Alto/Mesa Floating Point Packages

October 11, 1979

Read this carefully! There is a lot of information hidden away here.

FLOAT, FLOATIO, FLOATFNS, and FLOATDEBUG contain procedures that implement various operations on REALs. FLOAT must be STARTed (see *InitFloat* below); it stuffs the appropriate locations in the system dispatch vector so that REAL operations will be defined, and must be present for all the other packages. The necessary **TYPE** and **PROCEDURE** declarations appear in RealDefs and FloatFnDefs and are described below. Three error conditions cause the signal **FloatingPointError** of type **FloatingError**. These are: division by 0, exponent overflow (x NOT IN[$-2^{127}..2^{127}$], and exponent overflow from Fix (floating point value is outside the range of the destination type). RESUMEing from this signal will return from the floating point operation with a 'reasonable' value.

There are two versions of these packages. One has microcode implementations of the floating point operations and can only be used on Alto II's with Mesa ROMs or 3K CRAM. The other version has BCPLCode operations and will run on any Alto. *There are also two versions of RealDefs*, one of which has INLINE microcode calls for Fix, FixI, and FixC, and the other of which does not. The INLINE version is compatible with the microcode floating point.

Fine point about the signal FloatingPointError: Mesa 5.0 implements the ENABLE clause by placing a catch phrase on every procedure call within the enabled block. Because of this, an ENABLE FloatingPointError on a block containing floating point operations will not catch signals generated by those operations, but it will catch signals generated by floating point operations executed within procedures called from inside the block. This nuance causes BCPL-Float and uCode-Float to behave in slightly different ways -- the BCPL-Float version of Fix is a procedure call, so an ENABLE clause on a block containg Fix will work, but the Fix in uCode-Float is an Inline call to microcode, so the ENABLE will not work.

Fine point about FloatDebug: The only version of FloatDebug uses the non-microcode floating point operations. This is because the correct microcode Ram image could conceivably not be loaded when the debugger swaps in. Do not use FloatDebug anywhere except with the debugger, equivalent procedures are available in FloatIO. The file MakeFloatDebug.cm in FloatDebug.dm is only a *suggested* command file for installing the debugger, XMesa users may wish to add XCoreMap.

The file BCPL-Float.dm contains everything needed for the non-microcode version; uCode-Float.dm contains everything needed for the microcode version. FloatDebug.dm contains the modules needed for Debugger access to REALs. All sources are in FloatSources.dm. If you use floating point at all, you must include Float.bcd in your configuration. If you read and print floats, you must include FloatIO.bcd. If you use any of the procedures in FloatFnDefs, you must include FloatFns.bcd. *These are the only required modules.*

in FLOAT: (exports RealDefs)

FloatingError: TYPE = {noError,FixRangeOverflow,ExponentOverflow,DivideBy0};

FloatingPointError: SIGNAL[f: FloatingError];

InitFloat: PROCEDURE;

START ing Float or calling InitFloat sets up SD entries used by the floating point package. One of these must be done before doing any floating point operations. (InitFloat is called from the mainline code of Float, so it can be initialized by loading on the command line.) Fine point: since Float: PROGRAM is no longer exported, the only ways to initialize the floating point package are to call InitFloat or to load the module Float with the command line.

Fix: PROCEDURE [REAL] RETURNS [LONG INTEGER];

Fix returns the integer part of the REAL number. If the integer will not fit into 32 bits, an error signal is generated.

FixI: PROCEDURE [REAL] RETURNS [INTEGER];

FixC: PROCEDURE [REAL] RETURNS [CARDINAL];

These versions of fix return the integer part of the REAL number as INTEGER or CARDINAL. If the integer will not fit into the type, an error signal is generated.

in FloatIO: (exports RealDefs)

WriteFloat: PROCEDURE [REAL];

AppendFloat: PROCEDURE [STRING, REAL];

WriteFloat writes the REAL number onto the default output stream (using IODefs.WriteChar); AppendFloat adds onto STRING. Scientific notation is used whenever the number is $<10^{-4}$ or $>10^{7}$.

WFWriteFloat: PROCEDURE [rp: UNSPECIFIED, f: STRING, p: PROCEDURE [CHARACTER]];

WFWriteEFloat: PROCEDURE [rp: UNSPECIFIED, f: STRING, p: PROCEDURE [CHARACTER]];

These procedures are intended for use with the WriteFormatted package via the SetCode procedure. The UNSPECIFIED is treated as a POINTER TO REAL. The PROCEDURE [CHARACTER] is used to print the output. The STRING is a format control string of the form "[lf][.[rf]]" where brackets indicate options. Lf is the total allowed field width, rf is the number of fractional digits to print, and the minus sign indicates left justification. If either lf or rf is 0, as many digits as are appropriate will be printed. WFWriteEFloat forces the output to be in scientific notation, otherwise scientific notation is used whenever the number is $<10^{-4}$ or $>10^{7}$.

ReadFloat: PROCEDURE RETURNS [REAL];

Reads a REAL number from the default input stream (using IODefs.ReadID). The input may be in scientific notation.

StringToFloat: PROCEDURE [s: STRING] RETURNS [REAL];

Like ReadFloat, but termination is also caused by reaching the end of the string.

in FloatFns: (exports FloatFnDefs)

Exp: PROCEDURE [REAL] RETURNS [REAL];

For an input argument n, returns e^n (e=2.718...). (computed by continued fractions).

Log: PROCEDURE [base,arg: REAL] RETURNS [REAL];

Computes logarithm to the base **base** of **arg** (by Ln(arg)/Ln(base)).

Ln: PROCEDURE [REAL] RETURNS [REAL];

Computes the natural logarithm (base e) of the input argument.

SqRt: PROCEDURE [REAL] RETURNS [REAL];

Calculates the square root of the input value by Newton's iteration.

Root: PROCEDURE [index,arg: REAL] RETURNS [REAL];

Calculates the **index**th root of **arg** by $e^{(Ln(arg)/index)}$.

Power: PROCEDURE [base,exponent: REAL] RETURNS [REAL];

Calculates **base** to the **exponent** power by $e^{(exponent*Ln(base))}$.

Sin: **PROCEDURE [radians: REAL] RETURNS [REAL];**

SinDeg: **PROCEDURE [degrees: REAL] RETURNS [REAL];**

Cos: **PROCEDURE [radians: REAL] RETURNS [REAL];**

CosDeg: **PROCEDURE [degrees: REAL] RETURNS [REAL];**

Tan: **PROCEDURE [radians: REAL] RETURNS [REAL];**

TanDeg: **PROCEDURE [degrees: REAL] RETURNS [REAL];**

Computes the trigonometric function by polynomial; good to 7.33 decimal places.

ArcTan: **PROCEDURE [y, x: REAL] RETURNS [radians: REAL];**

ArcTanDeg: **PROCEDURE [y, x: REAL] RETURNS [degrees: REAL];**

Good to 8.7 decimal places.

in FloatDebug: (exports DebugRealDefs; when loaded with your debugger, the debugger will print REALs numerically). These are the same procedures as AppendFloat *and* StringToFloat *in FloatIO.*

AppendRealNumber: PROCEDURE [s: STRING,R: REAL];

Converts R to a STRING, and appends this STRING to the input STRING s.

StringToRealNumber: PROCEDURE [s: STRING] RETURNS [REAL];

Same as StringToFloat (in Floatio).

Floating Point Format

A Mesa REAL is a 32 bit object consisting of a sign bit, 8 bit binary exponent, and 23 bit mantissa. The sign is encoded as two's complement, so that a double word integer compare can be used for floating compare. The exponent is encoded as excess-128. The mantissa is always normalized and the leading "1" is not actually stored; a "hidden-bit". The word containing the sign bit is stored at a higher memory address than the word containing the low-order mantissa bits.