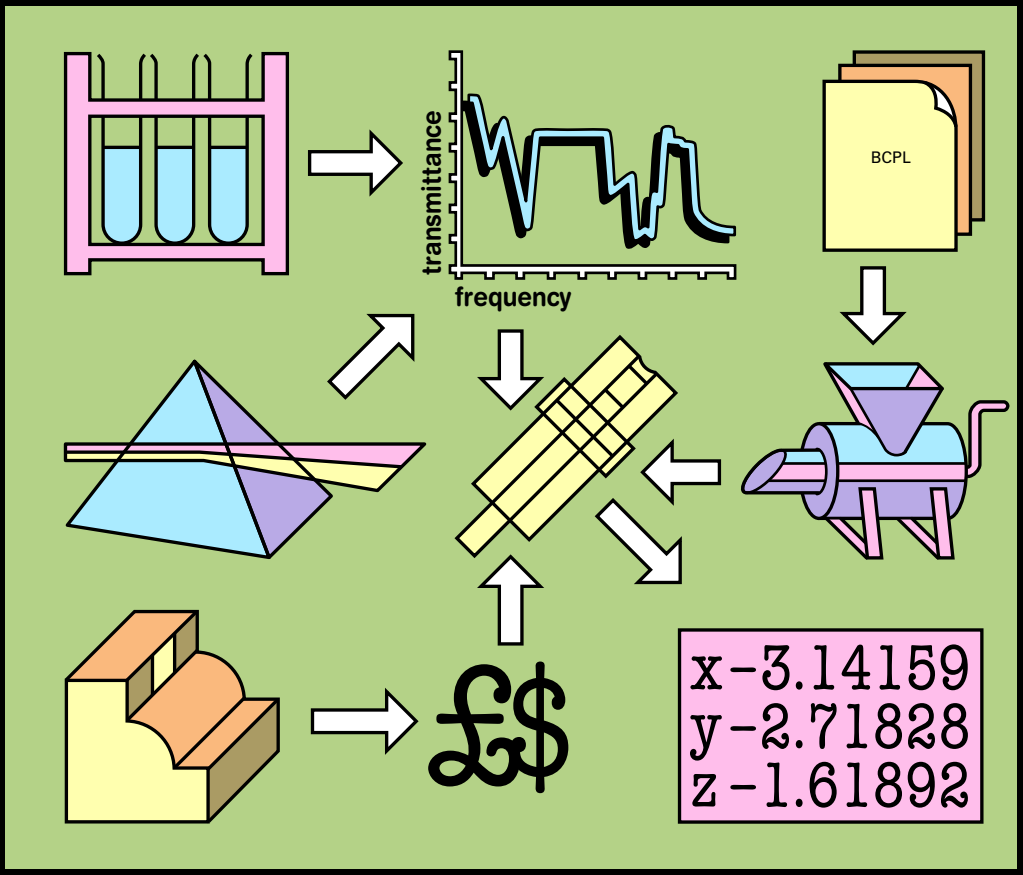
BCPL
Calculations Package
on the BBC Microcomputer

User Guide

CHRIS JOBSON



User Guide

BCPL
Calculations Package
on the BBC Microcomputer

CHRIS JOBSON

ACORNSOFT

Disclaimer

Richards Computer Products Ltd. and Acornsoft Ltd. reserve the right to make improvements in the product described in this book at any time and without notice. Every effort has been made to ensure the accuracy of the material presented in this book. Nevertheless, experience shows that some textual and software errors always remain to be discovered. If you find any errors, or have any suggestions on how to improve this book, please contact Richards Computer Products Limited, Brookside, Westbrook Street, Blewbury, Didcot, Oxfordshire OX11 9QA, England.

First published in 1983, by Acornsoft Limited

FIRST EDITION

Re-mastered by dv8 in 2020
First revision, November 2020

For the latest revision of this book go to:
stardot.org.uk/forums/viewtopic.php?f=42&t=21166

Published by:

Acornsoft Limited
4a Market Hill
Cambridge
CB2 3NJ
England

Contents

1 Introduction

The BCPL Calculations Package is designed for use with the BCPL CINTCODE system on the BBC Microcomputer. This guide assumes that the reader is familiar with the BCPL CINTCODE system.

The package provides three different sets of calculation procedures which together meet the requirements of a wide range of applications. The three types of calculation supported are floating point, fixed point and fast integer.

Floating point

The floating point procedures are likely to be most useful in scientific, mathematical and engineering applications, where the ability to handle both very large and very small numbers is important.

Floating point calculations can be performed on numbers which contain decimal points and which may be too large or too small to be held in a single word. The internal format used conforms to the IEEE standard (using a 40-bit mantissa and a seven-bit exponent). It allows a large range of numbers to be represented but limits the precision of each number to approximately twelve decimal digits.

The smallest non-zero positive number that can be represented is approximately 2.17*10^(-19). The largest positive number that can be represented is approximately 1.84*10^19.

It is not possible to hold a floating point number with the required accuracy in a single 16-bit word. Instead numbers are held in three-word vectors. Procedures are provided to convert between 16-bit integers and floating point numbers and to read and write floating point numbers.

All calculations involving floating point numbers are performed by calling procedures. The procedures supplied support the basic arithmetic operations, square root and various trigonometrical and exponential functions.

Fixed point

Fixed point calculations are likely to be most useful in commercial and financial applications where complete accuracy is important. For example, calculations on amounts of up to almost ten thousand million pounds can be performed which are accurate to one penny (or even halfpenny).

The fixed point procedures allow exact calculations to be performed on numbers with up to ten digits before the decimal point and up to four digits after the decimal point. Thus they do not support such a wide range of numbers as the floating point procedures, but they provide better accuracy within the more limited range.

As with floating point, it is not possible to hold a fixed point number with the required accuracy in a single 16-bit word. Instead numbers are held in four-word vectors. Procedures are provided to convert between 16-bit integers and fixed point numbers and to read and write fixed point numbers.

All calculations involving fixed point numbers are performed by calling procedures. The procedures supplied support the basic arithmetic operations, square root, natural logarithm and exponentiation. The procedures used for these fixed point operations have the same names and interfaces as the corresponding floating point procedures.

Fast integer

The fast integer procedures are primarily intended for use in graphics programs, where their speed allows rapid updating of the screen. The accuracy they give is sufficient for use in plotting curves in all display modes. They may also be useful in scientific, mathematical and engineering applications which do not require great accuracy.

The procedures allow quick calculation of certain mathematical functions to a limited accuracy using ordinary 16-bit integers. The functions provided are square root, sine, cosine and arcsine (which finds the angle whose sine is a given value). The procedures are all considerably faster than their floating point equivalents.

The square root procedure gives the square root of the product of two 16-bit numbers. The other procedures work with numbers scaled by a factor of 10000. Thus calling the sine procedure with a parameter of 5000 calculates the sine of 0.5 radians (which is approximately 0.4794) and returns the answer multiplied by 10000 (i.e. 4794).

Use of more than one number format

Floating point and fixed point procedures performing the same function are given the same name and the same global number and have similar parameters. Thus it is simple to write a program that will work with both floating point and fixed point numbers. It is not, however, possible to use both number formats within one program unless the program itself links and unlinks the appropriate procedures at appropriate times. Note that the internal formats of floating point and fixed point numbers are quite different.

The fast integer procedures are completely independent of the floating point and fixed point procedures and may be used by a program whether or not it also uses floating point or fixed point.

HOW TO USE THIS GUIDE

The three parts of the package are described individually in chapters 2, 3 and 4. Chapter 2 covers floating point, chapter 3 covers fixed point and chapter 4 covers fast integer. Chapter 5 contains information on linking the specific procedures needed by a particular program. Chapter 5 also lists the files supplied as part of the package.

Experienced users who require a quick reference to the facilities of the package may refer to the summaries in chapter 6.

Users requiring details of the internal formats of floating point and fixed point numbers should refer to the Appendix.

Users needing to calculate mathematical functions not directly supported (e.g. arccosine using the fast integer procedures) may find a suitable formula in the Appendix.

2 Floating point

This chapter contains all the information needed by a user of the floating point facilities of the BCPL Calculations Package except for the details of linking in the procedures required (which are described in chapter 5).

It contains a description of various features of the floating point facilities followed by one section for each of the floating point procedures.

Standardisation

This implementation of floating point is a superset of BCPL Language Extension A13b defined in 'A Proposed Definition of the Language BCPL' published in October 1979.

Programmers wishing to write programs which are compatible with other BCPL systems supporting this language extension should not make use of the internal format of floating point numbers and should use only the following procedures:

FABS, FCOMP, FDIV, FFIX, FFLOAT, FMINUS, FMULT, FNEG, FPLUS, READFP and WRITEFP.

In some implementations FFIX may return the integer part of a floating point number (like FINT in this implementation) rather than the nearest integer to the number.

FLOATING POINT NUMBERS

Floating point numbers are held in three-word vectors. Each number is stored as a binary fraction and a binary exponent (the Appendix contains details of the exact format) so that the value of the number is given by (fraction)*(two to the power exponent).

This method of storing numbers allows both very large and very small numbers to be represented, but as the numbers represented become larger the accuracy (in absolute terms) with which they can be represented decreases (although the accuracy relative to the magnitude of the number is approximately constant). This is because the number of distinct values that can be represented between any two successive powers of two is the same. For example there are exactly the same number of distinct floating point values between 4096 and 8192 as there are between 0.25 and 0.5 and thus real numbers between 0.25 and 0.5 can be approximated more accurately by floating point values than real numbers between 4096 and 8192.

The range of numbers that can be represented is approximately ±2.17*10^(-19) to ±1.84*10^19. Numbers are held with an accuracy of approximately 12 significant figures and this accuracy is maintained by the basic arithmetic procedures. The accuracy of the mathematical procedures is generally around 11 significant figures for most of the valid argument range, but may be less than this for arguments close to special values (e.g. the tangent of angles close to pi/2).

Special values

As well as representing real numbers, floating point vectors can represent three special values - illegal, infinity and minus infinity.

The special illegal value is generated as the result of any procedure which is called with parameters that do not allow a sensible result to be produced (e.g. dividing zero by zero). If this value is generated as an intermediate result in some calculation then it is propagated through the calculation i.e. any floating point procedure given this special value as a parameter produces this value as its result. The special illegal value is displayed by WRITEFP and WRITESG as a string of question marks ('?').

The special value infinity is generated as the result of any procedure whose result is too big to be held as a floating point number. Thus it may represent a large but finite number (e.g. as the result of 10^19 times itself) or it may genuinely represent infinity (e.g. as the result of one divided by zero). Calculations involving infinity behave as one would expect. Thus infinity plus any number gives infinity, one divided by infinity gives zero etc. Infinity is displayed by WRITEFP and WRITESG as a string of asterisks ('*').

The special value minus infinity is simply the negative of infinity and behaves as one would expect. Thus minus one divided by zero gives minus infinity and minus infinity times any negative number gives infinity. Minus infinity is displayed by WRITEFP and WRITESG as '-' followed by a string of asterisks.

Using floating point numbers

Vectors must be set up for all floating point numbers used in a program. The calculations header file, FPHDR, declares a manifest constant FP.LEN which may be used to obtain vectors for use as floating point numbers e.g.

```
GET "FPHDR"
LET my.number = VEC FP.LEN
LET another.number = GETVEC( FP.LEN)
```

In fact the declaration of FP.LEN in FPHDR is designed for use with both the floating point and fixed point calculation routines and so the vector it obtains is one word larger than it need be for floating point use. Thus when writing programs which are not required to work with both floating point and fixed point then space will be saved by allocating vectors of the exact size. This may be done either by using (FP.LEN-1) instead of FP.LEN or by redefining FP.LEN as 2 after the GET statement for FPHDR.

Initialising floating point numbers

Because floating point numbers are held as vectors they cannot be initialised like ordinary variables by using LET statements such as:

```
LET I = 3
```

There are three ways to initialise floating point numbers - using FLIT, using FFLOAT and using TABLES.

The procedure FLIT allows a number to be specified as an ASCII string. For example, to initialise a number to -2.5 the following code could be used:

```
LET minus.five.by.two = VEC FP.LEN-1
FLIT( "-2.5", minus.five.by.two)
```

The disadvantage with this method is that each time the program is run the relatively slow ASCII to floating point conversion must be performed.

A better method, which can be used only if the floating point number is to be initialised to an integer between MININT and MAXINT, is to use FFLOAT e.g.

```
LET one.thousand = VEC FP.LEN-1
FFLOAT( 1000, one.thousand)
```

The third method is to set up a table containing the internal representation of the number required. This provides the fastest execution time but has the major disadvantage that programs cannot easily be transferred to other BCPL systems.

If the number is to be used only as a constant then a TABLE is all that is needed. If the number is to be used as a variable then a TABLE should be set up and copied to the variable. The following example illustrates both techniques:

```
LET constant.ten.thousand = TABLE #X384C, #X0080,
                                  #X0000
LET mynum = VEC FP.LEN-1

// Initialise mynum to 100000000

MOVE( (TABLE #X7D59, #X4078, #X0000),
      mynum, FP.LEN)

// Assume there is code here to do calculations
// which store some result in mynum

// Divide mynum by 10000

FDIV( mynum, constant.ten.thousand, mynum)
```

Using this method of initialisation means that the internal representation of the initial value must be determined when the program is coded. The easiest method of doing this is to write a small program which reads a floating point number and displays it in hex, e.g.

```
GET "LIBHDR"
GET "FPHDR"
LET START() BE
$( LET number = VEC FP.LEN-1
   READFP( number)
   FOR I = 0 TO FP.LEN-1 DO
      WRITEF("%X4  ", number!I)
   NEWLINE()
$)
```

Before such a program can be run it must be linked with the floating point procedures as described in chapter 5.

FLOATING POINT PROCEDURES

The calculations header file FPHDR contains global declarations for all the floating point procedures. The procedures can be grouped by function as follows:

Arithmetic procedures

FABS	find absolute value of number;
FCOMP	compare two numbers;
FDIV	divide two numbers;
FMINUS	subtract two numbers;
FMULT	multiply two numbers;
FNEG	negate a number;
FPLUS	add two numbers;
FSGN	find sign of a number.

Input/output procedures

FLIT	convert ASCII string to number;
READFP	read a number from the current input stream;
WRITEFP	write a number to the current output stream with a specified number of decimal places;
WRITESG	write a number to the current output stream with a specified number of significant digits.

Conversion procedures

FFIX find nearest integer to floating point number;

FFLOAT convert integer to floating point number;

FINT find integer part of floating point number.

Mathematical procedures

FACOS arccosine;
FASIN arcsine;
FATAN arctangent;
FCOS cosine;
FDEG convert radians to degrees;
FE return value of the constant e;
FEXP exponential function (e to the power x);
FLN natural logarithm (base e);
FPI return value of the constant pi;
FRAD convert degrees to radians;
FRND generate a pseudo-random number;
FSIN sine;
FSQRT square root;
FTAN tangent.

The trigonometric procedures work with angles expressed in radians rather than degrees.

Use of floating point procedures

Any floating point procedure which gives a floating point result must have a parameter specifying the vector to be used for the result as well as parameters specifying the arguments to the procedure. Thus FPLUS, which adds two numbers together, has three parameters which are the vectors containing the two numbers to be added and the vector to contain the sum. By convention the vector for the result is always the last parameter.

The value returned by such a procedure is always the address of the result vector. This allows the use of a floating point procedure call as the parameter to a floating point procedure.

For example, if veca, vecb, vecc and vecd are all floating point numbers then

```
vecd := veca + vecb*vecc
```

could be coded either as:

```
FMULT( vecb, vecc, vecd)   // vecd := vecb*vecc
FPLUS( veca, vecd, vecd)   // vecd := veca+vecd
```

or as:

```
FPLUS( veca, FMULT( vecb, vecc, vecd), vecd)
```

As the example above shows, the same vector may be used both as an argument and as the result. Thus the procedure call

```
FPLUS( veca, veca, veca)
```

would double the number in veca.

It is sometimes necessary to provide one or more vectors simply to hold intermediate results in a calculation. Thus to calculate

```
veca := ( veca + vecb)/( veca - vecb)
```

another vector would be needed (vecz, say) and the calculation could be coded as:

```
FPLUS( veca, vecb, vecz)  // vecz := veca+vecb
FMINUS( veca, vecb, veca) // veca := veca-vecb
FDIV( vecz, veca, veca)   // veca := vecz/veca
```

The last two lines could be combined as

```
FDIV( vecz, FMINUS( veca, vecb, veca), veca)
```

but it is important to note that the calculation cannot be coded as one statement. An attempt to do so such as:

```
FDIV( FPLUS( veca, vecb, vecz),
      FMINUS( veca, vecb, veca),
      veca)
```

could fail because the order of evaluation of procedure arguments is not defined and therefore the call to FMINUS, which changes veca, might be made before the call to FPLUS which needs the original value of veca.

ERROR HANDLING

As a general rule the floating point procedures handle errors by setting the global variable FPEXCEP to a non-zero error code identifying the particular error that has occurred. The result returned depends on the type of error, but is intended to be the most sensible possible in the circumstances.

If an error occurs at an early stage in a complex calculation it is likely to cause other errors in later stages. To prevent these other errors changing FPEXCEP, and hence masking the original error, the procedures only change FPEXCEP if it is zero. Thus in the sections describing individual procedures, statements such as 'FPEXCEP is set to 7' should strictly be interpreted as 'if FPEXCEP is zero then it is set to 7'.

By setting FPEXCEP to zero before a series of calculations and checking it afterwards a program can discover if one or more errors have occurred and, if so, the cause of the first to occur.

FPEXCEP is not initialised to zero before a program is entered and it is never reset to zero by any of the floating point procedures.

The error codes are:

1 Division by zero. The result is either infinity or minus infinity, depending on the sign of the dividend.

2 Attempt to convert a floating point number of magnitude greater than MAXINT to an integer. The result is either MAXINT or MININT, depending on the sign of the number.

3 Overflow. The true result of an operation has a magnitude greater than the maximum value that can be held as a floating point number. The result returned is either infinity or minus infinity depending on the sign of the true result.

4 Underflow. The true result of an operation is non-zero but smaller in magnitude than the smallest non-zero number that can be held as a floating point value. The result returned is zero. This error is only generated by the basic arithmetic procedures (FPLUS, FMINUS, FMULT, FDIV). Other procedures (e.g. FSIN) return a result of 0 but do not change FPEXCEP.

5 Square root of negative number. The result is the square root of the absolute value of the number.

6 Arcsin or arccos of a number greater in magnitude than one. The result is the special illegal value.

7 Illegal operation. Some examples of illegal operations are dividing zero by zero, multiplying infinity by zero, subtracting infinity from infinity and dividing infinity by infinity. The result is the special illegal value.

8 Natural logarithm of zero or a negative number. The result is minus infinity if the parameter is zero or the natural logarithm of the absolute value of the number otherwise.

9 Reading a floating point number from a string using FLIT has not used all the characters in the string. This error indicates that the string cannot be interpreted as a valid floating point number (e.g. if the string were "123.5.6" FLIT would return a value of 123.5 and set FPEXCEP to 9).

If a procedure generates a result of infinity or the special illegal value purely as a result of a parameter being infinity or the special illegal value then this is not considered an error and FPEXCEP is not changed. For example, adding two large numbers together might generate an overflow error in which case FPEXCEP would be set to 3 and the result would be set to infinity whereas adding any finite value to infinity produces a result of infinity but does not change FPEXCEP.

FABS - absolute value

Purpose:

To obtain the absolute value of a floating point number.

Example:

```
result.b := FABS( veca, vecb)
```

Function:

Vecb is set to the absolute value of **veca**, i.e. to a positive number with the same magnitude as **veca.**

The result is a pointer to the destination vector **vecb.**

FACOS - arccosine

Purpose:

To obtain the arccosine of a floating point number, i.e. the angle whose cosine is the given number.

Examples:

```
result.b := FACOS( veca, vecb)
FACOS( cos.x, x)
```

Function:

Veca should contain a number between -1 and 1 inclusive. **Vecb** is set to the angle (in radians) whose cosine is **veca.** This angle is in the range 0 to pi inclusive.

The result is a pointer to the destination vector **vecb.**

If **veca** is greater in magnitude than 1 then FPEXCEP is set to 6 and **vecb** is set to the special illegal value.

FASIN - arcsine

Purpose:

To obtain the arcsine of a floating point number, i.e. the angle whose sine is the given number.

Examples:

```
result.b := FASIN( veca, vecb)
FASIN( sin.x, x)
```

Function:

Veca should contain a number between -1 and 1 inclusive. **Vecb** is set to the angle (in radians) whose sine is **veca.** This angle is in the range -pi/2 to pi/2 inclusive.

The result is a pointer to the destination vector **vecb.**

If **veca** is greater in magnitude than 1 then FPEXCEP is set to 6 and **vecb** is set to the special illegal value.

FATAN - arctangent

Purpose:

To obtain the arctangent of a floating point number, i.e. the angle whose tangent is the given number.

Examples:

```
result.b := FATAN( veca, vecb)
FATAN( tan.x, x)
```

Function:

Vecb is set to the angle (in radians) whose tangent is **veca.** This angle is in the range -pi/2 to pi/2 inclusive.

The result is a pointer to the destination vector **vecb.**

FCOMP - compare

Purpose:

To compare two floating point numbers.

Example:

```
comparison := FCOMP( veca, vecb)
```

Function:

Comparison is

-1	if **veca** is less than **vecb**
0	if **veca** is equal to **vecb**
1	if **veca** is greater than **vecb.**

Remarks:

If one of the parameters to FCOMP is the special illegal value the result is unpredictable (the value is treated either as infinity or as minus infinity).

The limitations in accuracy of floating point arithmetic mean that numbers which should theoretically be equal may actually differ by a small amount (i.e. small relative to the sizes of the numbers involved). Thus when comparing numbers for equality it may be safer to compare their difference with some relatively small value.

FCOS - cosine

Purpose:

To obtain the cosine of an angle. The angle and the result are both floating point numbers.

Examples:

```
result.b := FCOS( veca, vecb)
FCOS( x, cos.x)
```

Function:

Vecb is set to the cosine of the angle given by **veca**. **Veca** must be in radians.

The absolute value of **veca** should be less than 8192*pi (approximately 25735). If it is outside this range then FPEXCEP is set to 7 and **vecb** is set to the special illegal value.

The result is a pointer to the destination vector **vecb.**

FDEG - radians to degrees

Purpose:
To convert an angle expressed in radians to one expressed in degrees. Both angles are floating point numbers.

Examples:
```
result.b := FDEG( veca, vecb)
FDEG( radians, degrees)
```

Function:
Vecb is set to the number of degrees equivalent to **veca** radians. One radian is approximately equal to 57 degrees.

The result is a pointer to the destination vector **vecb.**

FDIV - divide

Purpose:
To divide two floating point numbers.

Example:
```
result.c := FDIV( veca, vecb, vecc)
```

Function:
Vecc is set equal to the result of dividing **veca** by **vecb.**

If **veca** and **vecb** are both zero or are both infinite then FPEXCEP is set to 7 and **vecc** to the special illegal value. If **vecb** is zero and **veca** is non-zero then FPEXCEP is set to 1 and **vecc** is set to infinity or minus infinity depending on the sign of **veca.** If **vecb** is infinite and **veca** is not then **vecc** is set to zero.

The result is a pointer to the destination vector **vecc.**

FE - give value of e

Purpose:
To set a floating point number equal to the mathematical constant e.

Example:
```
result.a := FE( veca)
```

Function:
Veca is set equal to the value of e and the result is a pointer to **veca.**

The value of e is approximately 2.718281828.

FEXP - exponential function

Purpose:
To calculate the value of e raised to a power. The power and the result are both floating point numbers.

Examples:
```
result.b := FEXP( veca, vecb)
FEXP( power, e.to.the.power)
```

Function:
Vecb is set to e to the power **veca.**

The result is a pointer to the destination vector **vecb.**

FFIX - nearest integer

Purpose:
To return the nearest integer to a floating point number.

Example:

```
nearest.integer := FFIX( veca)
```

Function:

The result is the nearest integer to the floating point number **veca.** If **veca** is exactly between two integers then it is rounded up if positive, down if negative. For example if the value of **veca** is 234.5 then the nearest integer is defined to be 235 whereas if the value of **veca** is -234.5 then the nearest integer is defined to be -235.

If the nearest integer to **veca** is greater than MAXINT or less than MININT then FPEXCEP is set to 2 and the result is MAXINT or MININT as appropriate.

Remarks:

If **veca** is the special illegal value the result is unpredictable but will be either MAXINT or MININT (with FPEXCEP set to 2).

FFLOAT - convert integer to floating point

Purpose:

To set a floating point number to the value of an integer.

Examples:

```
result.a := FFLOAT( integer, veca)
FFLOAT( -1, minus.one)
```

Function:

Veca is set to the value of **integer.**

The result is a pointer to the destination vector **veca.**

FINT - integer part

Purpose:

To return the integer part of a floating point number.

Example:

```
integer.part := FINT( veca)
```

Function:

The result is the integer part of the floating point number **veca.** This is defined to be the largest integer not greater than **veca.** Thus if **veca** were -2.1 the result would be -3 whereas if **veca** were 2.1 the result would be 2.

If the integer part of **veca** is greater than MAXINT or less than MININT then FPEXCEP is set to 2 and the result is MAXINT or MININT as appropriate.

Remarks:

If **veca** is the special illegal value the result is unpredictable but will be either MAXINT or MININT (with FPEXCEP set to 2).

FLN - natural logarithm

Purpose:

To calculate the natural logarithm of a given value. The value and the result are both floating point numbers.

Examples:

```
result.b := FLN( veca, vecb)
FLN( x, ln.x)
```

Function:

Vecb is set to the natural logarithm (i.e. logarithm base e) of **veca.**

If **veca** is 0 FPEXCEP is set to 8 and **vecb** is set to minus infinity. If **veca** is negative FPEXCEP is set to 8 and **vecb** is set to the logarithm of the absolute value of **veca.**

The result is a pointer to the destination vector **vecb.**

FLIT - convert string to number

Purpose:

To initialise a floating point number using a string.

Examples:

```
result.a := FLIT( string, veca)
FLIT( "-0.5", minus.a.half)
```

Function:

Veca is set up with the number specified by **string.** The conversion used is the same as used in the procedure READFP (see below). Thus all the following strings would be valid to specify the value 0.25:

- "0.25"
- " .25"
- "+0.25E+0"
- " 250E-3"
- "0.0025E2".

The result is a pointer to the destination vector **veca.**

Remarks:

If the conversion stops (because a character which is not valid as part of a floating point number is encountered) before the end of the string is reached then FPEXCEP is set to 9. The result returned in **veca** is the number represented by the string up to the invalid character (0 if the first character is invalid.

Thus if the string were "123.5A+06" the result would be 123.5. If the string were "Z12" the result would be 0.

Even trailing spaces cause FPEXCEP to be set to 9. Thus a string of "123 " would give a result of 123 but would also set FPEXCEP.

FMINUS - subtract

Purpose:
To subtract one floating point number from another.

Example:
result.c := FMINUS(veca, vecb, vecc)

Function:
Vecc is set equal to the result of subtracting **vecb** from **veca.**

If **veca** and **vecb** are both infinity or both minus infinity then FPEXCEP is set to 7 and **vecc** is set to the special illegal value.

The result is a pointer to the destination vector **vecc.**

FMULT - multiply

Purpose:
To multiply one floating point number by another.

Example:
result.c := FMULT(veca, vecb, vecc)

Function:
Vecc is set equal to the product of **veca** and **vecb.**

If one of **veca** and **vecb** is zero and the other is infinity or minus infinity then FPEXCEP is set to 7 and **vecc** is set to the special illegal value.

The result is a pointer to the destination vector **vecc.**

FNEG - negate

Purpose:
To find the negative of a floating point number.

Example:
```
result.b := FNEG( veca, vecb)
```

Function:
Vecb is set equal to the negative of **veca** and the result is a pointer to **vecb.**

FPI - give value of pi

Purpose:
To set a floating point number equal to the mathematical constant pi.

Example:
```
result.a := FPI( veca)
```

Function:
Veca is set equal to the value of pi and the result is a pointer to **veca.**

The value of pi is approximately 3.14159.

FPLUS - add

Purpose:
To add one floating point number to another.

Example:
```
result.c := FPLUS( veca, vecb, vecc)
```

Function:
Vecc is set equal to the sum of **veca** and **vecb.**

If one of **veca** and **vecb** is infinity and the other is minus infinity then FPEXCEP is set to 7 and **vecc** is set to the special illegal value.

The result is a pointer to the destination vector **vecc.**

FRAD - degrees to radians

Purpose:
To convert an angle expressed in degrees to one expressed in radians. Both angles are floating point numbers.

Examples:
```
result.b := FRAD( veca, vecb)
FRAD( degrees, radians)
```

Function:
Vecb is set to the number of radians equivalent to **veca** degrees. One radian is approximately equal to 57 degrees.

The result is a pointer to the destination vector **vecb.**

FRND - random

Purpose:

To generate a series of pseudo-random floating point numbers.

Examples:

```
result.b := FRND( veca, vecb)
FRND( seed, random.number)
```

Function:

Vecb is set to a pseudo-random number derived from **veca.** The number is in the range 0 <= **vecb** < 1.

The result is a pointer to the destination vector **vecb.**

Remarks:

This procedure is normally used to obtain a series of random numbers by starting with an arbitrary seed and then using the previous random number as the seed for the next one.

Since there are only a finite number of floating point numbers between 0 and 1 the series must eventually enter a repeating cycle, but this should not occur until at least 65535 numbers have been obtained.

When testing a program using random numbers it is useful to be able to repeat tests using the same series of numbers. This can be done by starting with the same seed.

There are various possible methods to obtain a random seed. One is to use the current time in some way. Another is to measure the time taken by the operator to respond to some prompt and use this.

FSGN - sign of number

Purpose:
To test whether a floating point number is less than zero, equal to zero or greater than zero.

Example:
```
sign := FSGN( veca)
```

Function:
Sign is

-1	if **veca** is less than zero
0	if **veca** is equal to zero
1	if **veca** is greater than zero.

FSIN - sine

Purpose:
To obtain the sine of an angle. The angle and the result are both floating point numbers.

Examples:
```
result.b := FSIN( veca, vecb)
FSIN( x, sin.x)
```

Function:
Vecb is set to the sine of the angle given by **veca**. **Veca** must be in radians.

The absolute value of **veca** should be less than 8192*pi (approximately 25735). If it is outside this range then FPEXCEP is set to 7 and **vecb** is set to the special illegal value.

The result is a pointer to the destination vector **vecb**.

FSQRT - square root

Purpose:

To calculate the square root of a floating point number.

Examples:

```
result.b := FSQRT( veca, vecb)
FSQRT( x, square.root.of.x)
```

Function:

Vecb is set to the square root of **veca.** If **veca** is negative then the absolute value is used and FPEXCEP is set to 5.

The result is a pointer to the destination vector **vecb.**

FTAN - tangent

Purpose:

To obtain the tangent of an angle. The angle and the result are both floating point numbers.

Examples:

```
result.b := FTAN( veca, vecb)
FTAN( x, tan.x)
```

Function:

Vecb is set to the tangent of the angle given by **veca.** **Veca** must be in radians.

The absolute value of **veca** should be less than 8192*pi (approximately 25735). If it is outside this range then FPEXCEP is set to 7 and **vecb** is set to the special illegal value.

The result is a pointer to the destination vector **vecb.**

READFP - read a number

Purpose:

To read a floating point number from the current input stream.

Example:

```
result.a := READFP( veca)
```

Function:

Veca is set up with a floating point number read from the current input stream. If the input stream does not provide a valid floating point number then **veca** is set to zero. The input stream is left so that the next character to be read is the character following the number.

The result is a pointer to **veca**.

Remarks:

Numbers may be input in both normal and scientific notation e.g. '104.6' or '1.046E2', where 'E' means 'times ten to the power'.

Leading layout characters (space, new line, new page etc) are ignored but numbers may not contain embedded layout characters.

The format of acceptable numbers is an optional sign ('+' or '-') followed by zero or more digits followed by an optional decimal point followed by zero or more digits followed by an optional exponent. The format of the exponent is 'E' or 'e' followed by an optional sign followed by zero or more digits.

Some examples of acceptable numbers are:

1234, -67.9, .123, +888.6123, 123e8, +.00005E15, -1234.56789E-10.

If the number is specified with more significant digits than can be accurately held then it is rounded. If the absolute value of the number is larger than can be held then FPEXCEP is set to 3 and **veca** is set to infinity.

WRITEFP - write a number with a specified number of decimal places

Purpose:

To write a floating point number either in a scientific format or in a format showing a specific number of decimal places.

Example:

WRITEFP(veca, width, places)

Function:

The floating point number in the vector **veca** is written to the current output stream in a field of **width** characters (or the minimum field width possible if **width** is less than 8).

If **places** is greater than zero then **veca** is output with that number of decimal places (provided that **width** allows this). If **places** is zero (or negative), or **width** is not sufficient for the specified number of decimal places, the number is output in scientific notation e.g. 1.2345E+05.

Remarks:

WRITEFP should be used in preference to WRITESG (see below) if the scientific format is required or if the approximate magnitude of the numbers to be displayed is known (so that a sensible choice can be made for the number of decimal places required).

If the scientific form of output is used then the following fields are output:

- spaces if necessary to make up the field width;

- a space if the number is positive or '-' if it is negative;

- one digit followed by a decimal point. This digit is chosen to be non-zero unless the value being output is zero;

- between one and 11 digits followed by the exponent. The number of digits displayed is governed by **width**. If fewer than 11 digits are displayed the last digit is rounded up if appropriate. The exponent is 'E' followed by '+' or '-' followed by a two-digit number (with a leading zero if necessary). If the value being output is zero the exponent is 'E+00'.

Thus if the number being displayed were -123.4567 and **width** were 20 the characters output would be

bb-1.23456700000E+02

where 'b' denotes a space character. Note that if this format is used and **width** is less than 8 then a width of 8 is used.

If the scientific form of output is not used then the following fields are output:

- spaces if necessary to make up the field width;

- a space if the number is positive or '-' if it is negative;

- one or more digits (representing the integral part of the number) followed by a decimal point. If the number is less than zero this field is '0.';

- **places** digits (representing the fractional part of the number). If less than 12 significant digits are output the last digit is rounded up if appropriate. If 12 significant digits have been output then this field is padded with trailing '0's if necessary. Thus at most 12 significant digits are shown.

Thus if the number being displayed were -123.4567, **width** were 15 and **places** were 6 the characters output would be

bbbb-123.456700

where 'b' denotes a space character. Note that this format uses a minimum field width of four characters.

If the value to be output is infinity or minus infinity then the format used is a space or '-' followed by enough asterisks ('*') to fill the field (if **width** is less than 4 then three asterisks are used).

If the value to be output is the special illegal value then the format used is as for infinity except that question marks ('?') are used instead of asterisks.

WRITESG - write a number with a specified number of significant digits

Purpose:

To write a floating point number showing a specific number of significant digits.

Example:

WRITESG(veca, width, digits)

Function:

The floating point number in the vector **veca** is written to the current output stream in a field of **width** characters (or the minimum field width possible if **width** proves to be too small). **Digits** significant digits are shown (except that leading zeros on the integral part and trailing zeros on the fractional part are suppressed).

If **width** is too small to allow **digits** significant digits to be shown in the normal format then a scientific format is used.

Remarks:

WRITESG is more appropriate than WRITEFP (see above) when the approximate magnitude of the numbers to be displayed is not known and so it is not sensible to display a fixed number of decimal places.

If **digits** is less than 2 or greater than 12 then it is treated as 2 or 12 respectively.

Veca is rounded to **digits** significant figures and then displayed in one of the following formats if **width** allows:

(1) if the rounded value is an integer:

- spaces if necessary to make up the field width;
- a space if the number is positive or '-' if it is negative;
- one or more digits (leading zeros are suppressed unless the value to be displayed is zero when a single '0' is output).

(2) if the rounded value is not an integer and is greater in magnitude than 1:

- spaces if necessary to make up the field width;
- a space if the number is positive or '-' if it is negative;
- one or more digits followed by '.' followed by one or more digits. At most **digits** digits are displayed. Trailing zeros are suppressed.

(3) if the value is not an integer and is less than 1 in magnitude:

- spaces if necessary to make up the field width;
- a space if the number is positive or '-' if it is negative;
- '0.' followed by zero or more '0's followed by up to **digits** digits (the first of which is non-zero). Trailing zeros are suppressed.

For example if the number to be displayed were -123.45, **width** were 10 and **digits** were 3 then the characters output would be:

bbbbbb-123

where ‘b’ denotes a space. If **digits** were 4, however, the output would be:

bbbb-123.5

If **width** is too small for the appropriate format as described above then the following format is used:

- spaces if necessary to make up the field width;

- a space if the number is positive or ‘-’ if it is negative;

- one digit followed by a decimal point. This digit is chosen to be non-zero unless the value being output is zero;

- **digits**-1 digits followed by the exponent. The exponent is ‘E’ followed by ‘+’ or ‘-’ followed by a two-digit number (with a leading zero if necessary). If the value being output is zero the exponent is ‘E+00’.

If the value to be output is infinity or minus infinity then the format used is a space or ‘-’ followed by enough asterisks (‘*’) to fill the field (if **width** is less than 4 then three asterisks are used).

If the value to be output is the special illegal value then the format used is as for infinity except that question marks (‘?’) are used instead of asterisks.

3 Fixed point

This chapter contains all the information needed by a user of the fixed point facilities of the BCPL Calculations Package except for the details of linking in the procedures required (which are described in chapter 5).

It contains a description of various features of the fixed point facilities followed by one section for each of the fixed point procedures.

Standardisation

This implementation of fixed point is designed to be a compatible superset of the floating point facilities specified by BCPL Language Extension A13b defined in 'A Proposed Definition of the Language BCPL' published in October 1979.

Programmers wishing to write programs which are compatible with other BCPL systems supporting this language extension should not make use of the internal format of fixed point numbers and should use only the following procedures:

FABS, FCOMP, FDIV, FFIX, FFLOAT, FMINUS, FMULT, FNEG, FPLUS, READFP and WRITEFP.

In some floating point implementations FFIX may return the integer part of a floating point number (i.e. the equivalent of FINT in this implementation) rather than the nearest integer to the number.

FIXED POINT NUMBERS

Fixed point numbers are held in four-word vectors. Each number is stored as a 14-digit decimal value with ten digits before the decimal point and four digits after it. Storing numbers as decimal digits avoids the rounding errors inherent in the binary floating point representation used by the floating point routines.

Thus calculations involving numbers of magnitude up to 9999999999.9999 may be performed with an accuracy of ±0.00005.

Note that there is no way of representing numbers larger in magnitude than 9999999999.9999 or numbers which are non-zero but are smaller in magnitude than 0.0001.

Using fixed point numbers

Vectors must be set up for all fixed point numbers used in a program. The calculations header file, FPHDR, declares a manifest constant FP.LEN which may be used to obtain vectors for use as fixed point numbers e.g.

```
GET "FPHDR"
LET my.number = VEC FP.LEN
LET another.number = GETVEC( FP.LEN)
```

Initialising fixed point numbers

Because fixed point numbers are held as vectors they cannot be initialised like ordinary variables by using LET statements such as:

```
LET I = 3
```

There are three ways to initialise fixed point numbers - using FLIT, using FFLOAT and using TABLEs.

The procedure FLIT allows a number to be specified as an ASCII string. For example, to initialise a number to -2.5 the following code could be used:

```
LET minus.five.by.two = VEC FP.LEN
FLIT( "-2.5", minus.five.by.two)
```

The disadvantage with this method is that each time the program is run the relatively slow ASCII to fixed point conversion must be performed.

A better method, which can be used only if the fixed point number is to be initialised to an integer between MININT and MAXINT, is to use FFLOAT e.g.

```
LET one.thousand = VEC FP.LEN
FFLOAT( 1000, one.thousand)
```

The third method is to set up a table containing the internal representation of the number required. This provides the fastest execution time but has the major disadvantage that programs cannot easily be transferred to other BCPL systems supporting equivalent floating point procedures.

If the number is to be used only as a constant then a TABLE is all that is needed. If the number is to be used as a variable then a TABLE should be set up and copied to the variable. The following example illustrates both techniques:

```
LET constant.1234 = TABLE #X0000, #X0000,
                          #X3412, #X0000
LET mynum = VEC FP.LEN

// Initialise mynum to 87654321.9553

MOVE( (TABLE #X0000, #X6587, #X2143, #X5395),
      mynum, FP.LEN+1)
```

```
// Assume there is code here to do calculations
// which store some result in mynum, then
// divide mynum by 1234

FDIV( mynum, constant.1234; mynum)
```

Using this method of initialisation means that the internal representation of the initial value must be determined when the program is coded. The easiest method of doing this is to write a small program which reads a fixed point number and displays it in hex, e.g.

```
GET "LIBHDR"
GET "FPHDR"
LET START() BE
$( LET number = VEC FP.LEN
   READFP( number)
   FOR I = 0 TO FP.LEN DO
      WRITEF("%X4 ", number!I)
   NEWLINE()
$)
```

Before such a program can be run it must be linked with the fixed point procedures as described in chapter 5.

FIXED POINT PROCEDURES

The calculations header file FPHDR contains global declarations for all the fixed point procedures. The procedures can be grouped by function:

Arithmetic procedures

```
FABS     find absolute value of number;
FCOMP    compare two numbers;
FDIV     divide two numbers;
FMINUS   subtract two numbers;
FMULT    multiply two numbers;
FNEG     negate a number;
FPLUS    add two numbers;
FSGN     find sign of a number.
```

Input/output procedures

FLIT convert ASCII string to number;

READFP read a number from the current input stream;

WRITEFP write a number to the current output stream.

Conversion procedures

FFIX find nearest integer to fixed point number;

FFLOAT convert integer to fixed point number;

FINT find integer part of fixed point number.

Mathematical procedures

FE return value of the constant e;
FEXP exponential function (e to the power x);
FLN natural logarithm (base e);
FSQRT square root.

Use of fixed point procedures

Any fixed point procedure which gives a fixed point result must have a parameter specifying the vector to be used for the result as well as parameters specifying the arguments to the procedure. Thus FPLUS, which adds two numbers together, has three parameters which are the vectors containing the two numbers to be added and the vector to contain the sum. By convention the vector for the result is always the last parameter.

The value returned by such a procedure is always the address of the result vector. This allows the use of a fixed point procedure call as the parameter to a fixed point procedure.

For example, if veca, vecb, vecc and vecd are all fixed point numbers then

```
vecd := veca + vecb*vecc
```

could be coded either as:

```
FMULT( vecb, vecc, vecd)   // vecd := vecb*vecc
FPLUS( veca, vecd, vecd)   // vecd := veca+vecd
```

or as:

```
FPLUS( veca, FMULT( vecb, vecc, vecd), vecd)
```

As the example above shows, the same vector may be used both as an argument and as the result. Thus the procedure call

```
FPLUS( veca, veca, veca)
```

would double the number in veca.

Note that it is sometimes necessary to provide one or more vectors simply to hold intermediate results in a calculation. Thus to calculate

```
veca := ( veca + vecb)/( veca - vecb)
```

another vector would be needed (vecz, say) and the calculation could be coded as:

```
FPLUS( veca, vecb, vecz)  // vecz := veca+vecb
FMINUS( veca, vecb, veca) // veca := veca-vecb
FDIV( vecz, veca, veca)   // veca := vecz/veca
```

The last two lines could be combined as

```
FDIV( vecz, FMINUS( veca, vecb, veca), veca)
```

It is important to note that the calculation cannot be coded as one statement. An attempt to do so such as:

```
FDIV( FPLUS( veca, vecb, vecz),
      FMINUS( veca, vecb, veca),
      veca)
```

could fail because the order of evaluation of procedure arguments is not defined and therefore the call to FMINUS, which changes veca, might be made before the call to FPLUS which needs the original value of veca.

ERROR HANDLING

As a general rule the fixed point procedures handle errors by setting the global variable FPEXCEP to a non-zero error code identifying the particular error that has occurred. The result returned depends on the type of error, but is intended to be the most sensible possible in the circumstances.

If an error occurs at an early stage in a complex calculation it is likely to cause other errors in later stages. To prevent these other errors changing FPEXCEP, and hence masking the original error, the procedures only change FPEXCEP if it is zero. Thus in the sections describing individual procedures, statements such as 'FPEXCEP is set to 7' should strictly be interpreted as 'if FPEXCEP is zero then it is set to 7'.

By setting FPEXCEP to zero before a series of calculations and checking it afterwards a program can discover if one or more errors have occurred and, if so, the cause of the first to occur.

FPEXCEP is not initialised to zero before a program is entered and it is never reset to zero by any of the fixed point procedures.

The error codes are:

1 Division by zero. The result is either the largest value that can be represented (9999999999.9999) or the negative of this value, depending on the sign of the dividend.

2 Attempt to convert a fixed point number of magnitude greater than MAXINT to an integer. The result is either MAXINT or MININT, depending on the sign of the number.

3 Overflow. The true result of an operation has a magnitude greater than the maximum value that can be held as a fixed point number. The result returned is either the largest value that can be represented or the negative of this value depending on the sign of the true result.

5 Square root of negative number. The result is the square root of the absolute value of the number.

8 Natural logarithm of zero or a negative number. The result is -9999999999.9999 if the parameter is zero or the natural logarithm of the absolute value of the number otherwise.

9 Reading a fixed point number from a string using FLIT has not used all the characters in the string. This error indicates that the string cannot be interpreted as a valid fixed point number (e.g. if the string were "123.5.6" FLIT would return a value of 123.5 and set FPEXCEP to 9).

FABS - absolute value

Purpose:
To obtain the absolute value of a fixed point number.

Example:
```
result.b := FABS( veca, vecb)
```

Function:
Vecb is set to the absolute value of **veca**, i.e. to a positive number with the same magnitude as **veca.**

The result is a pointer to the destination vector **vecb.**

FCOMP - compare

Purpose:
To compare two fixed point numbers.

Example:
```
comparison := FCOMP( veca, vecb)
```

Function:
Comparison is

-1	if **veca** is less than **vecb**
0	if **veca** is equal to **vecb**
1	if **veca** is greater than **vecb.**

FDIV - divide

Purpose:
To divide one fixed point number by another.

Example:
```
result.c := FDIV( veca, vecb, vecc)
```

Function:

Vecc is set equal to the result of dividing **veca** by **vecb.**

If **vecb** is zero and **veca** is non-zero then FPEXCEP is set to 1 and **vecc** is set to plus or minus 9999999999.9999 depending on the sign of **veca.** If both **veca** and **vecb** are zero then FPEXCEP is set to 1 and **vecc** is undefined.

The result is a pointer to the destination vector **vecc.**

FE - give value of e

Purpose:

To set a fixed point number equal to the mathematical constant e.

Example:

```
result.a := FE( veca)
```

Function:

Veca is set equal to the value of e and the result is a pointer to **veca.**

The value of e is approximately 2.7183.

FEXP - exponential function

Purpose:

To calculate the value of e raised to a power. The power and the result are both fixed point numbers.

Examples:

```
result.b := FEXP( veca, vecb)
FEXP( power, e.to.the.power)
```

Function:

Vecb is set to e to the power **veca.**

The result is a pointer to the destination vector **vecb.**

FFIX - nearest integer

Purpose:
To return the nearest integer to a fixed point number.

Example:
```
nearest.integer := FFIX( veca)
```

Function:
The result is the nearest integer to the fixed point number **veca.** If **veca** is exactly between two integers then it is rounded up if positive, down if negative. For example if the value of **veca** is 234.5 then the nearest integer is defined to be 235 whereas if the value of **veca** is -234.5 then the nearest integer is defined to be -235.

If the nearest integer to **veca** is greater than MAXINT or less than MININT then FPEXCEP is set to 2 and the result is MAXINT or MININT as appropriate.

FFLOAT - convert integer to fixed point

Purpose:
To set a fixed point number to the value of an integer.

Examples:
```
result.a := FFLOAT( integer, veca)
FFLOAT( -1, minus.one)
```

Function:
Veca is set to the value of **integer.**

The result is a pointer to the destination vector **veca.**

FINT - integer part

Purpose:

To return the integer part of a fixed point number.

Example:

```
integer.part := FINT( veca)
```

Function:

The result is the integer part of the fixed point number **veca.** This is defined to be the largest integer not greater than **veca.** Thus if **veca** were -2.1 the result would be -3 whereas if **veca** were 2.1 the result would be 2.

If the integer part of **veca** is greater than MAXINT or less than MININT then FPEXCEP is set to 2 and the result is MAXINT or MININT as appropriate.

FLN - natural logarithm

Purpose:

To calculate the natural logarithm of a given value. The value and the result are both fixed point numbers.

Examples:

```
result.b := FLN( veca, vecb)
FLN( x, ln.x)
```

Function:

Vecb is set to the natural logarithm (i.e. logarithm base e) of **veca.**

If **veca** is 0 FPEXCEP is set to 8 and **vecb** is set to -9999999999.9999. If **veca** is negative FPEXCEP is set to 8 and **vecb** is set to the logarithm of the absolute value of **veca.**

The result is a pointer to the destination vector **vecb.**

FLIT - convert string to number

Purpose:
To initialise a fixed point number using a string.

Examples:
```
result.a := FLIT( string, veca)
FLIT( "-0.5", minus.a.half)
```

Function:
Veca is set up with the number specified by **string.** The conversion used is the same as used in the procedure READFP (see below). Thus the following strings would be valid to specify the value 0.25:

- " 0.25"
- ".25"
- "+000.25".

The result is a pointer to the destination vector **veca.**

Remarks:
If the conversion stops (because a character which is not valid as part of a fixed point number is encountered) before the end of the string is reached then FPEXCEP is set to 9. The result returned in **veca** is the number represented by the string up to the invalid character (0 if the first character is invalid).

Thus if the string were "123.5A+06" the result would be 123.5. If the string were "Z12" the result would be 0.

Even trailing spaces cause FPEXCEP to be set to 9. Thus a string of "123 " would give a result of 123 but would also set FPEXCEP.

FMINUS - subtract

Purpose:
To subtract one fixed point number from another.

Example:
result.c := FMINUS(veca, vecb, vecc)

Function:
Vecc is set equal to the result of subtracting **vecb** from **veca.**

The result is a pointer to the destination vector **vecc.**

FMULT - multiply

Purpose:
To multiply one fixed point number by another.

Example:
result.c := FMULT(veca, vecb, vecc)

Function:
Vecc is set equal to the product of **veca** and **vecb.**

The result is a pointer to the destination vector **vecc.**

FNEG - negate

Purpose:
To find the negative of a fixed point number.

Example:
```
result.b := FNEG( veca, vecb)
```

Function:
Vecb is set equal to the negative of **veca** and the result is a pointer to **vecb.**

FPLUS - add

Purpose:
To add one fixed point number to another.

Example:
```
result.c := FPLUS( veca, vecb, vecc)
```

Function:
Vecc is set equal to the sum of **veca** and **vecb.**

The result is a pointer to the destination vector **vecc.**

FSGN - sign of number

Purpose:
To test whether a fixed point number is less than zero, equal to zero or greater than zero.

Example:
```
sign := FSGN( veca)
```

Function:
Sign is

-1	if **veca** is less than zero
0	if **veca** is equal to zero
1	if **veca** is greater than zero.

FSQRT - square root

Purpose:

To calculate the square root of a fixed point number.

Examples:

```
result.b := FSQRT( veca, vecb)
FSQRT( x, square.root.of.x)
```

Function:

Vecb is set to the square root of **veca.** If **veca** is negative then the absolute value is used and FPEXCEP is set to 5.

The result is a pointer to the destination vector **vecb.**

READFP - read a number

Purpose:

To read a fixed point number from the current input stream.

Example:

```
result.a := READFP( veca)
```

Function:

Veca is set up with a fixed point number read from the current input stream. If the input stream does not provide a valid fixed point number then **veca** is set to zero. The input stream is left so that the next character to be read is the character following the number.

The result is a pointer to **veca.**

Remarks:

Leading layout characters (space, new line, new page etc) are ignored but numbers may not contain embedded layout characters.

The format of acceptable numbers is an optional sign ('+' or '-') followed by zero or more digits followed by an optional decimal point followed by-zero or more digits.

Some examples of acceptable numbers are:

1234, -67.9, .123, +888.6123.

If the number is specified with more than four decimal places then it is rounded. If the absolute value of the number is larger than can be held then FPEXCEP is set to 3 and **veca** is set to plus or minus 9999999999.9999 as appropriate.

WRITEFP - write a number

Purpose:
To write a fixed point number.

Example:
WRITEFP(veca, width, places)

Function:
The fixed point number in the vector **veca** is written to the current output stream in a field of **width** characters (or the minimum field width possible if **width** is too small to output the number with **places** decimal places).

If **places** is greater than zero then **veca** is output with that number of decimal places. If **places** is zero (or negative) then **veca** is output with no decimal point.

Remarks:

The following fields are output:

- spaces if necessary to make up the field width;

- a space if the number is positive or '-' if it is negative;

- one or more digits (representing the integral part of the number). For numbers less than one this digit may be '0';

- if **places** is greater than zero then a decimal point followed by **places** digits. All digits after the first four are '0'.

If **places** is less than four the number is rounded to the specified number of decimal places.

Thus if the number being displayed were -123.4567, **width** were 15 and **places** were 6 the characters output would be

bbbb-123.456700

where 'b' denotes a space character.

4 Fast integer

This chapter contains all the information needed by a user of the fast integer facilities of the BCPL Calculations Package except for the details of linking in the procedures required (which are described in chapter 5).

The following fast integer procedures are provided:

ASN calculate arcsine
COS calculate cosine
SIN calculate sine
SQR calculate square root.

Formulae for calculating other functions (e.g. arccosine) are given in the Appendix.

Use of the fast integer procedures

The three trigonometric procedures (ASN, COS and SIN) scale their arguments and their results by a factor of 10000. Thus the following code sets the variable theta to the angle whose sine is 0.5:

```
theta := ASN( 5000)  // 5000 is 0.5 * 10000
```

This sets theta to 5236 which represents 5236 divided by 10000 radians i.e. 0.5236 radians.

Similarly the code:

```
sin.theta := SIN( 5236)
```

sets the variable sin.theta to 5000.

The accuracy of these procedures is ±2 in the last digit (i.e. ±0.0002 in real terms).

The square root procedure has two parameters and returns the square root of their product. This is especially convenient when working with the scaled quantities used by the trigonometric procedures. For example if the variable scaled.num represents some number (x, say) times 10000, then

```
SQR( scaled.num, 10000)
```

returns a value which is 10000 times the square root of x.

The result provided by SQR is as accurate as possible, i.e. the error is no more than ±0.5.

The standard library procedure MULDIV is often useful when working with the fast integer procedures. For example the following procedure calculates the arctangent of its parameter (with the usual scaling by 10000), assuming that the parameter is less than about 15000, using the formula:

```
arctan(x) = arcsin(x/(sqr(1+x*x)))
```

```
LET ATN(X) = VALOF
$( LET T = MULDIV( X, X, 10000)  // x*x
   T := SQR( 10000+T, 10000)     // sqr(1+x*x)
   T := MULDIV( X, 10000, T)     // x/sqr(1+x*x)
   RESULTIS ASN( T)
$)
```

Note that this procedure can be written in one line, but the coding above shows each stage in the calculation separately for clarity. The factor of 10000 is maintained at each stage.

ASN - arcsine

Purpose:
To calculate the arcsine (in radians) of a value, i.e. the angle whose sine is that value.

Example:
```
angle := ASN( value)
```

Function:
Value is expected to be in the range -10000 to 10000, representing -1.0000 to 1.0000. **Angle** is an integer in the range -15708 to 15708 representing -pi/2 to pi/2 radians.

Remarks:
If value is less than -10000 or greater than 10000 then it is treated as -10000 or 10000 as appropriate.

COS - cosine

Purpose:
To calculate the cosine of an angle in radians.

Example:
```
cosine := COS( angle)
```

Function:
Angle is interpreted as 10000 times the angle whose cosine is required. **Cosine** is set to 10000 times the cosine of that angle i.e. it is in the range -10000 to 10000.

The range of **angle** is -32767 to 32767 representing -3.2767 to 3.2767 radians i.e. a little over -pi to pi.

SIN - sine

Purpose:
To calculate the sine of an angle in radians.

Example:
```
sine := SIN( angle)
```

Function:
Angle is interpreted as 10000 times the angle whose sine is required. **Sine** is set to 10000 times the sine of that angle i.e. it is in the range -10000 to 10000.

The range of **angle** is -32767 to 32767 representing -3.2767 to 3.2767 radians i.e. a little over -pi to pi.

SQR - square root

Purpose:
To calculate the square root of the product of two integers.

Example:
```
root := SQR( num.a, num.b)
```

Function:
The absolute values of **num.a** and **num.b** are multiplied together, giving a 30-bit result, and **root** is set to the positive square root of this result.

5 Linking the procedures

This chapter describes how to link in the various calculation procedures with programs that use them. It also contains a list of all the files supplied as part of the BCPL Calculations Package.

The calculations procedures are supplied as a number of CINTCODE files, each containing one or more sections each of which contains one or more procedures. Certain procedures use other procedures (e.g. the floating point trigonometric procedures use the floating point arithmetic procedures). Lists are given below showing the file and section containing each procedure and the other sections needed by each section.

The procedures have been grouped into files in a way that should be convenient for many applications. In some cases, however, it may be required to produce a CINTCODE file containing a specific set of procedures. This can be achieved by extracting the sections required from the existing files using the utility NEEDCIN. All sections which need other sections contain the appropriate NEEDS directives and so NEEDCIN can be used to check that all necessary sections have been included. Note that NEEDS directives specify section names, not procedure names.

There are three methods of linking in the procedures:

- Add the CINTCODE sections containing the required procedures to the CINTCODE of the program. Thus when the program is run the procedures are linked with it.

The sections may be added to the program either with the utility JOINCIN or, if the program contains suitable NEEDS directives, with the utility NEEDCIN.

This method is probably the best in most cases.

- Form the required procedures into a number of CINTCODE files and use the 'LINK file SYSTEM' command to link these files to the global vector. The files remain linked until specifically unlinked by the UNLINK command.

  This method may be most appropriate where a number of different programs using the calculations facilities are to be run as it saves having to include a copy of the procedures in each program.

- Link the procedures required by explicit code within the program (i.e. by calls to LOADSEG and GLOBIN).

  This method may be appropriate for programs which use both fixed and floating point procedures at different times, or for programs which use the calculation procedures for only part of the time and need the heap space taken up by the procedures for other purposes at other times.

LIST OF FILES

This section lists the files supplied as part of the BCPL Calculations Package.

FLARITH Floating point arithmetic, conversion and general purpose mathematical procedures - FABS, FCOMP, FDIV, FFIX, FFLOAT, FINT, FMINUS, FMULT, FNEG, FPLUS, FRND, FSGN and FSQRT.

FLEXP Floating point exponential procedures - FE, FEXP and FLN.

FLIO Floating point input/output procedures - FLIT, READFP, WRITEFP and WRITESG.

FLTRIG Floating point trigonometric procedures - FACOS, FASIN, FATAN, FCOS, FDEG, FPI, FRAD, FSIN and FTAN.

FPHDR Header file containing global declarations for all calculations procedures and manifest declaration for FP.LEN.

FXARITH Fixed point arithmetic, conversion and general purpose mathematical procedures - FABS, FCOMP, FDIV, FFIX, FFLOAT, FINT, FMINUS, FMULT, FNEG, FPLUS, FSGN and FSQRT.

FXEXP Fixed point exponential procedures - FE, FEXP and FLN.

FXIO Fixed point input/output procedures - FLIT, READFP and WRITEFP.

INTCALC Fast integer procedures - ASN, COS, SIN and SQR.

In addition a number of example files are provided. The names of these files all begin with the characters 'EX'. The file EXHELP is a text file giving details of the example files.

All examples are provided in both source and CINTCODE form. Copyright is waived on the source code of these examples. Thus the examples may be used and modified as required.

LIST OF PROCEDURES AND SECTIONS

This section lists, for each procedure, the names of the file and section containing that procedure. It also lists, for each section, the name of the file containing that section and any other sections needed by that section.

Floating point procedures

Procedure	File	Section
FABS	FLARITH	FLAR2
FACOS	FLTRIG	FASIN
FASIN	FLTRIG	FASIN
FATAN	FLTRIG	FTAN
FCOMP	FLARITH	FLAR1
FCOS	FLTRIG	FSIN
FDEG	FLTRIG	FCONST
FDIV	FLARITH	FLAR1
FE	FLEXP	FLEXP
FEXP	FLEXP	FLEXP
FFIX	FLARITH	FLCONV
FFLOAT	FLARITH	FLCONV
FINT	FLARITH	FLCONV
FLN	FLEXP	FLEXP
FLIT	FLIO	FLIO1
FMINUS	FLARITH	FLAR1
FMULT	FLARITH	FLAR1
FNEG	FLARITH	FLAR2
FPI	FLTRIG	FCONST
FPLUS	FLARITH	FLAR1
FRAD	FLTRIG	FCONST
FRND	FLARITH	FRND
FSGN	FLARITH	FLAR2
FSIN	FLTRIG	FSIN
FSQRT	FLARITH	FLSQRT
FTAN	FLTRIG	FTAN
READFP	FLIO	FLIO1
WRITEFP	FLIO	FLIO2
WRITESG	FLIO	FLIO2

Fixed point procedures

Procedure	File	Section
FABS	FXARITH	FXAR2
FCOMP	FXARITH	FXAR1
FDIV	FXARITH	FXAR1
FE	FXEXP	FXEXP
FEXP	FXEXP	FXEXP
FFIX	FXARITH	FXCONV
FFLOAT	FXARITH	FXCONV
FINT	FXARITH	FXCONV
FLN	FXEXP	FXEXP
FLIT	FXIO	FXIO
FMINUS	FXARITH	FXAR1
FMULT	FXARITH	FXAR1
FNEG	FXARITH	FXAR2
FPLUS	FXARITH	FXAR1
FSGN	FXARITH	FXAR2
FSQRT	FXARITH	FXSQRT
READFP	FXIO	FXIO
WRITEFP	FXIO	FXIO

Fast integer procedures

Procedure	File	Section
ASN	INTCALC	INTASN
COS	INTCALC	INTSIN
SIN	INTCALC	INTSIN
SQR	INTCALC	INTSQR

Sections

Section	File	Other sections needed
FASIN	FLTRIG	FLAR1, FLAR2, FLSQRT
FCONST	FLTRIG	FLAR1
FLAR1	FLARITH	None
FLAR2	FLARITH	None
FLCONV	FLARITH	None
FLEXP	FLEXP	FLAR1, FLAR2, FLCONV
FLIO1	FLIO	FLAR1, FLCONV
FLIO2	FLIO	FLAR1, FLAR2, FLCONV

Section	File	Other sections needed
FLSQRT	FLARITH	FLAR1, FLAR2
FRND	FLARITH	FLAR1
FSIN	FLTRIG	FLAR1, FLAR2, FLCONV
FTAN	FLTRIG	FLAR1, FLAR2, FLCONV
FXAR1	FXARITH	None
FXAR2	FXARITH	None
FXCONV	FXARITH	None
FXEXP	FXEXP	FXAR1, FXAR2, FXCONV
FXIO	FXIO	FXAR1
FXSQRT	FXARITH	FXAR1, FXAR2

6 Summaries

ERROR CODES

This section lists the error codes returned in the global FPEXCEP. The codes apply to both floating point and fixed point unless otherwise stated.

Code	Meaning
1	Division by zero.
2	FFIX or FINT would give result greater than MAXINT or less than MININT.
3	Overflow.
4	Underflow (not applicable to fixed point).
5	Square root of negative number.
6	Arcsin or arccos of number greater in magnitude than one (not applicable to fixed point).
7	Illegal floating point operation (not applicable to fixed point).
8	Natural logarithm of zero or negative number.
9	FLIT has not read to the end of the string.

GLOBALS

This section lists, in alphabetical order, the globals declared by the header file FPHDR and their associated global numbers.

Name	Number	Name	Number
ASN	187	FMINUS	158
COS	186	FMULT	159
FABS	152	FNEG	160
FACOS	176	FPEXCEP	151
FASIN	175	FPI	183
FATAN	177	FPLUS	161
FCOMP	153	FRAD	182
FCOS	173	FRND	171
FDEG	181	FSGN	170
FDIV	154	FSIN	172
FE	184	FSQRT	180
FEXP	179	FTAN	174
FFIX	155	READFP	162
FFLOAT	156	SIN	185
FINT	164	SQR	188
FLIT	157	WRITEFP	163
FLN	178	WRITESG	165

MANIFESTS

The header file FPHDR declares one manifest:

FP.LEN = 3.

PROCEDURES

In this section the procedures are shown in **bold** and the parameters and returned values are given appropriate names.

Floating point and fixed point procedures

The procedures are grouped by function. Procedures marked with an asterisk ('*') are applicable to floating point only.

Arithmetic procedures

```
result.b   := FABS( veca, vecb)
comparison := FCOMP( veca, vecb)
result.c   := FDIV( veca, vecb, vecc)
result.c   := FMINUS( veca, vecb, vecc)
result.c   := FMULT( veca, vecb, vecc)
result.b   := FNEG( veca, vecb)
result.c   := FPLUS( veca, vecb, vecc)
sign       := FSGN( veca, vecb)
```

Input/output procedures

```
result.a := FLIT( string, veca)
result.a := READFP( veca)
WRITEFP( veca, width, places)
WRITESG( veca, width, digits) *
```

Conversion procedures

```
nearest.integer := FFIX( veca)
result.a        := FFLOAT( integer, veca)
integer.part    := FINT( veca)
```

Mathematical procedures

```
result.x      := FACOS( vec.cos.x, vec.x) *
result.x      := FASIN( vec.sin.x, vec.x) *
result.x      := FATAN( vec.tan.x, vec.x) *
result.cos.x  := FCOS( vec.x, vec.cos.x) *
result.deg    := FDEG( vec.rad, vec.deg) *
result.e      := FE( vec.e)
result.exp.x  := FEXP( vec.x, vec.exp.x)
result.ln.x   := FLN( vec.x, vec.ln.x)
result.pi     := FPI( vec.pi) *
result.rad    := FRAD( vec.deg, vec.rad) *
result.rnd    := FRND( vec.seed, vec.rnd) *
result.sin.x  := FSIN( vec.x, vec.sin.x) *
result.sqrt.x := FSQRT( vec.x, vec.sqrt.x)
result.tan.x  := FTAN( vec.x, vec.tan.x) *
```

Fast integer procedures

```
angle          := ASN( value)
cosine         := COS( angle)
sine           := SIN( angle)
sqrt.a.times.b := SQR( a, b)
```

Appendix

FLOATING POINT NUMBER FORMAT

Floating point numbers are held in three-word vectors. Each vector is treated as an array of six bytes, where byte 0 is the least significant byte of vector!0, byte 1 is the most significant byte of vector!0 etc.

The format is as follows:

Byte 0: Bit 7 is the sign bit (0 if number is positive, 1 if number is negative). Bits 0 to 6 are the exponent as an unsigned binary integer with a bias of 63 (i.e. 0 represents -63, 1 represents -62 etc).

Byte 1: Most significant eight bits of the mantissa (bit 7 is the most significant bit).

Byte 2: Next most significant eight bits of the mantissa.

etc

Byte 5: Least significant eight bits of the mantissa.

Exponents of 0 and 127 have special meanings described below. For exponents in the range 1 to 126 the interpretation of the number is:

(1+mantissa)*(2^(exponent-63))

where the most significant bit of the mantissa represents 0.5, the next most significant bit represents 0.25 etc.

For example the number 1 is represented by the six bytes (in hex):

3F 00 00 00 00 00

interpreted as (1+0)*(2^0)

and the number -0.75 is represented by:

BE 80 00 00 00 00

interpreted as -(1+0.5)*(2^(-1)).

Since words are stored with the least significant byte first, the last example coded as a TABLE would be:

TABLE #X80BE, #X0000, #X0000

Exponent of 0

Any number with an exponent of 0 is treated as 0 in calculations and is displayed as 0, whatever the values of the mantissa and the sign bit.

Exponent of 127

Exponents of 127 are used for the special illegal value and the values infinity and minus infinity. The illegal value has byte 1 equal to 255 (FF hex) and may be either positive or negative. Infinity has byte 1 equal to zero and the sign bit clear. Minus infinity has byte 1 equal to zero and the sign bit set. In all three cases bytes 2 to 5 are irrelevant. The interpretation of a number with an exponent of 127 and byte 1 neither 255 nor zero is undefined - some procedures may treat it as the illegal value, others may treat it as infinity or minus infinity.

FIXED POINT NUMBER FORMAT

Fixed point numbers are held in four-word vectors. Each vector is treated as an array of eight bytes, where byte 0 is the least significant byte of vector!0, byte 1 is the most significant byte of vector!0 etc.

The format is as follows:

Byte 0: The sign byte. A value of 0 denotes a positive number. A value of 0A hex denotes a negative number. The meaning of any other value is undefined and may be interpreted differently by different procedures.

Byte 1: The two most significant digits of the number in binary coded decimal i.e. bits 4 to 7 are interpreted as a binary number between 0 and 9 which is the most significant decimal digit of the number being represented and bits 0 to 3 are similarly interpreted as the next most significant decimal digit.

Byte 2: The two next most significant decimal digits.

etc

Byte 7: The two least significant decimal digits.

Note that both plus zero and minus zero can be represented, but they are treated identically by all the fixed point procedures.

As an example the number 87654321.9553 is represented by the following eight bytes (in hex):

00 00 87 65 43 21 95 53

and the negative of that number is represented by:

0A 00 87 65 43 21 95 53

Since words are stored with the least significant byte first, the last example, if coded as a TABLE, would be:

```
TABLE #X000A, #X6587, #X2143, #X5395
```

USEFUL FORMULAE

This section lists some formulae that may be useful in calculating mathematical functions for which procedures are not provided.

Formulae applicable to fast integer procedures

Tangent:
```
TAN(x) = SIN(x)/COS(x)
```

Arccosine:
```
ACS(x) = 15708 - ASN(x)
```

Arctangent:
```
ATN(x) = ASN(x/SQR(1+x*x))
```

General formulae

Secant:
```
SEC(x) = 1/COS(x)
```

Cosecant:
```
CSC(x) = 1/SIN(x)
```

Cotangent:
```
COT(x) = 1/TAN(x)
```

Arc secant:
```
ARCSEC(x) = ATN(SQR(x*x-1))+(SGN(x)-1)*pi/2
```

Arccosecant:
```
ARCCSC(x) = ATN(1/SQR(x*x-1))+(SGN(x)-1)*pi/2
```

```
Arccotangent:
ARCCOT(x) = -ATN(x)+pi/2

Hyperbolic sine:
SINH(x) = (EXP(x)-EXP(-x))/2

Hyperbolic cosine:
COSH(x) = (EXP(x)+EXP(-x))/2

Hyperbolic tangent:
TANH(x) = 1-2*EXP(-x)/(EXP(x)+EXP(-x))

Hyperbolic secant:
SECH(x) = 2/(EXP(x)+EXP(-x))

Hyperbolic cosecant:
CSCH(x) = 2/(EXP(x)-EXP(-x))

Hyperbolic cotangent:
COTH(x) = 1+2*EXP(-x)/(EXP(x)-EXP(-x))

Hyperbolic arcsine:
ARCSINH(x) = LN(x+SQR(x*x+1))

Hyperbolic arccosine:
ARCCOSH(x) = LN(x+SQR(x*x-1))

Hyperbolic arctangent:
ARCTANH(x) = LN((1+x)/(1-x))/2

Hyperbolic arcsecant:
ARCSECH(x) = LN((SQR(1-x*x)+1)/x)

Hyperbolic arccosecant:
ARCCSCH(x) = LN((SGN(x)*SQR(x*x+1)+1)/x)

Hyperbolic arccotangent:
ARCCOTH(x) = LN((x+1)/(x-1))/2
```

PRINTED IN ENGLAND BY POPPER & COMPANY LTD, WELWYN GARDEN CITY, HERTFORDSHIRE

BCPL Calculations Package

on the BBC Microcomputer

Acornsoft Limited, 4a Market Hill, Cambridge, CB2 3NJ, England

This BCPL system was developed by Richards Computer Products Limited, a company that specialises in providing BCPL systems. It was developed in collaboration with Dr Martin Richards of the Computing Laboratory, Cambridge University, who invented BCPL in 1967.

SNL10/B