This chapter describes a number of utility routines that you can use to perform mathematical and logical operations supported directly by the Macintosh Operating System. In particular, this chapter discusses how you can

- perform low-level logical manipulation of bits and bytes when using a compiler that does not directly support such manipulations
- save disk space by using simple compression and decompression routines
- obtain a pseudorandom number
- perform mathematical operations with two fixed-point data types supported directly by the Operating System
- convert numeric variables of different types

You need to read this chapter only if you need access to any of these features. With the exception of the mathematical operations and conversions, the routines this chapter describes are intended for programmers who occasionally need to access some of these features and do not require that the algorithms used to implement them be sophisticated. For example, if you are developing an advanced mathematical application, the pseudorandom number generator built into the Operating System might be too simplistic to fit your needs. Similarly, if you wish to access individual bits of memory in a time-critical loop, the Operating System routines that perform these operations are probably too slow to be practical.

You do not need any prior knowledge of the Operating System to read this chapter, which begins by describing the building blocks of memory in any operating system: bits, bytes, words, and long words. After subsequent discussions of the built-in compression and decompression routines provided by the Operating System, this chapter illustrates how you can use the Operating System's Mathematical and Logical Utilities. The chapter concludes with a reference to all mathematical and logical routines supported by the Operating System. If you are an experienced programmer, you might be able to skip directly to that section to determine which routine you need.

This chapter does not describe the numeric data types supported by the Standard Apple Numerics Environment (SANE) that the Operating System does not support directly. For more information on such data types, consult the *Apple Numerics Manual* and *Inside Macintosh: PowerPC Numerics*.

About the Mathematical and Logical Utilities

This section begins by introducing the building blocks of memory and then discusses some low-level routines the Mathematical and Logical Utilities provide, such as routines that compress data and generate pseudorandom numbers. Finally, the section concludes by introducing two fixed-point data types the Operating System supports.

Bits, Bytes, Words, and Long Words

This section describes the fundamental memory units used in all computer systems and discusses some of the operations that you can perform on them using the Mathematical and Logical Utilities. If you already know what bits, bytes, words, and long words are, you can skip this section.

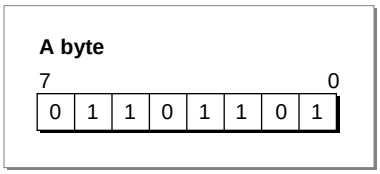
A ***bit*** is the atomic memory unit. Each bit can be set to one of two values. Often these values are called 0 and 1. A bit is said to be cleared when its value is 0 and set when its value is 1.

Eight bits form a single ***byte.*** The first bit in a byte is bit number 7, and the last bit is bit number 0. Bit number 7 is called the ***most significant bit*** or the ***high-order bit,*** and bit number 0 is the ***least significant bit*** or the ***low-order bit.*** A byte can thus store 2^8, or 256, different possible values. In Pascal, a byte is thus defined like this:

```
TYPE
   Byte = 0..255;
```

Figure 3-1 illustrates a byte set to the base-10 value 109.

Figure 3-1 A byte set to 109 ($6D)



The base-10 value 109 is equivalent to the binary value 01101101. This sequence of binary digits exactly corresponds to the status of each bit in the byte illustrated in Figure 3-1. A byte value is typically represented by two hexadecimal digits. The value in Figure 3-1, for example, is equivalent to $6D.

Sometimes it is useful to quickly convert between hexadecimal and binary number formats during debugging when examining the values of individual bits in a byte. Table 3-1 provides an easy way to do this on a digit-by-digit basis.

Table 3-1 Converting hexadecimal digits to binary values

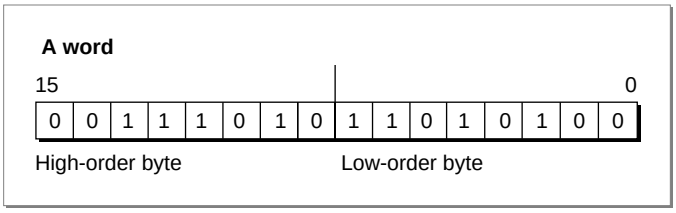
Hexadecimal	Binary
$0	0000
$1	0001
$2	0010
$3	0011
$4	0100
$5	0101
$6	0110
$7	0111
$8	1000
$9	1001
$A	1010
$B	1011
$C	1100
$D	1101
$E	1110
$F	1111

For example, the hexadecimal value $A8 is equivalent to the binary value 10101000 because the hexadecimal digit $A is equivalent to 1010 and the digit $8 is equivalent to 1000. You can use Table 3-1 to convert numbers in both directions.

While you can always think of a byte as a particular value from $00 to $FF, sometimes that value is irrelevant. For example, an application might use a byte simply as a way to store eight flag bits; in this case, the application cares about only individual bits within the byte and not the value of the byte as a whole. Also, bytes are often used to store signed values, in which case a byte can be considered equivalent to values from –$80 to +$7F. If you use a low-level debugger like MacsBug to examine individual bytes in memory, you should also be aware that different compilers might use bytes in different ways.

Two bytes form a ***word.*** A word is thus a 16-bit quantity and can be used to store 2^{16} (or 65,536) possible values. A ***word boundary*** is the memory location that divides two words. The first byte in a word is known as the high-order byte, and the second byte is known as the low-order byte. A pointer to a word points to the high-order byte. Figure 3-2 illustrates a word.

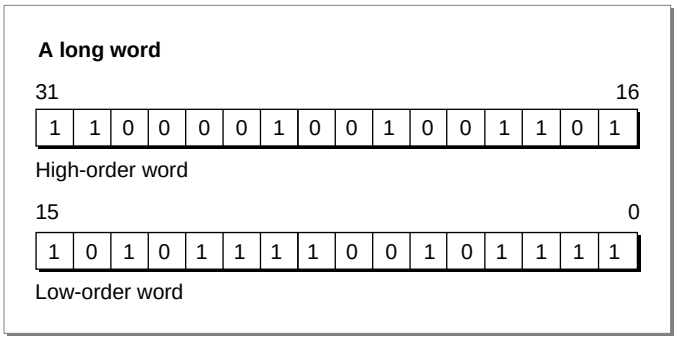
Figure 3-2 A word set to $3AD4



In Figure 3-2, the high-order byte is set to $3A. The low-order byte is set to $D4. The word thus has the value $3AD4.

Two words form a ***long word.*** A long word is thus a 32-bit quantity and can be used to store 2^{32} (or 4,294,967,296) values. A ***long-word boundary*** is the memory location that divides two long words. A long word consists of a high-order word and a low-order word, as illustrated in Figure 3-3.

Figure 3-3 A long word set to $C24DAF2F



In Figure 3-3, the high-order word is set to $C24D. The low-order word is set to $AF2F. The long word thus has the value $C24DAF2F.

Variables of type `Integer` are signed words, and variables of type `LongInt` are signed long words. On current versions of the Operating System, a memory address is stored using all 32 bits of a long word.

Typically, Macintosh compilers align all values on word boundaries (and in some cases on long-word boundaries). This means that when you declare a variable of type `Byte` in Pascal, the compiler is in fact likely to allocate 2 bytes of memory to store the byte; the extra byte is called a ***pad byte.*** In this case, when you attempt to test bits in a byte you have allocated, the compiler might test the corresponding bit in the wrong byte.

In Pascal, there are two easy ways to avoid this problem. One is to aggregate variables of type `Boolean` and of type `Byte` in a packed record. In this case, as long as the packed record's size is a number of bytes that is a multiple of 4, no pad bytes are added. The

second technique is, for variables in which you wish to test individual bits, to allocate 2 or 4 bytes for the variable (using a variable of type `Integer` or `LongInt`, respectively).

Bit Manipulation and Logical Operations

The Mathematical and Logical Utilities provide a number of routines that provide bit-level and byte-level control over memory, as described in "Performing Low-Level Manipulation of Memory" beginning on page 3-14. Given a pointer and offset, these routines can manipulate any specific bit in a stream of bits.

The `BitTst`, `BitSet`, and `BitClr` routines allow you to test and clear individual bits within a byte. These functions are introduced in "Testing and Manipulating Bits" on page 3-14.

Note
The `BitTst`, `BitSet`, and `BitClr` routines use a bit-numbering scheme that is opposite that of the MC680x0 microprocessor. This reversed bit-numbering scheme is described in the next section. ◆

The `BitAnd`, `BitOr`, `BitXor`, and `BitNot` functions allow you to perform logical operations on long words, and the `BitShift` function allows you to shift the bits in a long word to the right or to the left. These functions are introduced in "Performing Logical Operations on Long Words" on page 3-16.

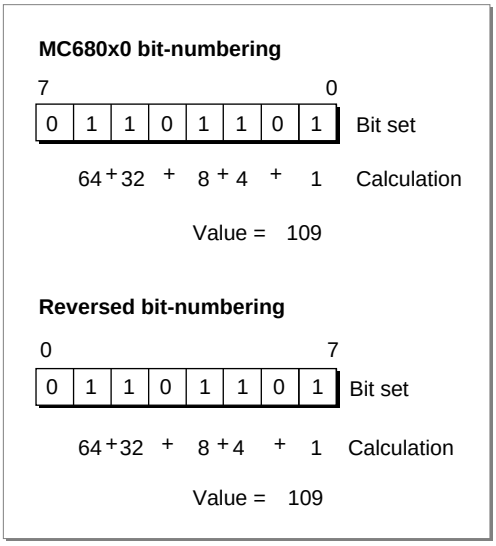
You might also need to extract one of a long word's words. The `HiWord` and `LoWord` functions allow you to do this and are described in "Extracting a Word From a Long Word" on page 3-18. Finally, you might need to set a group of bytes' values directly. The `StuffHex` procedure enables you to hardcode hexadecimal values to bytes anywhere in memory and is described in "Hardcoding Byte Values" on page 3-19.

Reversed Bit-Numbering

Three of the routines described in this chapter (the `BitTst`, `BitSet`, and `BitClr` routines) use a bit-numbering scheme that is opposite from that of the bit-numbering scheme used by the MC680x0 microprocessor.

The `BitTst`, `BitSet`, and `BitClr` routines count the bit numbers from left to right. That is, the most significant bit has the bit number 0. The MC680x0 bit number notation counts the bit numbers from right to left. (That is, the most significant bit has the biggest bit number.) Figure 3-1 illustrates these bit-numbering schemes.

Figure 3-4 Bit-numbering schemes



When using routines other than the `BitTst`, `BitSet`, and `BitClr` routines or if you are an assembly-language programmer, you should use the MC680x0 bit-numbering scheme.

To convert from MC680x0 bit notation to the scheme described in this section, subtract the MC680x0 bit number from the highest bit number. For example, to clear bit number 3 in a byte, you must clear bit number 4 (7–3 = 4).

Data Compression

The Mathematical and Logical Utilities include two procedures, `PackBits` and `UnpackBits`, that allow you to provide rudimentary data compression and decompression, respectively. The procedures are not powerful enough to provide effective compression for applications that primarily concern themselves with data compression. Also, if you are compressing sound, image, or video data, the Sound Manager (described in *Inside Macintosh: Sound*) and the Image Compression Manager (described in *Inside Macintosh: QuickTime*) provide far more effective compression algorithms.

You can use the `PackBits` and `UnpackBits` procedures to conserve memory both in RAM and on disk. However, because decompressing data is time consuming, typically you compress data using the `PackBits` procedure before saving a file or resource to disk and decompress data using the `UnpackBits` procedure after reading the data back from disk. Because the time required for compression and decompression using `PackBits` and `UnpackBits` is usually trivial compared to the time it takes to access a typical hard disk, the routines provide a simple, low-overhead way for an application to minimize the size of its data files.

The `PackBits` procedure is effective when an uncompressed buffer of data is likely to have many consecutive bytes containing the same value. For example, some applications use data structures that include fields that the application reserves for future use. These fields are typically all set to 0. The `PackBits` procedure senses that there is a long string of consecutive bytes containing the same value and compresses the string of bytes by using 1 byte to indicate that the subsequent compressed byte represents a number of consecutive uncompressed bytes.

`PackBits` was originally intended as an easy way to compress black-and-white image data, such as MacPaint documents. However, because each pixel of a color picture is typically represented by multiple bytes of data, `PackBits` is unlikely to provide effective compression for such pictures.

If there is no reason to think that your data format might contain long strings of consecutive bytes, then the `PackBits` procedure is probably not useful and might even increase the size of your files. The `PackBits` procedure packs data 127 bytes at a time. If within the 127 bytes there is no series of 3 consecutive bytes containing the same value, then there are no gains to be made from compression. In this case, the `PackBits` procedure must use an initial byte to specify that the 127 subsequent bytes contain uncompressed data. You can compute the worst-case performance of `PackBits` (that is, the maximum number of output bytes) by using the following formula:

```
maxDstBytes := srcBytes + (srcBytes+126) DIV 127;
```

where `maxDstBytes` stands for the maximum number of destination bytes and `srcBytes` stands for the number of bytes in the uncompressed source data.

You can, if desired, pack a buffer of data, and then pack the packed buffer again. However, packing data twice not only is slower than packing data once, but also is likely to result in a larger output buffer than just packing data once. If your application does pack data twice, it should unpack the data twice.

Note
In current versions of system software, you can request that `PackBits` pack up to 32,767 bytes. The `PackBits` procedure then processes the input buffer in 127-byte chunks. In versions of system software prior to version 6.0.2, however, you should pass to `PackBits` only buffers up to 127 bytes in length. ◆

Pseudorandom Number Generation

Because digital computers continuously execute instructions, it is impossible for a computer to select a truly random number. To force the computer to output a number, the programmer must create an algorithm, but because algorithms always execute in the same way, the numbers an algorithm produces cannot be truly random. Random numbers are often necessary in software applications, however. For example, an entertainment software application might need to ensure that the user is not faced

with the exact same game every time. Or a spreadsheet application might offer a randomization function for business users attempting to simulate various possible scenarios.

To get around the impossibility of producing truly random numbers, computer scientists rely on pseudorandom number generation algorithms. These are complex numeric algorithms used to produce a series of numbers. All such series eventually repeat, but typically not until the pseudorandom number generation algorithm has been executed millions or even billions of times. Because the series is generated by an algorithm, it is possible to discern a pattern; given the first few numbers of a series, a clever user might be able to guess the next number. Typically, however, these algorithms are complicated enough to make the numbers appear random, at least to the casual observer.

Of course, because pseudorandom number generation algorithms are algorithms, they produce the same series of numbers every time. However, you can seed the pseudorandom number generator to force it to start somewhere in the middle of the series. By seeding the generator to a constantly changing variable when your application starts up, your application can produce different results each time. The value typically used to seed the pseudo-random number generator is the current date and time. Of course, time isn't random—it moves forward at a constant linear rate—but in the absence of a stopped system clock, the user will never launch your application at the same time twice, so you can be confident that your application will produce different results each time it is executed.

The Macintosh Operating System's pseudorandom number generation algorithm is accessible through the `Random` function. The `Random` function returns a pseudorandom integer from –32767 to 32767. The value that the `Random` function produces depends on the `randSeed` global variable. The `Random` function changes `randSeed` while generating a pseudorandom number, thus enabling a subsequent call to `Random` to produce the next number in the series. You only need to seed the global variable once, at the start of your program.

The pseudorandom number generation algorithm is designed so that as the number of times `Random` is executed approaches infinity, the percentage difference in the number of times any two integers in the range –32767 to 32767 are produced approaches 0. Thus, the pseudorandom number generator is said to produce pseudo-random numbers that are uniformly distributed in the range –32767 to 32767.

This chapter does not describe the algorithm that `Random` uses to generate pseudorandom numbers. While the algorithm is sufficiently complex for most applications, applications that perform mathematical or statistical analysis might require a better pseudo-random number generator. Consult the computer science literature for information on sophisticated pseudorandom number generation algorithms.

Fixed-Point Data Types

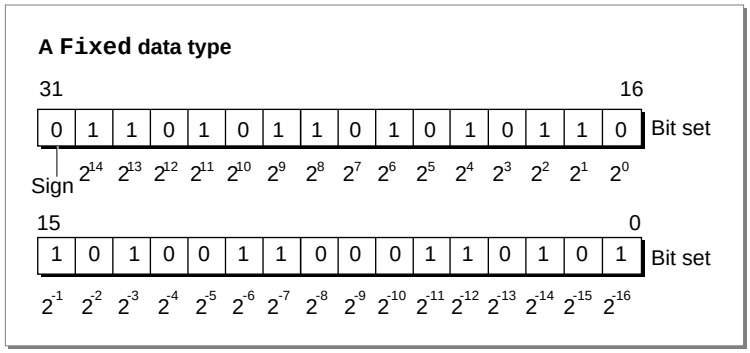
The Operating System supports two fixed-point data types, that is, numeric types that consist of integral and fractional components. Depending on the type of information you are representing with a fixed-point data type, these might be better suited for your needs than the types `Integer`, `LongInt`, and the many floating-point types supported by the Standard Apple Numerics Environment.

A variable of type `Fixed` is defined like this:

```
TYPE
   Fixed = LongInt;
```

A variable of type `Fixed` is a 32-bit signed quantity containing an integer part in the high-order word and a fractional part in the low-order word. Figure 3-5 illustrates the format for `Fixed`.

Figure 3-5 The `Fixed` data type



The high-order word consists of the integral component of the fixed-point number, and the low-order word consists of the fractional component of the fixed-point number. Each bit, other than the most significant bit, represents a power of 2, as indicated in Figure 3-5.

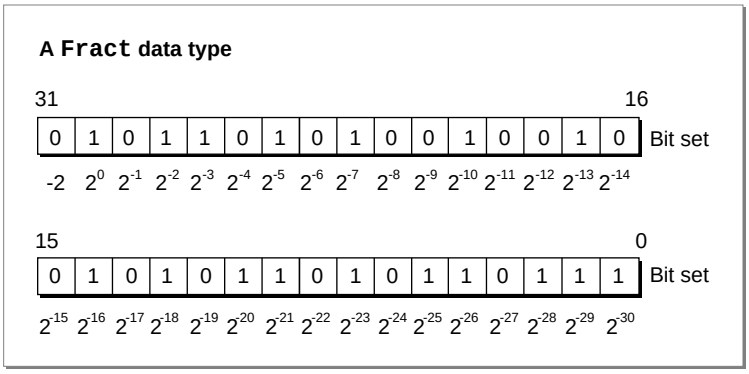
Negative numbers of type `Fixed` are the two's complement; that is, the negative numbers are formed by treating the fixed-point number as a long integer, inverting each bit, and adding 1 to the least significant bit.

The `Fract` data type is useful for allowing accurate representation of small numbers, that is, numbers between –2 and 2. It is defined just like `Fixed`:

```
TYPE
   Fract = LongInt;
```

Figure 3-6 illustrates the format for `Fract`.

Figure 3-6 The `Fract` data type



Like a `Fixed` number, a `Fract` number is a 32-bit quantity, but its implicit binary point is to the right of bit 30 of the number; that is, a `Fract` number has 2 integer bits and 30 fraction bits. As with the type `Fixed`, a number is negated by taking its two's complement. Thus, `Fract` values range between –2 and $2 - (2^{-30})$, inclusive.

All routines that operate on fixed-point numbers handle boundary cases uniformly. Results are rounded by adding half a unit in magnitude in the last place of the stored precision and then chopping toward zero. Overflows are set to the maximum representable value with the correct sign ($80000000 for negative results and $7FFFFFFF for positive results). Division by zero results in $8000000 if the numerator is negative and $7FFFFFFF otherwise; thus, the special case 0/0 yields $7FFFFFFF.

Angle-Slope Conversion

The Mathematical and Logical Utilities provide two functions for applications that need to draw lines at particular angles. For example, a mathematical plotting application might need to draw a 30-degree line. The `SlopeFromAngle` and `AngleFromSlope` functions provide simple conversion between slope and angle values. Slopes and angles are defined in such a way as to be convenient to a computer programmer rather than correspond to the conventional mathematical interpretation.

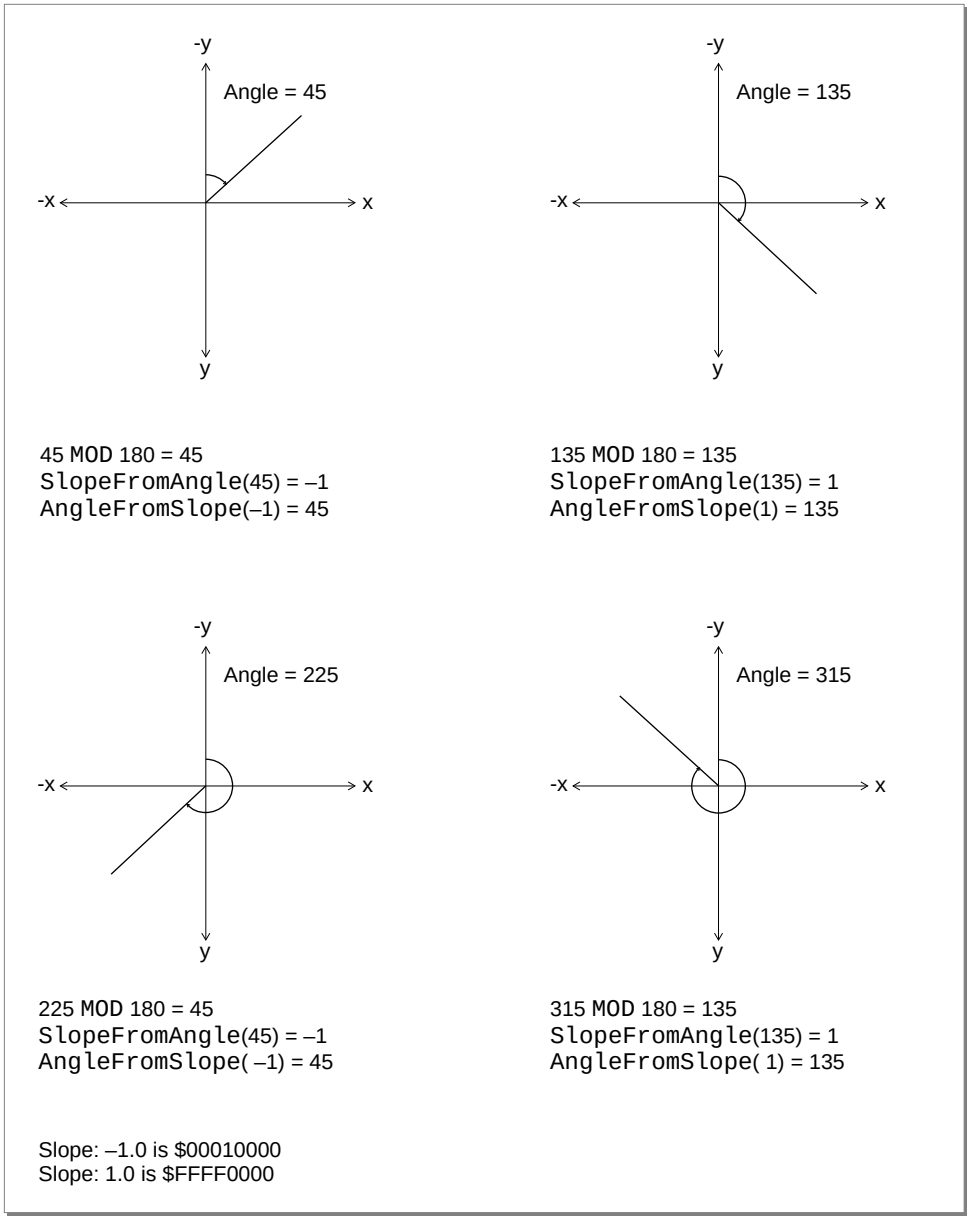
Note
You should not rely on the `SlopeFromAngle` and `AngleFromSlope` functions to produce values that will allow you to draw lines at a precise angle on the screen. The functions do not take into account the size of pixels on a screen. If pixels on a screen are not perfect squares, a 30-degree angle might appear to be a different angle to the user. ◆

Since QuickDraw and other computer imaging schemes typically invert the y-axis (making positive down and negative up), the angle-slope conversion routines use this convention as well. Angles are measured clockwise relative to the negative y-axis (that is, relative to 12 o'clock), and are taken `MOD` 180, so that a 270-degree angle is considered to be equivalent to a 90-degree angle.

Slopes are defined as $\Delta x/\Delta y$, the horizontal change divided by the vertical change for any two points on a line with the slope. Note that mathematicians typically measure slopes $\Delta y/\Delta x$. The convention of angle-slope conversion is convenient for applications that plot a number of lines in a graph one horizontal line at a time.

Figure 3-7 shows some equivalencies between angle and slope values for the angle-slope conversion routines.

Figure 3-7 Some slope and line equivalencies using the conventions of the angle-slope conversion routines



The `AngleFromSlope` function is useful primarily only when speed is more important than accuracy because the function might return an angle off by as much as 1 degree from the actual angle. The function returns values between 1 and 180 (inclusive), and thus never returns an angle value between 0 and 1 degrees. If your application is likely to need precise differentiation in angles, you should probably develop alternative routines to handle angle-slope conversions.

`SlopeFromAngle(0)` is 0, and `AngleFromSlope(0)` is 180. For all `x` except for 0, however, `AngleFromSlope(SlopeFromAngle(x)) = x` is true. But the reverse, `SlopeFromAngle(AngleFromSlope(x)) = x` is not necessarily true.

Using the Mathematical and Logical Utilities

This section describes how you can take advantage of the Mathematical and Logical Utilities supported by the Operating System, it describes how you can

- test and set individual bits, perform logical operations on long words, divide a long word into its high word and low word, and set memory values directly.
- use the `PackBits` and `UnpackBits` procedures to compress and decompress data.
- seed the pseudo-random number generator and obtain random integers or long integers within a given range.
- perform simple calculations involving fixed-point numbers and convert fixed-point numbers to other numeric types.

Performing Low-Level Manipulation of Memory

The Mathematical and Logical Utilities provide several routines to perform bit-level and byte-level manipulation of memory. These routines are provided primarily for Pascal programmers. C and assembly-language programmers can use these routines also; however, in general it is easier and more efficient to achieve the same effects as these routines by using built-in C or assembly constructs.

Testing and Manipulating Bits

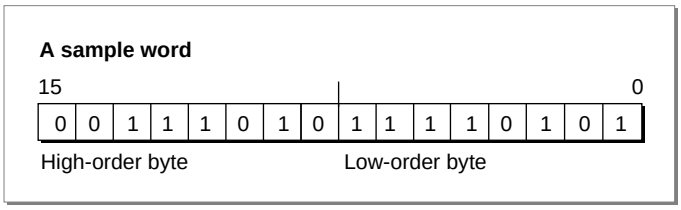
The `BitTst` function lets you test whether a given bit is set. The function requires that you specify a bit through an offset from a pointer. Listing 3-1 is an example of an application-defined function that tests a specified bit.

Listing 3-1 Testing bits

```
FUNCTION MyTestBit (bytePtr: Ptr; bitNum: LongInt): Boolean;
BEGIN
   MyTestBit := BitTst(bytePtr, bitNum);
END;
```

The `bytePtr` parameter specifies a pointer to a byte in memory. The `bitNum` parameter specifies the number of the bit to be tested as an offset from `bytePtr`. For example, you can use the application-defined function `MyTestBit` to test specific bits of the word specified in Figure 3-8.

Figure 3-8 A sample word (in MC680x0 notation)



Using the word in Figure 3-8, the call `BitTst(myPtr, 0)` returns `FALSE` because bit number 0 in the first byte is not set. But the call `BitTst(myPtr, 11)` returns `TRUE` because bit number 3 in the second byte is set.

When using the `BitTst` function, be sure to specify bits as positive offsets from the high-order bit rather than using the normal MC680x0 notation (see "Reversed Bit-Numbering" on page 3-7). Listing 3-2 illustrates a use of the `BitTst` function in conjunction with a bit traditionally identified with MC680x0 notation.

Listing 3-2 Determining whether a handle is purgeable using the `BitTst` function

```
FUNCTION MyHandleIsPurgeable (myHandle: Handle): Boolean;
CONST
   kMyBitNum68000 = 6;
VAR
   propertiesByte: SignedByte;
BEGIN
   propertiesByte := HGetState(myHandle);
   MyHandleIsPurgeable := BitTst(@propertiesByte,
                                 7 - kMyBitNum68000);
END;
```

The `MyHandleIsPurgeable` function defined in Listing 3-2 determines whether a handle references a relocatable block by examining the properties byte for that handle. The purgeable bit is, in MC680x0 notation, bit number 6 of the properties byte; because `BitTst` uses reverse numbering, so bit number 7 – 6 = 1 is tested.

The `BitSet` and `BitClr` procedures require that you specify bits using the same scheme as with the `BitTst` procedure (see "Reversed Bit-Numbering" on page 3-7). The `BitSet` procedure sets a bit (that is, sets its value to 1), while `BitClr` clears a bit

(that is, sets its value to 0). For example, if you issue the following two calls to the `BitSet` procedure

```
BitSet(bytePtr, 5);
BitClr(bytePtr, 7);
```

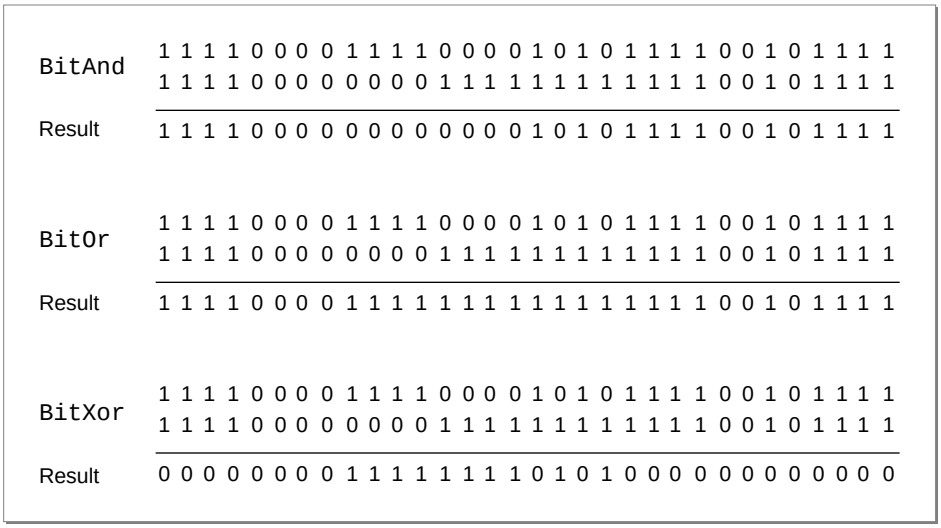
bit 5 (using the reversed bit-numbering scheme) of the byte in memory pointed to by the `bytePtr` parameter is set to 1, and bit 7 (using reversed bit-numbering) of the same byte is cleared.

Note
In C, you can test bits by using the `&` operator. You can set and clear bits by using the `|=` and `&=` operators, respectively. In all three cases, one operand should be the byte (or word or long word you wish to manipulate), and the other should be a value in which only the relevant bit is set or cleared. Many Pascal compilers also support built-in operations that accomplish these tasks efficiently. Note that C uses the MC680x0 bit-numbering scheme (normal bit-numbering). ◆

Performing Logical Operations on Long Words

The Macintosh Operating System provides routines that allow you to perform basic bitwise logical operations, including the `AND`, `OR`, and `XOR` operations on long words. Each of the functions takes two long integers as parameters and returns another long integer. You can use these functions on other 32-bit data types, as long as you cast values to `LongInt` as required by your compiler. The functions that perform the `AND`, `OR`, and `XOR` operations are `BitAnd`, `BitOr`, and `BitXor` respectively. Figure 3-9 illustrates these functions.

Figure 3-9 The `BitAnd`, `BitOr`, and `BitXor` functions



As shown in Figure 3-9, the `BitAnd` function returns a long word in which each bit is set if and only if the corresponding bit is set in both long words passed in. The `BitOr` function returns a long word in which each bit is set if and only if the corresponding bit is set in either long word passed in. The `BitXor` function returns a long word in which each bit is set if and only if one but not both of the corresponding bits in the long words passed in is set.

Note
In C, you can achieve the same effects as the `BitAnd`, `BitOr`, and `BitXor` functions by using the `&`, `|`, and `^` operators, respectively, in conjunction with the `=` assignment operator. Many Pascal compilers also support built-in operations that accomplish these tasks more efficiently. ◆

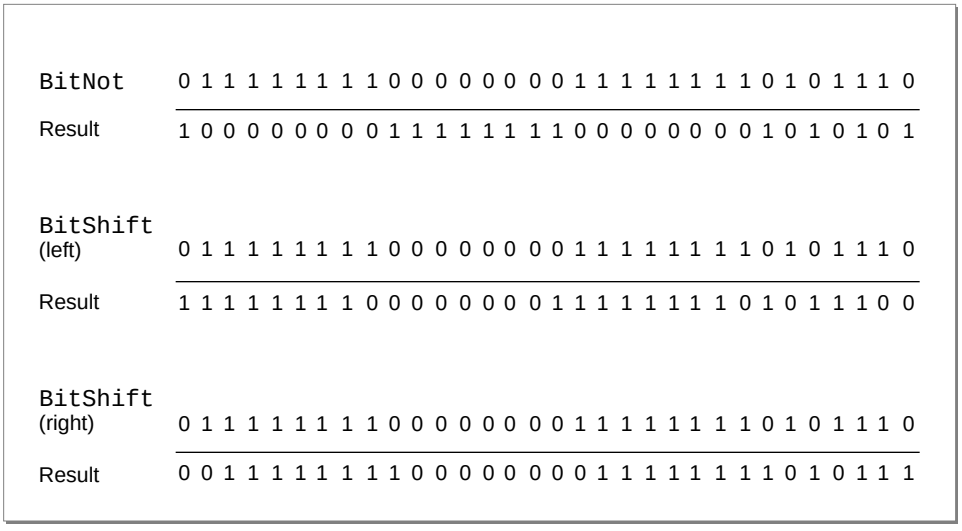
A common use of the `BitAnd` function is to mask out certain bytes within a long word (that is, clear all bits in those bytes). For example, to mask out the second byte of a long word stored in a variable `value`, you could write the following code:

```
value := BitAnd(value, $FF00FFFF);
```

The Macintosh Operating System also offers two bit-manipulation routines that simulate unary operators, the `BitNot` and the `BitShift` functions, which perform the `NOT` operation and bit-shifting, respectively. You specify the long integer on which to perform the operation as a parameter to the `BitNot` and `BitShift` functions. In addition, you specify how to shift the bits as a parameter to the `BitShift` function.

Figure 3-10 illustrates `BitNot` and `BitShift`.

Figure 3-10 The `BitNot` and `BitShift` functions



As shown in Figure 3-10, the `BitNot` function returns a long word in which each bit is set if and only if the corresponding bit in the long word passed in is not set. The `BitShift` function shifts bits—to the left if the `count` parameter is greater than 0 and to the right if the `count` parameter is less than 0. (Shifting to the left means shifting towards the high-order bit.) When shifting `count` bits to the left, the `count` low-order bits are set to 0; when shifting `count` bits to the right, the `count` high-order bits are set to 0.

Note
In C, you can achieve the same effect as the `BitNot` function more efficiently by using the `^` operator on the value whose bits are to be inverted and the value $FFFFFFFF. You can achieve the same effect as the `BitShift` function more efficiently by using the `>>` operator for shifting to the right and the `<<` operator for shifting to the left. Many Pascal compilers support built-in operations that accomplish these tasks efficiently. ◆

Extracting a Word From a Long Word

Often a long word stored as a variable of type `LongInt` is used to hold two different pieces of information in its two different words. For example, when a disk-inserted event occurs, the `message` field of the event record contains the drive number in the low-order word and a result code in the high-order word. To access these two types of information, you can use the `HiWord` and `LoWord` functions. For example:

```
VAR
   x: LongInt;
   high, low: Integer;
   high := HiWord(x);
   low := LoWord(x);
```

The `HiWord` function returns the high-order word of the long word passed in, and the `LoWord` function returns the low-order word of the long word passed in. You can use these functions with types other than `LongInt` and `Integer`, as long as they are 4 bytes and 2 bytes, respectively, and, if you are using Pascal, you cast the quantities to the correct types.

The Operating System does not provide any routines that allow you to set the high-order or low-order words of a long integer. It might seem that you could set the low-order word by calling the `BitAnd` function with the original long integer and the low-order word as parameters, and set the high-order word by calling `BitAnd` with the original long integer and the high-order word shifted left 16 bytes as parameters. The problem with this approach is that when you pass an integer variable to `BitAnd`, the compiler automatically casts the variable to a long integer. But for both integers and long integers, it is the leftmost byte that indicates the sign of the number. So when a negative integer is cast to a long integer, the low-order word of the long integer is not equal to the original integer.

However, you can use the Memory Manager's `BlockMove` procedure to directly copy the bytes of a word to the high-order or low-order word of a long word. See *Inside Macintosh: Memory* for more information. Or, if you wish to set both the high-order word and the low-order word of a long integer at once, you can define the following type:

```
TYPE MyLongWordType =
PACKED RECORD
   myHiWord:      Integer;       {high-order word}
   myLoWord:      Integer;       {low-order word}
END;
```

Then you can define a variable of this type and set the high-word and low-word fields. By casting a long integer to `MyLongWordType`, you could also extract a word from a long word more efficiently than you can using the `HiWord` and `LoWord` functions.

Hardcoding Byte Values

Occasionally, you might need to set a group of bytes in memory to specific hexadecimal values. For example, suppose your application uses a data structure with a 16-byte flags field and you wish to initialize each of the bytes in the flags field to particular values. While there are a number of ways that you might do this, the `StuffHex` procedure provides a simple, though usually inefficient, option.

You provide a pointer to any data structure in memory, and a string of hexadecimal digits as parameters to the `StuffHex` procedure. For example:

```
StuffHex(@x, 'D34E0F29');
```

Of course, it would in this case be just as easy—and more efficient—to write the following code:

```
x := $D34E0F29;
```

The `StuffHex` procedure is perhaps most useful when you wish to assign a large or odd number of bytes or set the values of particular bytes within a variable. For example, to set the low-order word of a long integer `x` to $64B5, you could use the following code:

```
StuffHex(Ptr(ORD4(@x) + 2), '64B5');
```

You could use this code rather than use the techniques described in the previous section, "Extracting a Word From a Long Word."

Note that `Ptr` and `ORD4` are used here simply to satisfy Pascal type-casting rules.

The `StuffHex` procedure might also be useful if you are developing a calculator or other application that allows users to enter hexadecimal values directly.

Compressing Data

The `PackBits` and `UnpackBits` procedures, introduced in "Data Compression" on page 3-8, allow you to compress (or decompress) data stored in RAM. Typically, you use `PackBits` before writing data to disk and `UnpackBits` immediately after writing data from disk.

Both procedures require that you pass in the `srcPtr` and `dstPtr` parameters values that point to the beginning of the source buffer and the destination buffer, respectively. The `PackBits` procedure compresses the data in the source buffer and stores the result in the destination buffer; the `UnpackBits` procedure decompresses the data in the source buffer and stores the result in the destination buffer. You must also pass to the `PackBits` procedure and the `UnpackBits` procedure a value that specifies the size of the original, uncompressed data. Because you must pass this information to `UnpackBits`, you typically use these procedures only to compress a data structure with a fixed size, so that this size can be passed as a parameter to `PackBits`.

Your application is responsible for allocating memory for both the source and the destination buffers. When `PackBits` and `UnpackBits` complete operation, the `srcPtr` and `dstPtr` parameter are incremented so that `srcPtr` points to the memory immediately following the source bytes, and `dstPtr` points to the data immediately following the destination bytes. This feature was originally designed to allow you to pack large buffers of data at once in chunks, although `PackBits` can automatically chunk large data buffers in versions of system software 6.0.2 and later. In any case, your application must store copies of `srcPtr` and `dstPtr` to access the start of the source or destination buffer after calling `PackBits` or `UnpackBits`.

One use of the compression routines might be to compress resources in your application's resource fork. Many types of resources can be made significantly smaller by compression. Listing 3-3 shows how you can pack data stored in a handle to a specified resource.

Listing 3-3 Packing data to a resource

```
PROCEDURE MyAddPackedResource (srcData: Handle; theType: ResType;
                               theID: Integer; name: Str255);
VAR
   srcBytes:         Integer;               {bytes of unpacked data}
   maxDstBytes:      LongInt;               {maximum length of packed data}
   dstData:          Handle;                {packed data}
   srcPtr:           Ptr;                   {pointer to unpacked data}
   dstPtr:           Ptr;                   {pointer to packed data}
   srcProperties:    SignedByte;            {properties of source handle}
BEGIN
   srcBytes := GetHandleSize(srcData);      {find size of source}
                                            {calculate maximum possible }
                                            { size of packed data}
```

```
   maxDstBytes := srcBytes + (srcBytes + 126) DIV 127;
   dstData := NewHandle(maxDstBytes + 2);    {allocate memory for source, }
                                             { plus length info}
   IF dstData <> NIL THEN                    {check for NIL handle}
   BEGIN
      BlockMove(@srcBytes, dstData^, 2);     {copy source into buffer}
      srcPtr := srcData^;                    {copy source pointer}
      dstPtr := Ptr(ORD4(dstData^) + 2);     {copy destination pointer}
      PackBits(srcPtr, dstPtr, srcBytes);    {pack source to destination}
                                             {shrink destination data}
      SetHandleSize(dstData, ORD4(dstPtr) - ORD4(dstData^));
      srcProperties := HGetState(srcData);   {get source handle properties}
      IF BitTst(@srcProperties, 2) THEN      {is source a real resource?}
         RemoveResource(srcData);            {remove current resource}
                                             {add to resource file}
      AddResource(dstData, theType, theID, name);
      WriteResource(dstData);                {write resource data}
      DetachResource(dstData);               {detach from resource map}
      DisposeHandle(dstData);                {dispose of destination data}
   END;
END;
```

The `MyAddPackedResource` procedure declared in Listing 3-3 initially allocates a destination buffer to hold compressed data that is big enough to hold the compressed data in a worst-case scenario, plus 2 bytes to store information at the beginning of the resource about the size of the source data. Because `PackBits` does not move memory, the handle storing the destination buffer does not need to be locked. However, to prevent the `PackBits` procedure from changing the value of a master pointer, you should only pass copies of the dereferenced handle to the procedure. After `PackBits` returns, `MyAddPackedResource` determines how much memory the compressed data takes up by computing how much the `dstPtr` variable has changed. `MyAddPackedResource` then resizes the handle containing the compressed data to the appropriate size. Finally, `MyAddPackedResource` writes the new resource, after first removing the existing resource if the source handle is a handle to a resource. For more information on resources, see *Inside Macintosh: More Macintosh Toolbox*.

Having used the `MyAddPackedResource` procedure to compress resource data, your application needs to be able read the resource and decompress it using the `UnpackBits` procedure. Listing 3-4 shows how you might accomplish this.

Listing 3-4 Decompressing data from a packed resource

```
FUNCTION MyGetPackedResource (theType: ResType; theID: Integer): Handle;
VAR
   srcData:          Handle;                  {handle to packed data}
```

```
   dstData:          Handle;                  {handle to unpacked data}
   srcPtr:           Ptr;                     {pointer to packed data}
   dstPtr:           Ptr;                     {pointer to unpacked data}
   dstBytes:         Integer;                 {number of unpacked bytes}
BEGIN
   srcData := GetResource(theType, theID);    {get the resource}
   BlockMove(srcData^, @dstBytes, 2);         {read number of bytes of }
                                              { unpacked data}
   dstData := NewHandle(dstBytes);            {allocate memory for }
                                              { unpacked data}
   IF dstData <> NIL THEN
   BEGIN
      srcPtr := Ptr(ORD4(srcData^) + 2);      {copy source pointer}
      dstPtr := dstData^;                     {copy destination pointer}
      UnpackBits(srcPtr, dstPtr, dstBytes);   {unpack source to }
                                              { destination}
   END;
   IF srcData <> NIL THEN                     {if there was a resource}
   BEGIN
      DetachResource(srcData);                {detach from resource map}
      DisposeHandle(srcData);                 {dispose the resource}
   END;
   MyGetPackedResource := dstData;            {return destination handle}
END;
```

The `MyGetPackedResource` function reads in a resource that has previously been packed, determines the size of the unpacked data by copying the first 2 bytes of the resource data, and allocates a relocatable block of this size. The remainder of the data is unpacked using the `UnpackBits` procedure, and the original packed resource data is disposed of.

Obtaining Pseudorandom Numbers

The `Random` function makes it easy to obtain pseudorandom numbers. Before you use `Random`, however, you should seed the pseudo-random number generator. Listing 3-5 shows a common technique for doing this.

Listing 3-5 Seeding the pseudo-random number generator

```
PROCEDURE MySeedGenerator;
BEGIN
   GetDateTime(randSeed);
END;
```

The `MySeedGenerator` procedure defined in Listing 3-5 simply uses the Date and Time Utilities' `GetDateTime` procedure to copy the number of seconds since midnight, January 1, 1904, to the global variable `randSeed`. You might use some other volatile long-word value—such as the mouse location—to seed the pseudo-random number generator, or you might even take a word from one source and a word from another. However, just using `GetDateTime` is sufficient for most applications.

Sometimes you wish to obtain a pseudo-random integer from a large range of integers; for example, you might need a pseudo-random integer in the range of –20,000 to 20,000. Listing 3-6 shows how you might do this.

Listing 3-6 A simple way of obtaining a large random integer from a range of pseudo-random numbers

```
FUNCTION MyRandomLargeRange (min, max: Integer): Integer;
VAR
   randInt:       Integer;
BEGIN
   REPEAT
      randInt := Random
   UNTIL (randInt >= min) AND (randInt <= max);
   MyRandomLargeRange := randInt;
END;
```

The `MyRandomLargeRange` function defined in Listing 3-6 simply calls the `Random` function until it returns an acceptable value. This approach is efficient when you need a random integer from a range of integers that is wide, though not quite as wide as the range the `Random` function returns by default. However, if you need a random number from a small range—for example, a random number from 1 to 10—the `MyRandomLargeRange` function is inefficient. Listing 3-7 shows an alternative approach.

Listing 3-7 Obtaining a pseudo random integer from a small range of numbers

```
FUNCTION MyRandomRange (min, max: Integer): Integer;
CONST
   kMinRand = -32767.0;
   kMaxRand = 32767.0;
VAR
   myRand:     Integer;
   x:          Real;          {Random scaled to [0..1]}
BEGIN
   {find random number, and scale it to [0.0..1.0]}
   x := (Random - kMinRand) / (kMaxRand + 1.0 - kMinRand);
```

```
   {scale x to [min, max + 1.0], truncate, and return result}
   MyRandomRange := TRUNC(x * (max + 1.0 - min) + min);
END;
```

The `MyRandomRange` function defined in Listing 3-7 first scales the integral value returned by the `Random` function to a floating-point value from 0 up to, but not including, 1. The function then scales the result to a real number greater than or equal to `min` but less than `max + 1`. By truncating extra decimal places, the correct result is achieved. Note that to force the compiler to perform floating-point calculations, all constants in the function are expressed as real numbers rather than as integers.

Sometimes an application might require a pseudo-random long integer. Listing 3-8 shows how you can do this.

Listing 3-8 Obtaining a pseudo-random long integer

```
FUNCTION MyRandomLongInt: LongInt;
TYPE
   MyLongWordType = PACKED RECORD
      myHiWord:   Integer;            {high-order word}
      myLoWord:   Integer;            {low-order word}
   END;
VAR
   myLongWord:    MyLongWordType;     {random long word}
BEGIN
   {obtain random high-order word}
   myLongWord.myHiWord := Random;
   {obtain random low-order word}
   myLongWord.myLoWord := Random;
   {cast and return result}
   MyRandomLongInt := LongInt(myLongWord);
END;
```

The `MyRandomLongInt` function defined in Listing 3-8 uses a technique discussed in "Extracting a Word From a Long Word" on page 3-18 to stuff a pseudo-random number in the high-order word of a long integer and another pseudo-random number in the low-order word of the long integer. If you need to obtain a long integer within a specified range, you can define routines analogous to Listing 3-6 and Listing 3-7 but use the `MyRandomLongInt` function in place of the `Random` function.

Using Fixed-Point Data Types

Most high-level language compilers include built-in support for the `Fixed` and `Fract` data types so that you can perform regular mathematical operations with fixed-point variables. Also, the algorithms for performing addition and subtraction on `Fixed` and

`Fract` variables are the same as the algorithms for performing such operations on variables of type `LongInt`.

The Operating System, however, includes several routines that allow you to convert `Fixed` and `Fract` variables to other formats, including SANE's `Extended` data type, and allow you to perform some simple operations on `Fixed` and `Fract` variables. If you need more sophisticated numeric functions, consult the *Apple Numerics Manual*.

To perform multiplication and division of fixed-point numbers, you can use the `FixMul`, `FixDiv`, `FracMul`, and `FracDiv` functions, which allow you to multiply `Fixed` point numbers with each other or with other long integers.

You can multiply and divide 32-bit quantities of different types using these functions. The format of the result in this case depends on the particular function being used. See descriptions of the individual functions in "Multiplying and Dividing Fixed-Point Numbers" beginning on page 3-38 for more information.

Using the `FracSqrt`, `FracCos`, `FracSin`, and `FixATan2` functions, you can perform a few special arithmetic operations involving variables of type `Fixed` and `Fract`.

The `FracSqrt` function allows you to obtain the square root of a variable of type `Fract`, interpreting bit 0 as having weight 2 rather than –2. The `FracCos` and `FracSin` provide support for the trigonometric cosine and sine functions. The `FixATan2` function provides support for the arctangent function. The arguments to all of these functions should be expressed in radians, not in degrees.

Note
To provide fast trigonometric approximations, these trigonometric functions use values of π correct only to 4 decimal places. You should thus use alternative SANE routines when you require better precision. ◆

To convert among 32-bit numeric types, you can use the `Long2Fix`, `Fix2Long`, `Fix2Frac`, and `Frac2Fix` functions.

Each of the functions returns its parameter converted into the appropriate format.

You can also convert fixed-point values to and from the SANE `Extended` floating-point type using the `Fix2X`, `X2Fix`, `Frac2X`, and `X2Frac` functions.

Two additional functions, `FixRatio` and `FixRound`, allow you to perform special conversions on variables of type `Fixed`.

The `FixRatio` function returns the fixed-point quotient of the `numer` and `denom` parameters. The `FixRound` function rounds a variable of type `Fixed` to the nearest integer. If the value is halfway between two integers (0.5), it is rounded to the integer with the higher absolute value. To round a negative fixed-point number, negate it, round it, and then negate it again.

Note
To convert a variable of type `Fixed` to a variable of type `Integer` simply use the `HiWord` function to extract the integral component of the fixed-point number. ◆

The Operating System also provides the `LongMul` procedure that allows you to multiple two 32-bit quantities and obtain a 64-bit quantity.

Table 3-2 summaries the routines that perform operations on the `Fixed` and `Fract` data types.

Table 3-2 Routines for fixed-point data types

Routine	Description
`FixMul`	Multiply a variable of type `Fixed` with another variable of type `Fixed` or with a variable of type `Fract` or `LongInt`
`FixDiv`	Divide two variables of the same type (`Fixed`, `Fract`, or `LongInt`) or divide a `LongInt` or `Fract` number by a `Fixed` number
`FracMul`	Multiply a variable of type `Fract` with another variable of type `Fract` or with a variable of type `Fixed` or `LongInt`
`FracDiv`	Divide two variables of the same type (`Fixed`, `Fract`, or `LongInt`) or divide a `LongInt` or `Fixed` number by a `Fract` number
`FracSqrt`	Compute the square root of a variable of type `Fract`
`FracCos`	Obtain the cosine of a variable of type `Fixed`
`FracSin`	Obtain the sine of a variable of type `Fixed`
`FixATan2`	Obtain the arctangent of a variable of type `Fixed, Fract`, or `LongInt`
`Long2Fix`	Convert a variable of type `LongInt` to `Fixed`
`Fix2Long`	Convert a variable of type `Fixed` to `LongInt`
`Fix2Frac`	Convert a variable of type `Fixed` to `Fract`
`Frac2Fix`	Convert a variable of type `Fract` to `Fixed`
`Fix2X`	Convert a variable of type `Fixed` to `Extended`
`X2Fix`	Convert a variable of type `Extended` to `Fixed`
`Frac2X`	Convert a variable of type `Fract` to `Extended`
`X2Frac`	Convert a variable of type `Extended` to `Fract`
`FixRatio`	Obtain the `Fixed` equivalent of a fraction
`FixRound`	Round a fixed-point number to the nearest integer
`LongMul`	Multiply two 32-bit quantities and obtain a 64-bit quantity

Mathematical and Logical Utilities Reference

This section provides a complete reference to the Mathematical and Logical Utilities routines provided by the Macintosh Operating System. The section "Data Structures" describes the 64-bit integer record. The section "Routines" describes the routines that the Operating System includes to allow you to perform simple mathematical and logical operations.

Data Structures

This section describes the 64-bit integer record. For information on the numeric formats of fixed-point numbers, see "Fixed-Point Data Types" beginning on page 3-11. For information on the format of other numeric data types, consult the *Apple Numerics Manual*.

64-Bit Integer Record

By using the `LongMul` procedure, you can multiply two 32-bit quantities and obtain a 64-bit quantity stored in a 64-bit integer record. The `Int64Bit` data type defines a 64-bit integer record.

```
TYPE Int64Bit =
RECORD
   hiLong: LongInt;
   loLong: LongInt;
END;
```

Field descriptions

`hiLong`	The high-order long integer of the 64-bit integer.
`loLong`	The low-order long integer of the 64-bit integer.

Routines

This section describes the Mathematical and Logical Utilities supported directly by the Macintosh Operating System. Note that none of the routines in this section moves memory; therefore, all of the described routines in this section can be called at interrupt time.

Testing and Setting Bits

This section describes the `BitTst` function and the `BitSet` and `BitClr` procedures. You can test a bit using `BitTst` and specify a bit's value using `BitSet` and `BitClr`. All three of these procedures use the reversed bit-numbering scheme described in the section "Reversed Bit-Numbering" on page 3-7.

BitTst

You can use the `BitTst` function to determine whether a given bit is set.

```
FUNCTION BitTst (bytePtr: Ptr; bitNum: LongInt): Boolean;
```

`bytePtr` A pointer to a byte in memory.

`bitNum` The bit to be tested, specified as a positive offset from the high-order bit of the byte pointed to by the `bytePtr` parameter. The bit being tested need not be in the byte pointed to by `bytePtr`.

DESCRIPTION

The `BitTst` function returns `TRUE` if the bit specified by the `bytePtr` and `bitNum` parameters is set (that is, has a value of 1) and returns `FALSE` if the specified bit is cleared (that is, has a value of 0).

SPECIAL CONSIDERATIONS

The bit-numbering scheme used by the `BitTst` function is the opposite of MC680x0 bit numbering. To convert an MC680x0 bit number to the format required by the `BitTst` function, subtract the MC680x0 bit number from the highest bit number.

SEE ALSO

For an example of the use of the `BitTst` function, see Listing 3-2 on page 3-15. For more information about reversed bit-numbering see, "Reversed Bit-Numbering" on page 3-7.

BitSet

You can use the `BitSet` procedure to set a particular bit.

```
PROCEDURE BitSet (bytePtr: Ptr; bitNum: LongInt);
```

`bytePtr` A pointer to a byte in memory.

`bitNum` The bit to be set, specified as a positive offset from the high-order bit of the byte pointed to by the `bytePtr` parameter. The bit being set need not be in the byte pointed to by `bytePtr`.

DESCRIPTION

The `BitSet` procedure sets (to a value of 1) the bit specified by the `bytePtr` and `bitNum` parameters.

SPECIAL CONSIDERATIONS

The bit-numbering scheme used by the `BitSet` procedure is the opposite of MC680x0 bit numbering. To convert an MC680x0 bit number to the format required by the `BitSet` procedure, subtract the MC680x0 bit number from the highest bit number.

SEE ALSO

For an example of the use of the `BitSet` procedure, see page 3-16. For more information about reversed bit-numbering see "Reversed Bit-Numbering" on page 3-7.

BitClr

You can use the `BitClr` procedure to clear a particular bit.

```
PROCEDURE BitClr (bytePtr: Ptr; bitNum: LongInt);
```

`bytePtr` A pointer to a byte in memory.

`bitNum` The bit to be cleared, specified as a positive offset from the high-order bit of the byte pointed to by the `bytePtr` parameter. The bit being cleared need not be in the same byte pointed to by `bytePtr`.

DESCRIPTION

The `BitClr` procedure clears (to a value of 0) the bit specified by the `bytePtr` and `bitNum` parameters.

SPECIAL CONSIDERATIONS

The bit-numbering scheme used by the `BitClr` procedure is the opposite of MC680x0 bit numbering. To convert an MC680x0 bit number to the format required by the `BitClr` procedure, subtract the MC680x0 bit number from the highest bit number.

SEE ALSO

For an example of the use of the `BitClr` procedure, see page 3-16. For more information about reversed bit-numbering, see "Reversed Bit-Numbering" on page 3-7.

Performing Logical Operations

The Operating System supports five functions to support bit-level logical operations. The `BitAnd`, `BitOr`, `BitXor`, `BitNot`, and `BitShift` functions perform `AND`, `OR`, `XOR`, `NOT`, and bit-shifting operations, respectively. These routines are intended primarily for Pascal programmers. If you are programming in C, you can typically use C operators to perform the same logical operations more efficiently.

BitAnd

You can use the `BitAnd` function to perform the `AND` logical operation on two long words.

```
FUNCTION BitAnd (value1, value2: LongInt): LongInt;
```

`value1` A long word.

`value2` A long word.

DESCRIPTION

The `BitAnd` function returns a long word that is the result of performing the `AND` operation on the long words specified by the `value1` and `value2` parameters. Each bit in the returned value is set if and only if the corresponding bit is set in both `value1` and `value2`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitAnd` function, see Figure 3-9 on page 3-16.

BitOr

You can use the `BitOr` function to perform the `OR` logical operation on two long words.

```
FUNCTION BitOr (value1, value2: LongInt): LongInt;
```

`value1` A long word.

`value2` A long word.

DESCRIPTION

The `BitOr` function returns a long word that is the result of performing the `OR` operation on the long words specified by the `value1` and `value2` parameters. Each bit in the returned value is set if and only if the corresponding bit is set in `value1` or `value2`, or in both `value1` and `value2`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitOr` function, see Figure 3-9 on page 3-16.

BitXor

You can use the `BitXor` function to perform the `XOR` logical operation on two long words.

```
FUNCTION BitXor (value1, value2: LongInt): LongInt;
```

`value1` A long word.

`value2` A long word.

DESCRIPTION

The `BitXor` function returns a long word that is the result of performing the `XOR` operation on the long words specified by the `value1` and `value2` parameters. Each bit in the returned value is set if and only if the corresponding bit is set in either `value1` or `value2`, but not in both `value1` and `value2`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitXor` function, see Figure 3-9 on page 3-16.

BitNot

You can use the `BitNot` function to perform the `NOT` logical operation on a long word.

```
FUNCTION BitNot (value: LongInt): LongInt;
```

`value` A long word.

DESCRIPTION

The `BitNot` function returns a long word that is the result of performing the `NOT` operation on the long word specified by the `value` parameter. Each bit in the returned value is set if and only if the corresponding bit is not set in `value`.

SEE ALSO

For an illustration of the result of performing an operation using the `BitNot` function, see Figure 3-10 on page 3-17.

BitShift

You can use the `BitShift` function to shift bits in a long word.

```
FUNCTION BitShift (value: LongInt; count: Integer): LongInt;
```

`value` A long word.

`count` The number of bits to shift. If this number is positive, `BitShift` shifts this many positions to the left; if this number is negative, `BitShift` shifts this many positions to the right. The value in this parameter is converted to the result of `MOD` 32.

DESCRIPTION

The `BitShift` function returns a long word that is the result of shifting the bits in the long word specified by the `value` parameter. The shift's direction and extent are determined by the `count` parameter. Zeroes are shifted into empty positions regardless of the direction of the shift.

SEE ALSO

For an illustration of the result of performing an operation using the `BitShift` function, see Figure 3-10 on page 3-17.

Getting and Setting Memory Values

The `HiWord` and `LoWord` functions allow you to extract a word from a long word. The `StuffHex` procedure provides a quick way to convert hexadecimal values stored in a string into byte values in memory.

To copy a range of bytes from one memory location to another, you should ordinarily use the Memory Manager's `BlockMove` procedure, which is described in *Inside Macintosh: Memory.*

HiWord

You can use the `HiWord` function to obtain the high-order word of a long word. One use of this function is to obtain the integral part of a fixed-point number.

```
FUNCTION HiWord (x: LongInt): Integer;
```

`x` The long word whose high word is to be returned.

DESCRIPTION

The `HiWord` function returns the high-order word of the long word specified by the `x` parameter.

LoWord

You can use the `LoWord` function to obtain the low-order word of a long word. One use of this function is to obtain the fractional part of a fixed-point number.

```
FUNCTION LoWord (x: LongInt): Integer;
```

`x` The long word whose low word is to be returned.

DESCRIPTION

The `LoWord` function returns the low-order word of the long word specified by the `x` parameter.

StuffHex

You can use the `StuffHex` procedure to hardcode byte values into memory.

```
PROCEDURE StuffHex (thingPtr: Ptr; s: Str255);
```

`thingPtr` A pointer to any data structure in memory. If `thingPtr` is an odd address, then `thingPtr` is interpreted as pointing to the next word boundary.

`s` A string of characters representing hexadecimal digits. Be sure that all characters in this string are hexadecimal digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F). Otherwise, `StuffHex` may set bytes in the data structure pointed to by `thingPtr` to arbitrary values. If there are an odd number of characters in the string, the last character is ignored.

DESCRIPTION

The `StuffHex` procedure sets bytes in memory beginning with that byte specified by the parameter `thingPtr`. The total number of bytes set is equivalent to `s[0] DIV 2` (that is, half the length of the string, ignoring the last character if the number of characters is odd).

Each byte to be set corresponds to two characters in the string. These characters should represent hexadecimal digits. For example, the string `'D41A'` results in 2 bytes being set to the values $D4 and $1A, respectively.

Although the `StuffHex` procedure sets the value of individual bytes, it does not move relocatable blocks. Thus, you can call it at interrupt time.

SPECIAL CONSIDERATIONS

The `StuffHex` procedure does no range checking to ensure that bytes being set are within the bounds of a certain data structure. If you do not use `StuffHex` carefully, you may change memory in the partition of your application or another application in unpredictable ways.

SEE ALSO

For examples of the use of the `StuffHex` procedure, see page 3-19.

Compressing and Decompressing Data

You can use the `PackBits` function to compress a source buffer of data into a destination buffer and the `UnpackBits` function to decompress a source buffer of `PackBits`-compressed data into a destination buffer.

PackBits

You can use the `PackBits` procedure to compress a data buffer stored in RAM.

```
PROCEDURE PackBits (VAR srcPtr, dstPtr: Ptr; srcBytes: Integer);
```

`srcPtr`	On entry, a pointer to the first byte of a buffer of data to be compressed. On exit, a pointer to the first byte following the bytes compressed.
`dstPtr`	On entry, a pointer to the first byte in which to store compressed data. On exit, a pointer to the first byte following the compressed data.
`srcBytes`	The number of bytes of uncompressed data to be compressed. In versions of software prior to version 6.0.2, this number must be 127 or less.

DESCRIPTION

The `PackBits` procedure compresses `srcBytes` bytes of data beginning at the location specified by the `srcPtr` parameter and stores it at the location specified by the `dstPtr` parameter. It then modifies the `srcPtr` and `dstPtr` variables to point to the first bytes after the uncompressed and compressed data, respectively.

Your application must allocate memory for the destination buffer itself. In general, you should allocate enough memory for a worst-case scenario. In the worst case, the destination buffer is 128 bytes long for each block of source data up to 127 bytes. Thus, you can use the following formula to determine how much space to allocate for the destination buffer:

```
maxDstBytes := srcBytes + (srcBytes+126) DIV 127;
```

where `maxDstBytes` stands for the maximum number of destination bytes.

The `PackBits` algorithm is most effective on data buffers in which there are likely to be series of bytes containing the same value. For example, resources of many formats often contain many consecutive zeros. If you have a data buffer in which there are only likely to be series of words or long words containing the same value, `PackBits` is unlikely to be effective.

Because your application must allocate memory for the source and destination buffers, `PackBits` does not move relocatable blocks. Thus, you can call it at interrupt time.

SPECIAL CONSIDERATIONS

Because `PackBits` changes the values of the `srcPtr` and `dstPtr` parameters, you should pass to `PackBits` only copies of pointers to the source and destination buffers. This allows you to access the beginning of the source and destination buffers after `PackBits` returns. Also, if the source or destination buffer is stored in an unlocked, relocatable block, this technique prevents `PackBits` from changing the value of a master pointer, which would make the original handle invalid.

SEE ALSO

For an example of the use of the `PackBits` procedure, see Listing 3-3 on page 3-20.

UnpackBits

You can use the `UnpackBits` procedure to decompress a data buffer containing data compressed by `PackBits`.

```
PROCEDURE UnpackBits (VAR srcPtr, dstPtr: Ptr; dstBytes: Integer);
```

`srcPtr` On entry, a pointer to the first byte of a buffer of data to be decompressed. On exit, a pointer to the first byte following the compressed data.

`dstPtr`	On entry, a pointer to the first byte in which to store decompressed data. On exit, a pointer to the first byte following the decompressed data.
`dstBytes`	The number of bytes of the data before compression. In general, you should either use `PackBits` to compress data structures of a fixed size that you can then pass in this parameter to `UnpackBits`, or store with the compressed data the original size of the uncompressed data.

DESCRIPTION

The `UnpackBits` procedure decompresses `srcBytes` bytes of data beginning at the location specified by the `srcPtr` parameter and stores it at the location specified by the `dstPtr` parameter. It then modifies the `srcPtr` and `dstPtr` variables to point to the first bytes after the compressed and decompressed data, respectively.

Because your application must allocate memory for the source and destination buffers, `UnpackBits` does not move relocatable blocks. Thus, you can call it at interrupt time.

SPECIAL CONSIDERATIONS

Because `UnpackBits` changes the values of the `srcPtr` and `dstPtr` parameters, you should pass to `UnpackBits` only copies of pointers to the source and destination buffers. This allows you to access the beginning of the source and destination buffers after `UnpackBits` returns. Also, if the source or destination buffer is stored in an unlocked, relocatable block, this technique prevents `UnpackBits` from changing the value of a master pointer, which would make the original handle invalid.

SEE ALSO

For an example of the use of the `UnpackBits` procedure, see Listing 3-4 on page 3-21.

Obtaining a Pseudorandom Number

You can gain access to the Operating System's pseudorandom number generator by using the `Random` function.

Random

You can use the `Random` function to obtain a pseudorandom integer.

```
FUNCTION Random: Integer;
```

DESCRIPTION

The `Random` function returns a pseudorandom integer, uniformly distributed in the range –32767 to 32767.

The value `Random` returns depends solely on the global variable `randSeed`, which the QuickDraw `InitGraf` procedure initializes to 1. Each time the `Random` function executes, it uses a numerical algorithm to change the value of `randSeed` to prevent it from returning the same value each time it is called.

To prevent your application from generating the same sequence of pseudo-random numbers each time it is executed, initialize the `randSeed` global variable, when your application starts up, to a volatile long word variable such as the current date and time. If you would like to generate the same sequence of pseudo-random numbers twice, on the other hand, simply set `randSeed` to the same value before calling `Random` for each sequence.

ASSEMBLY-LANGUAGE INFORMATION

You can access the global variable `randSeed` through the system global variable `RndSeed`.

SEE ALSO

Listing 3-5 on page 3-22, Listing 3-6 on page 3-23, Listing 3-7 on page 3-23, and Listing 3-8 on page 3-24 for examples of how to use the `Random` function.

Converting Between Angle and Slope Values

You can use the `SlopeFromAngle` and `AngleFromSlope` functions to convert between angle and slope values.

SlopeFromAngle

You can convert an angle value to a slope value using the `SlopeFromAngle` function.

```
FUNCTION SlopeFromAngle (angle: Integer): Fixed;
```

`angle` The angle, expressed in clockwise degrees from 12 o'clock and treated `MOD` 180. (90 degrees is thus at 3 o'clock and –90 degrees is at 9 o'clock.)

DESCRIPTION

The `SlopeFromAngle` function returns the slope corresponding to the angle specified in the `angle` parameter. Slopes are defined as $\Delta x/\Delta y$, the horizontal change divided by the vertical change between any two points on a line with the given angle. The negative y-axis is defined as being at 12 o'clock, and the positive y-axis at 6 o'clock. The x-axis is defined as usual, with the positive side defined as being at 3 o'clock.

SEE ALSO

For an example of the use of the `SlopeFromAngle` function, see Figure 3-7 on page 3-13.

AngleFromSlope

You can convert a slope value to an angle value using the `AngleFromSlope` function.

```
FUNCTION AngleFromSlope (slope: Fixed): Integer;
```

`slope` The slope, defined as $\Delta x/\Delta y$, which is the horizontal change divided by the vertical change between any two points on a line with the slope.

DESCRIPTION

The `AngleFromSlope` function returns the angle corresponding to the slope specified in the `slope` parameter treated `MOD` 180. Angles are defined in clockwise degrees from 12 o'clock. The negative y-axis is defined as being at 12 o'clock, and the positive y-axis at 6 o'clock. The x-axis is defined as usual, with the positive side defined as being at 3 o'clock.

SPECIAL CONSIDERATIONS

The `AngleFromSlope` function is most useful when you require speed more than accuracy in performing the calculation. The integer result is within 1 degree of the correct answer, but not necessarily within half a degree.

SEE ALSO

For an example of the use of the `AngleFromSlope` function, see Figure 3-7 on page 3-13.

Multiplying and Dividing Fixed-Point Numbers

The `FixMul` and `FracMul` functions allow you to multiply fixed-point numbers. The `FixDiv` and `FracDiv` functions allow you to divide fixed-point numbers. By performing appropriate type casting, you can multiply or divide a fixed-point number of one type with a fixed-point number of another type or a long integer.

FixMul

You can use the `FixMul` function to multiply a variable of type `Fixed` with another variable of type `Fixed` or with a variable of type `Fract` or `LongInt`.

```
FUNCTION FixMul (a, b: Fixed): Fixed;
```

a The first operand, which can be a variable of type `Fixed` or a variable of type `Fract` or `LongInt`.

b The second operand, which can be a variable of type `Fixed` or a variable of type `Fract` or `LongInt`.

DESCRIPTION

The `FixMul` function returns the product of the numbers specified in the `a` and `b` parameters. At least one of `a` and `b` should be a variable of type `Fixed`.

The returned value is in the format of a `LongInt` if one of `a` or `b` is a `LongInt`. It is a `Fract` number if one of `a` or `b` is `Fract`. It is a `Fixed` number if both `a` and `b` are `Fixed` numbers.

Overflows are set to the maximum representable value with the correct sign ($80000000 for negative results and $7FFFFFFF for positive results).

SEE ALSO

For a summary of the routines that perform operations on the `Fixed` and `Fract` data type, see Table 3-2 on page 3-26.

FixDiv

You can use the `FixDiv` function to divide two variables of the same type (`Fixed`, `Fract`, or `LongInt`) or to divide a `LongInt` or `Fract` number by a `Fixed` number.

```
FUNCTION FixDiv (a, b: Fixed): Fixed;
```

a The first operand, which can be a variable of type `Fixed` or a variable of type `Fract` or `LongInt`.

b The second operand, which can be a variable of type `Fixed` or it can be a variable of the same type as the variable in parameter `a`.

DESCRIPTION

The `FixDiv` function returns the quotient of the numbers specified in the `a` and `b` parameters. If the `b` parameter is in the format of a `Fixed` number, then the `a` parameter can be in the format of a `Fixed`, `Fract`, or `LongInt` number. If the b parameter is in the format of a `Fract` or `LongInt` number, then the `a` parameter must be in the same format.

The returned value is in the format of a `Fixed` number if both `a` and `b` are both `Fixed` numbers, both `Fract` numbers, or both `LongInt` numbers. Otherwise, the returned value is the same type as the number in the `a` parameter.

Division by zero results in $8000000 if `a` is negative, and $7FFFFFFF otherwise; thus the special case 0/0 yields $7FFFFFFF.

SEE ALSO

For a summary of the routines that perform operations on the Fixed and Fract data type, see Table 3-2 on page 3-26.

FracMul

You can use the `FracMul` function to multiply a variable of type `Fract` with another variable of type `Fract` or with a variable of type `Fixed` or `LongInt`.

```
FUNCTION FracMul (a, b: Fract): Fract;
```

`a`	The first operand, which can be a variable of type `Fract` or a variable of type `Fixed` or `LongInt`.
`b`	The second operand, which can be a variable of type `Fract` or a variable of type `Fixed` or `LongInt`.

DESCRIPTION

The `FracMul` function returns the product of the numbers specified in the `a` and `b` parameters. At least one of `a` or `b` should be a variable of type `Fract`.

The returned value is in the format of a `LongInt` number if one of `a` and `b` is a `LongInt` number. It is a `Fixed` number if one of `a` or `b` is a `Fixed` number. It is a `Fract` number if both `a` and `b` are `Fract` numbers.

Overflows are set to the maximum representable value with the correct sign ($80000000 for negative results and $7FFFFFFF for positive results).

SEE ALSO

For a summary of the routines that perform operations on the Fixed and Fract data type, see Table 3-2 on page 3-26.

FracDiv

You can use the `FracDiv` function to divide two variables of the same type (`Fract`, `Fixed`, or `LongInt`) or to divide a `LongInt` or `Fixed` number by a `Fract` number.

```
FUNCTION FracDiv (a, b: Fract): Fract;
```

a The first operand, which can be a variable of type `Fract` or a variable of type `Fixed` or `LongInt`.

b The second operand, which can be a variable of type `Fract` or a variable of the same type as the variable in parameter `a`.

DESCRIPTION

The `FracDiv` function returns the quotient of the numbers specified in the `a` and `b` parameters. If the `b` parameter is in the format of a `Fract` number, then the `a` parameter can be in the format of a `Fract`, a `Fixed`, or a `LongInt` number. If the `b` parameter is in the format of a `Fixed` or a `LongInt` number, then the `a` parameter must be in the same format.

The returned value is in the format of a `Fract` number if `a` and `b` are both `Fract` numbers, both `Fixed` numbers, or both `LongInt` numbers. Otherwise, the returned value is in the same format as the number in the `a` parameter.

Division by zero results in $8000000 if `a` is negative, and $7FFFFFFF otherwise; thus the special case 0/0 yields $7FFFFFFF.

Performing Calculations on Fixed-Point Numbers

The Operating System provides four functions that you can use to perform a few common calculations on fixed-point numbers. The `FracSqrt` function allows you to obtain the square root of a number. The `FracCos`, `FracSin`, and `FixATan2` functions allow you to obtain fast approximations of trigonometric functions on fixed-point numbers.

FracSqrt

You can use the `FracSqrt` function to obtain the square root of a `Fract` number.

```
FUNCTION FracSqrt (x: Fract): Fract;
```

x The `Fract` number to obtain a square root of. This parameter is interpreted as being unsigned in the range 0 through $4 - 2^{-30}$, inclusive. That is, the bit of a `Fract` number that ordinarily has weight –2 is instead interpreted as having weight 2.

DESCRIPTION

The `FracSqrt` function returns the square root of the `Fract` number you supply in the `x` parameter. The result is unsigned in the range 0 through 2, inclusive.

FracCos

You can use the `FracCos` function to obtain a fast approximation of the cosine of a `Fixed` number.

```
FUNCTION FracCos (x: Fixed): Fract;
```

`x` The `Fixed` number expressed in radians, whose cosine is to be calculated.

DESCRIPTION

The `FracCos` function returns the cosine, expressed in radians, of the `Fixed` number `x`.

The approximation of π/4 used to compute the cosine is the hexadecimal value 0.C910, making the approximation of π equal to 3.1416015625, while π itself equals 3.14159265.... Despite the approximation of π, the cosine value obtained is usually correct to several decimal places.

FracSin

You can use the `FracSin` function to obtain a fast approximation of the sine of a `Fixed` number.

```
FUNCTION FracSin (x: Fixed): Fract;
```

`x` The `Fixed` number expressed in radians, whose sine is to be calculated.

DESCRIPTION

The `FracSin` function returns the sine, expressed in radians, of the `Fixed` number `x`.

The approximation of π/4 used to compute the sine is the hexadecimal value 0.C910, making the approximation of π equal to 3.1416015625, while π itself equals 3.14159265.... Despite the approximation of π, the sine value obtained is usually correct to several decimal places.

FixATan2

You can use the `FixATan2` function to obtain a fast approximation of the arctangent of a fraction.

```
FUNCTION FixATan2 (x, y: LongInt): Fixed;
```

x The numerator of the fraction whose arctangent is to be obtained. This variable can be a `LongInt`, `Fixed`, or `Fract` number.

y The denominator of the fraction whose arctangent is to be obtained. The number supplied in this variable must be of the same type as that of the number supplied in the `x` parameter.

DESCRIPTION

The `FixATan2` function returns, in radians, the arctangent of `y/x`.

The approximation of π/4 used to compute the arctangent is the hexadecimal value 0.C910, making the approximation of π equal to 3.1416015625, while π itself equals 3.14159265.... Thus `FixATan2(1, 1)`equals the equivalent of the hexadecimal value 0.C910. Despite the approximation of π, the arctangent value obtained will usually be correct to several decimal places.

Converting Among 32-Bit Numeric Types

The Operating System includes functions that allow you to convert among variables of type `LongInt`, `Fixed`, and `Fract`. The `Long2Fix` and `Fix2Long` functions convert between `LongInt` variables and `Fixed` variables. The `Fix2Frac` functions and `Frac2Fix` functions convert between `Fixed` and `Fract` variables. Ordinarily, there is no need to convert between `LongInt` and `Fract` variables, because `Fract` variables are used only to represent very small numbers. If you wish to do so, however, you can combine functions shown in this section.

Long2Fix

You can use the `Long2Fix` function to convert a `LongInt` number to a `Fixed` number.

```
FUNCTION Long2Fix (x: LongInt): Fixed;
```

x The long integer to be converted to a `Fixed` number.

DESCRIPTION

The `Long2Fix` function returns the `Fixed` number equivalent to the long integer you supply in the `x` parameter. If `x` is greater than the maximum representable fixed-point number, the `Long2Fix` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `Long2Fix` returns $80000000.

Fix2Long

You can use the `Fix2Long` function to convert a `Fixed` number to a `LongInt` number.

```
FUNCTION Fix2Long (x: Fixed): LongInt;
```

`x` The `Fixed` number to be converted to a long integer.

DESCRIPTION

The `Fix2Long` function returns the long integer nearest to the `Fixed` number you supply in the `x` parameter. If `x` is halfway between two integers (0.5), it is rounded to the integer with the higher absolute value.

Fix2Frac

You can use the `Fix2Frac` function to convert a `Fixed` number to a `Fract` number.

```
FUNCTION Fix2Frac (x: Fixed): Fract;
```

`x` The `Fixed` number to be converted to a `Fract` number.

DESCRIPTION

The `Fix2Frac` function returns the `Fract` number equivalent to the `Fixed` number x. If `x` is greater than the maximum representable `Fract` number, the `Fix2Frac` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `Fix2Frac` returns $80000000.

Frac2Fix

You can use the `Frac2Fix` function to convert a `Fract` number to a `Fixed` number.

```
FUNCTION Frac2Fix (x: Fract): Fixed;
```

`x` The `Fract` number to be converted to a `Fixed` number.

DESCRIPTION

The `Frac2Fix` function returns the `Fixed` number that best approximates the `Fract` number you supply in the `x` parameter.

Converting Between Fixed-Point and Floating-Point Values

The Mathematical and Logical Utilities provide four functions that allow you to convert between fixed-point and floating-point values represented using SANE's `Extended` floating-point data type. The `Fix2X` function and the `X2Fix` function convert between `Fixed` and `Extended` numbers. The `Frac2X` and `X2Frac` functions convert between `Fract` and `Extended` numbers. See *Apple Numerics Manual* for information about numeric data types supported by SANE.

Fix2X

You can use the `Fix2X` function to convert a `Fixed` number to an `Extended` number.

```
FUNCTION Fix2X (x: Fixed): Extended;
```

`x` The `Fixed` number to be converted to an `Extended` number.

DESCRIPTION

The `Fix2X` function returns the `Extended` equivalent of the `Fixed` number you supply in the `x` parameter.

SPECIAL CONSIDERATIONS

Because the `Fix2X` function does not move memory, you can call it at interrupt time.

X2Fix

You can use the `X2Fix` function to convert an `Extended` number to a `Fixed` number.

```
FUNCTION X2Fix (x: Extended): Fixed;
```

`x` The `Extended` number to be converted to a `Fixed` number.

DESCRIPTION

The `X2Fix` function returns the best `Fixed` approximation of the `Extended` number you supply in the `x` parameter. If `x` is greater than the maximum representable `Fixed` number, the `X2Fix` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `X2Fix` returns $80000000.

Frac2X

You can use the `Frac2X` function to convert a `Fract` number to an `Extended` number.

```
FUNCTION Frac2X (x: Fract): Extended;
```

`x` The `Fract` number to be converted to an `Extended` number.

DESCRIPTION

The `Frac2X` function returns the `Extended` equivalent of the `Fract` number you supply in the `x` parameter.

X2Frac

You can use the `X2Frac` function to convert an `Extended` number to a `Fract` number.

```
FUNCTION X2Frac (x: Extended): Fract;
```

`x` The `Extended` number to be converted to a `Fract` number.

DESCRIPTION

The `X2Frac` function returns the best `Fract` approximation of the `Extended` number you supply in the `x` parameter. If `x` is greater than the maximum representable `Fract` number, the `X2Frac` function returns $7FFFFFFF. If `x` is less than the negative number with the highest absolute value, `X2Frac` returns $80000000.

Converting Between Fixed-Point and Integral Values

To convert the quotient of two integers to a `Fixed` number, you can use the `FixRatio` function. To obtain the integral portion of a number of type `Fixed`, typically you just use the `HiWord` function, described on page 3-33. However, you can also use the `FixRound` function to obtain the integer nearest a fixed-point number.

FixRatio

You can use the `FixRatio` function to obtain the `Fixed` equivalent of a fraction.

```
FUNCTION FixRatio (numer, denom: Integer): Fixed;
```

`numer` The numerator of the fraction.

`denom` The denominator of the fraction.

DESCRIPTION

The `FixRatio` function return the `Fixed` equivalent of the fraction `numer/denom`.

FixRound

You can use the `FixRound` function to round a fixed-point number to the nearest integer.

```
FUNCTION FixRound (x: Fixed): Integer;
```

`x` The `Fixed` number to be rounded.

DESCRIPTION

The `FixRound` function returns the `Integer` number nearest the `Fixed` number you supply in the `x` parameter. If the value is halfway between two integers (0.5), it is rounded up. Thus, 4.5 is rounded to 5, and –3.5 is rounded to –3.

To round a negative `Fixed` number so that values halfway between two integers are rounded to the number with the higher absolute value, negate the number, round it, and then negate it again.

Multiplying 32-bit values

To multiply a 32-bit value and return a 64-bit value, you can use the `LongMul` procedure.

LongMul

You can use the `LongMul` procedure to multiply two 32-bit quantities and obtain a 64-bit quantity.

```
Procedure LongMul (a, b: LongInt; VAR result: Int64Bit);
```

`a` The first operand, which is a variable of type `LongInt`.

`b` The second operand, which is a variable of type `LongInt`.

`result` A pointer to the returned value.

DESCRIPTION

Given two variables of type `LongInt`, the `LongMul` procedure multiplies the two variables specified in parameter `a` and `b`, and returns the value in the variable specified by the `result` parameter.

Summary of the Mathematical and Logical Utilities

Pascal Summary

Data Types

```
TYPE
   Fixed          = LongInt;        {fixed-point number}
   Fract          = LongInt;        {fractional number}

   Int64Bit =                       {64-bit integer record}
   RECORD
      hiLong:     LongInt;          {high-order long integer}
      loLong:     LongInt;          {low-order long integer}
   END;
```

Routines

Testing and Setting Bits

```
FUNCTION BitTst             (bytePtr: Ptr; bitNum: LongInt): Boolean;
PROCEDURE BitSet            (bytePtr: Ptr; bitNum: LongInt);
PROCEDURE BitClr            (bytePtr: Ptr; bitNum: LongInt);
```

Performing Logical Operations

```
FUNCTION BitAnd             (value1, value2: LongInt): LongInt;
FUNCTION BitOr              (value1, value2: LongInt): LongInt;
FUNCTION BitXor             (value1, value2: LongInt): LongInt;
FUNCTION BitNot             (value: LongInt): LongInt;
FUNCTION BitShift           (value: LongInt; count: Integer): LongInt;
```

Getting and Setting Memory Values

```
FUNCTION HiWord             (x: LongInt): Integer;
FUNCTION LoWord             (x: LongInt): Integer;
PROCEDURE StuffHex          (thingPtr: Ptr; s: Str255);
```

Compressing and Decompressing Data

```
PROCEDURE PackBits          (VAR srcPtr, dstPtr: Ptr; srcBytes: Integer);
PROCEDURE UnpackBits        (VAR srcPtr, dstPtr: Ptr; dstBytes: Integer);
```

Obtaining a Pseudorandom Number

```
FUNCTION Random             : Integer;
```

Converting Between Angle and Slope Values

```
FUNCTION SlopeFromAngle     (angle: Integer): Fixed;
FUNCTION AngleFromSlope     (slope: Fixed): Integer;
```

Multiplying and Dividing Fixed-Point Numbers

```
FUNCTION FixMul             (a, b: Fixed): Fixed;
FUNCTION FixDiv             (a, b: Fixed): Fixed;
FUNCTION FracMul            (a, b: Fract): Fract;
FUNCTION FracDiv            (a, b: Fract): Fract;
```

Performing Calculations on Fixed-Point Numbers

```
FUNCTION FracSqrt           (x: Fract): Fract;
FUNCTION FracCos            (x: Fixed): Fract;
FUNCTION FracSin            (x: Fixed): Fract;
FUNCTION FixATan2           (x, y: LongInt): Fixed;
```

Converting Among 32-Bit Numeric Types

```
FUNCTION Long2Fix           (x: LongInt): Fixed;
FUNCTION Fix2Long           (x: Fixed): LongInt;
FUNCTION Fix2Frac           (x: Fixed): Fract;
FUNCTION Frac2Fix           (x: Fract): Fixed;
```

Converting Between Fixed-Point and Floating-Point Values

```
FUNCTION Fix2X              (x: Fixed): Extended;
FUNCTION X2Fix              (x: Extended): Fixed;
FUNCTION Frac2X             (x: Fract): Extended;
FUNCTION X2Frac             (x: Extended): Fract;
```

Converting Between Fixed-Point and Integral Values

```
FUNCTION FixRatio           (numer, denom: Integer): Fixed;
FUNCTION FixRound           (x: Fixed): Integer;
```

Multiplying 32-bit Values

```
Procedure LongMul           (a, b: LongInt; VAR result: Int64Bit);
```

C Summary

Data Types

```
typedef long Fixed;                     /*fixed-point number*/
typedef long Fract;                     /*fractional number*/

struct Int64Bit {                       /*64-bit integer record*/
   long hiLong;                         /*high-order long integer*/
   long loLong;                         /*low-order long integer*/
};
typedef struct Int64Bit Int64Bit;
```

Routines

Testing and Setting Bits

```
pascal Boolean BitTst       (const void *bytePtr, long bitNum);
pascal void BitSet          (void *bytePtr, long bitNum);
pascal void BitClr          (void *bytePtr, long bitNum);
```

Performing Logical Operations

```
pascal long BitAnd          (long value1, long value2);
pascal long BitOr           (long value1, long value2);
pascal long BitXor          (long value1, long value2);
pascal long BitNot          (long value);
pascal long BitShift        (long value, short count);
```

Getting and Setting Memory Values

```
pascal short HiWord         (long x);
pascal short LoWord         (long x);
pascal void StuffHex        (void *thingPtr, ConstStr255Param s);
```

Compressing and Decompressing Data

```
pascal void PackBits        (Ptr *srcPtr, Ptr *dstPtr, short srcBytes);
```

```
pascal void UnpackBits      (Ptr *srcPtr, Ptr *dstPtr, short dstBytes);
```

Obtaining a Pseudorandom Number

```
pascal short Random         (void);
```

Converting Between Angle and Slope Values

```
pascal Fixed SlopeFromAngle
                            (short angle);
pascal short AngleFromSlope
                            (Fixed slope);
```

Multiplying and Dividing Fixed-Point Numbers

```
pascal Fixed FixMul         (Fixed a, Fixed b);
pascal Fixed FixDiv         (Fixed a, Fixed b);
pascal Fract FracMul        (Fract a, Fract b);
pascal Fract FracDiv        (Fract a, Fract b);
```

Performing Calculations with Fixed-Point Numbers

```
pascal Fract FracSqrt       (Fract x);
pascal Fract FracCos        (Fixed x);
pascal Fract FracSin        (Fixed x);
pascal Fixed FixATan2       (long x, long y);
```

Converting Among 32-Bit Numeric Types

```
pascal Fixed Long2Fix       (long x);
pascal long Fix2Long        (Fixed x);
pascal Fract Fix2Frac       (Fixed x);
pascal Fixed Frac2Fix       (Fract x);
```

Converting Between Fixed-Point and Floating-Point Values

```
pascal Extended Fix2X       (Fixed x);
pascal Fixed X2Fix          (Extended x);
pascal Extended Frac2X      (Fract x);
pascal Fract X2Frac         (Extended x);
```

Converting Between Fixed-Point and Integral Values

```
pascal Fixed FixRatio       (short numer, short denom);
pascal short FixRound       (Fixed x);
```

```
Multiplying 32-bit values
Pascal void LongMul         (long a, long b, Int64Bit *result);
```

Global Variables

`randSeed`	The seed to the pseudorandom number generator.