

```
           {
             //WriteNull(STACK);
             //return -ScErAddArgs;
           };
           /* WILL INCLUDE APPROPRIATE ROUTINE HERE */
      break;

      // default:
      //       WriteNull(STACK);
      //  return -ScErAddArgs;
    };
  WriteNull(STACK);
  return -ScErAddArgs3;  // never taken
             // "Addition err: unsupported type"
}
```

1.3.4 Subtraction

We subtract the top stack value *from* the one below, and replace the latter with the result.

```
Int16  DoSub (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Int32 i;
  double d;
  double d2;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     {
       WriteNull(STACK);
       return -ScErSubEmpty;
             // "Subtract err: no args"
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);

  t = *(STACK+top+15);
  switch (t)
    { case ’I’:
        if (*(STACK+top-XVI+15) != ’I’)
           {
             WriteNull(STACK);
             return -ScErSubArgs1;
             // "Subtract err: bad integer"
           };
        i = *((Int32 *)(STACK+top-XVI)); // deeper
        i -= *((Int32 *)(STACK+top));    // sub upper value
        if (i < 0)
           {
             WriteNull(STACK);
             return -ScErSubIUnder;  // [NO! fix me!]
             // "Subtract err: underflow"

             // WHAT ABOUT OVERFLOW ?????????????? (7ff..f - (-7ff..f))
           };
        w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
        return 1;

      case ’F’:
        if (*(STACK+top-XVI+15) != ’F’)
           {
             WriteNull(STACK);
             return -ScErSubArgs2;
             // "Subtract err: bad float"
           };
        xCopy ((Char *)&d, STACK+top, 8);
        xCopy ((Char *)&d2, STACK+top-XVI, 8);
        d2 -= d;
        w_DmWrite(STACK, top-XVI, &d2, 8, SMAX);
        return 1;

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

     // default:
     //   WriteNull(STACK);
     //   return -ScErSubArgs;
    };
  WriteNull(STACK);
  return -ScErSubArgs3;
             // "Subtract err: unsupported type"
}
```

1.3.5 Multiplication

Multiply is again similar; the clumsy `bitlength` operation checks the number of significant bits in a long integer. (For now, we’re only interested in positive integers, but we need to fix this quirk).

```
Int16 bitlength (Int32 i)
{
  Int16 l=0;
  while (i > 0)
    { i >>= 1;                 /* shift bits 1 right */
      l ++;
    };
  return l;
}

Int16  DoMul (Char * STACK, Char *  STACKSTRING)
{
  Int16 bottom;
  Int16 top;
  Char t;
  Int32 i;
  Int32 i2;

  double d;
  double d2;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  top -= XVI;
  if (top <= bottom)
     {
       WriteNull(STACK);
       return -ScErMulEmpty;
   // "Multiply err: no args"
     };
  w_DmWrite(STACK, oTOP, &top, 2, SMAX);

  t = *(STACK+top+15); // type?
  switch (t)
    { case 'I':
        if (*(STACK+top-XVI+15) != 'I')
           {
             WriteNull(STACK);
             return -ScErMulArgs1;
             // "Multiply err: bad integer arg"
           };
        i  = *((Int32 *)(STACK+top-XVI));
        i2 = *((Int32 *)(STACK+top));
        if (bitlength(i) + bitlength(i2) > 30) // ugly
           {
             WriteNull(STACK); // should actually be infinity?? [fix me]
             return -ScErMulOver1;
             // "Multiply err: overflow"
```

```
            };
         i *= i2;
         if (i > 999999999)
            {
              WriteNull(STACK);
              return -ScErMulOver2;
              // "Multiply err: our overflow"
            };
         w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
         return 1;

       case ’F’:
         if (*(STACK+top-XVI+15) != ’F’)
            {
              WriteNull(STACK);
              return -ScErMulArgs2;
              // "Multiply err: bad float"
            };
         xCopy ((Char *)&d, STACK+top, 8);
         xCopy ((Char *)&d2, STACK+top-XVI, 8);
         d2 *= d;
         w_DmWrite(STACK, top-XVI, &d2, 8, SMAX);
         return 1;

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

       //default:
       //  WriteNull(STACK);
       //  return -ScErMulArgs;
     };
   WriteNull(STACK);
   return -ScErMulArgs3;
              // "Multiply err: unsupported type"
}
```

The similarities are becoming boring.

1.3.6 Negation

Finally, negation.

```
 Int16  DoNegate (Char * STACK, Char *  STACKSTRING) {
  Int16 bottom;
  Int16 top;
  Char t;
  Int32 i;
  double d;

  bottom = *((Int16 *)(STACK+oSTART));
  top    = *((Int16 *)(STACK+oTOP));
  if (top <= bottom)
     {
       WriteNull(STACK);
       return -ScErNegEmpty;
             // "Negation err: no args"
     };
  top -= XVI;

  t = *(STACK+top+15);
  switch (t)
    { case 'I':
        i = *((Int32 *)(STACK+top));
        i = -i;  // temporary debugging! 29-1-2006
        w_DmWrite(STACK, top-XVI, &i, 4, SMAX);
      return 1;

      case 'F':
        xCopy ((Char *)&d, STACK+top, 8);
        d = -d;
        /* smarter would be simply to toggle the sign bit ?! */
        w_DmWrite(STACK, top, &d, 8, SMAX);
        return 1;

      case 'N':
            // simply fail..
           /* WILL INCLUDE APPROPRIATE ROUTINE HERE */
      break;

     // default:
     //   return -ScErNegArgs;
     //
    };
  WriteNull(STACK);
  return -ScErNegArgs;
   // "Negation err: bad arguments"
}
```

1.4 Our principal despatching function

The following ugly routine simply routes a call to the relevant subroutine. We assume the existence of a valid stack and stackstring, and don't check these. Although at present we don't actually *use* stackstring, we need to have the capability for longer numerics in future!

```
Int16 DoNumeric (UInt16 refnum, Int16 mycode, Char * STACK,
                 Char * STACKSTRING)
{
  switch (mycode)
    {
       case iA_SUB:
         return DoSub(STACK, STACKSTRING);

       case iA_ADD:
         return DoAdd(STACK, STACKSTRING);

       case iA_MUL:
         return DoMul(STACK, STACKSTRING);

       case iA_DIV:
         return DoDiv(STACK, STACKSTRING);

       case iA_MOD:
         return DoMod(STACK, STACKSTRING);

       case iA_NEG:
         return DoNegate(STACK, STACKSTRING);

     //  default:
     //    return 0;  // fail.
    };
  return 0; // otherwise fail
}
```

2 Header file: NUMERIC.h

In the header file we first include relevant header files, use the same ‘trap trickery’ we used in *ErrLib.tex*, and then provide function headers.

```
#ifndef NUMERIC_H
#define NUMERIC_H

#include <LibTraps.h>
#include <FloatMgr.h>

#ifndef NUMERIC_TRAP
#define NUMERIC_TRAP(trapno)  SYS_TRAP(trapno)
#endif

#define NUMERICDoNumeric       sysLibTrapCustom+0
#define NUMERICCountItems      sysLibTrapCustom+1
#define NUMERICReadInt32X      sysLibTrapCustom+2
#define NUMERICWriteInt32X     sysLibTrapCustom+3
#define NUMERICReadInt16X      sysLibTrapCustom+4
#define NUMERICWriteInt16X     sysLibTrapCustom+5
#define NUMERICAsc2Int32       sysLibTrapCustom+6

Err NUMERICOpen (UInt16 refNum)
    NUMERIC_TRAP(sysLibTrapOpen);

Err NUMERICClose (UInt16 refNum, UInt16 *numappsP)
    NUMERIC_TRAP(sysLibTrapClose);

Int16 DoNumeric (UInt16 refnum, Int16 mycode, Char * STACK,
                 Char * STACKSTRING)
    NUMERIC_TRAP(NUMERICDoNumeric);

Int16 CountItems (UInt16 refnum, Char * myptr, Int16 max, Char c)
    NUMERIC_TRAP(NUMERICCountItems);

Int32 ReadInt32X (UInt16 refnum, Char * myptr)
    NUMERIC_TRAP(NUMERICReadInt32X);

Int16 WriteInt32X (UInt16 refnum, Char * myptr, Int32 datum)
    NUMERIC_TRAP(NUMERICWriteInt32X);

Int16 ReadInt16X (UInt16 refnum, Char * myptr)
    NUMERIC_TRAP(NUMERICReadInt16X);

Int16 WriteInt16X (UInt16 refnum, Char * myptr, Int16 datum)
    NUMERIC_TRAP(NUMERICWriteInt16X);

Int32 Asc2Int32 (UInt16 refnum, Char * myptr, Int16 slen)
```

```
    NUMERIC_TRAP(NUMERICAsc2Int32);
```

We had a small selection of constants here, but we’ve moved them to *ErrLib.tex*.

```
#define ScErSubArgsN                238
// temporary
#endif
```

3 The Makefile

Our makefile is similar to the simple makefile we used in *ErrLib.tex*. The catch (and this is a big catch) is the hurdles we have to overcome to use floating point routines within a library! The key modifiers are

```
  -lnfm -lgcc
```

Take note that these have to be specified in *this order* or the whole thing blows, leaving non-empty data segments and crashing the PDA rather horribly. Oh, and don't forget `-nostdlib` either!

```
LIBPATH = c:/palmdev/sdk-4
CREATOR = JxVS
LIBPATH = c:/palmdev/sdk-4
VERSION = 1

CC = m68k-palmos-gcc -Wall -g -O2  -mdebug-labels
AS = m68k-palmos-as

all: NUMERIC-syslib.prc

NUMERIC-syslib.prc: NUMERIC.def NUMERIC
build-prc -o $@ NUMERIC.def NUMERIC
ls -l *.prc

NUMERIC_objs = NUMERIC.o NUMERIC-dispatch.o

NUMERIC: $(NUMERIC_objs) Makefile
$(CC) -shared -nostartfiles -nostdlib -o $@ $(NUMERIC_objs) -lnfm -lgcc
m68k-palmos-objdump --section-headers NUMERIC

NUMERIC.o: NUMERIC.c NUMERIC.h

NUMERIC-dispatch.o: NUMERIC-dispatch.s

NUMERIC-dispatch.s: NUMERIC.def
m68k-palmos-stubgen NUMERIC.def

clean:
rm -f *.o *.prc *-dispatch.? NUMERIC
```

Okay, the above could do with a bit of a spring clean.

4 The DEF file: NUMERIC.def

Nothing unusual here, just a listing of routines in order.

```
syslib { "NUMERIC Library" NuLi }

export {
  NUMERICOpen NUMERICClose nothing nothing DoNumeric
    CountItems ReadInt32X WriteInt32X ReadInt16X WriteInt16X
    Asc2Int32
  }
```

5 Change Log

From version 0.95, we introduce a change log.

5.1 Version 0.95