Blitz-64:

Software Reference Manual

Harry H. Porter III

HHPorter3@gmail.com

10 November 2022

This document describes the existing Blitz-64 software. It describes the packages **System**, **PrintPackage**, **MiscLib**, **HostInterface**, **Number**. It documents the **printf** and **sprintf** statements, widely used functions that appear in many KPL programs, and the functions built-in to the KPL language. It describes the implementation of the stack and the heap. It describes the use of floating point numbers.

Table of Contents

Chapter 1: System Package

Introduction

Code in the KPL programming language is broken into packages. This document describes the most important packages.

Every KPL program must include at least the package named "**System**", since that package contains several functions the KPL compiler assumes will be present.

If a program does not print—which means the program contains no use of **printf**—then the programmer can avoid using **PrintPackage** and perhaps any other package. But every program requires **System**.[1]

For example:

```
header MyExampleProgram
  uses System, PrintPackage
  functions
    main ()
endHeader

code MyExampleProgram
  function main ()
    printf ("Hello, world\n")
  endFunction
endCode
```

The basic package hierarchy is:

[1] Technically, **HostInterface** uses no packages, but it is not feasible to write programs using only **HostInterface**.

```
HostInterface
  System
    PrintPackage
      MiscLib, Number, etc…
```

Roughly speaking, **System** contains:

Useful constants
Useful type definitions
Functions required by the compiler
Common, generally useful functions
Error declarations
Error handling code
Declarations for assembly-coded functions from **runtime.s**

Useful Constants

These constant values are defined:

Name	***Value in hex***	***Value in decimal***
MIN_8	0xFFFF_FFFF_FFFF_FF80	-128
MAX_8	0x0000_0000_0000_007F	127
MAX_UNSIGNED_8	0x0000_0000_0000_00FF	255
MIN_16	0xFFFF_FFFF_FFFF_8000	-32,768
MAX_16	0x0000_0000_0000_7FFF	32,767
MAX_UNSIGNED_16	0x0000_0000_0000_FFFF	65,535
MIN_24	0xFFFF_FFFF_FFF8_0000	-8,388,608
MAX_24	0x0000_0000_0007_FFFF	8,388,607
MAX_UNSIGNED_24	0x0000_0000_000F_FFFF	16,777,216
MIN_32	0xFFFF_FFFF_8000_0000	-2,147,483,648
MAX_32	0x0000_0000_7FFF_FFFF	2,147,483,647
MAX_UNSIGNED_32	0x0000_0000_FFFF_FFFF	4,294,967,295
MIN_36	0xFFFF_FFF8_0000_0000	-34,359,738,368
MAX_36	0x0000_0007_FFFF_FFFF	34,359,738,367
MAX_UNSIGNED_36	0x0000_000F_FFFF_FFFF	68,719,476,736

```
MIN_64           0x8000_0000_0000_0000   -9,223,372,036,854,775,808
MAX_64           0x7FFF_FFFF_FFFF_FFFF    9,223,372,036,854,775,807
MAX_UNSIGNED_64  0xFFFF_FFFF_FFFF_FFFF   18,446,744,073,709,551,616
```

String-Related Functions

String

```
type String = ptr to array of byte
```

newString

```
function newString (size: int) returns String
```

This function allocates a new String on the heap, with room for up to **size** bytes. It sets **currentSize** = **maxSize**, effectively filling the array.

NOTE: The bytes are uninitialized. While initial allocations in the heap pick up zeros, this may not always be the case.

maximizeString

```
external function maximizeString  (p: ptr to array of byte)
```

This function is passed a String, i.e., a pointer to an array of bytes. It alters the current size of the array to its maximum size. It does not alter any elements so the values will be determined by whatever happened to be in memory.

Here is an example:

```
var s: String = "AAAxxxxx"     -- MAX size = 8
```

```
...
setArraySize (s, 3)     -- Now s = "AAA"
maximizeString (s)      -- Now s = "AAAxxxxx"
```

If the array is uninitialized, this function throws **ERROR_UninitializedArray**.

This function is very similar to **initializeArray**, and this function can often be used instead.

```
maximizeString (s)
initializeArray (s)   — Almost equivalent
```

If the array has already been initialized, then both functions will have the same effect. If the array has not been initialized, **maximizeString** will throw an error, while **initializeArray** will not.

The **maximizeString** function is implemented as an external assembly routine, and is not known to the KPL compiler.

strCopy

```
function strCopy (str: String) returns String
```

This function allocates and returns a new **String** with the same bytes as the argument **str**.

If there is extra space beyond the CURRENT_SIZE size of **str**, this will not be allocated in the new string; the returned string will have CURRENT_SIZE equal to MAX_SIZE.

strEqual

```
function strEqual (s1, s2: String) returns bool
```

This function returns true if and only if the two strings have the same size and contain the same bytes. Only bytes up to the current size are checked.

append

```
function append (str1, str2: String) returns String
```

This functions allocates a new string and copies **str1** appended with **str2** into the new string. It returns the new string.

append3

```
function append3 (str1, str2, str3: String) returns String
```

This functions allocates a new string and copies **str1** || **str2** || **str3** into the new string. It returns the new string.

overwriteString

```
function overwriteString (s1, s2: String)
```

The characters of **s2** are copied into **s1**.

More precisely, this function copies the bytes from **s2** to **s1**.

If the two strings have different sizes, this function will copy however many characters it can, i.e., it will copy "min (**s1**.size, **s2**.size)" characters. The sizes of **s1** and **s2** are not changed.

Note that if **s1** is longer than **s2**, you may not get exactly what you expect, since some of the characters originally in **s1** will remain visible.

Only the CURRENT sizes are used; bytes beyond the current size are neither read nor written.

appendStrings

```
function appendStrings (target, source: String)
```

Modifies **target**, setting **target** := **target** || **source**.

This function appends the **source** string to the end of the **target** string. This function modifies **target** in place. If **target** is not large enough, or if either array is uninitialized, it throws

```
ERROR_AppendStringProblem (target, source: String)
```

and makes no changes to **target**.[2]

Here is an example:

```
var
  dest: String = "ABC            "   -- Extra room in the array
  src: String = "xxxxx"
...
setArraySize (dest, 3)
appendStrings (dest, src)    -- Sets dest to "ABCxxxxx"
```

[2] Perhaps we should implement this function in assembly code for performance reasons. If we find that both pointers are are properly aligned, we can copy 8 bytes at a time, in a tight loop, leading to 5/8 instructions per byte copied. The source pointer will always be properly aligned. But if the target pointer is not properly aligned, we can use code like this, for 11/16 instructions per byte copied.

```
        load.d    r1,...
    loop:
        load.d    r4,...
        inject1d  r1,r1,r4,...
        inject2d  r2,r2,r4,...
        store.d   ...,r1
        load.d    r4,...
        inject1d  r2,r2,r4,...
        inject2d  r1,r1,r4,...
        store.d   ...,r2
        addi      ...
        addi      ...
        bxxx      loop
        store.x   ...,r1
```

If the **targetString** is not large enough, or if either array is uninitialized, this function will throw **AppendStringError**.

Currently, this function is implemented in KPL, but it will likely be changed in the future to assembly code for efficiency reasons.

addStringToBuffer

```
function addStringToBuffer (targetBuffer: String,
                            p: int,
                            str: String)
                                returns int
```

This function copies the characters of **str** into **targetBuffer** at position **p**. Whatever bytes were previously there are overwritten.

It returns **nextPos**, the index in **targetBuffer** immediately following the last character copied. (This number can be used for the next call to add more characters.)

The CURRENT size of **targetBuffer** is assumed to be large enough. If not, an array indexing error will be thrown.

The pointer **str** may be null; if so, then no bytes are copied.

digitValue

```
function digitValue (ch: int)   returns int
```

If the argument is an ASCII digit—i.e., within '0'...'9'—return the corresponding integer between 0 and 9. Otherwise it returns -1.

hexCharToInt

```
function hexCharToInt (ch: int) returns int
```

This function is passed a character. If it is a hex digit, i.e.,

```
0, 1, 2, … 9, a, b, … f, A, B, … F
```

then it returns its value (0..15). Otherwise, it returns -1.

For example, the ASCII code for 'F' is 70. So when applied to 70, this function returns 15.

intToString

```
function intToString (i: int) returns String
```

This function is passed an integer. It allocates a **String** on the heap with the characters giving the decimal representation for this value, and returns a pointer to it.

The format of the result is shown by these examples:

"123"
"-7"

This function handles **MIN_64** correctly, by returning "-9223372036854775808".

This function ignores **printPreferences**; commas will never be inserted.

This function always allocates a new **String** on the heap. It is the responsibility of the caller to free this memory.

intToHexString

```
function intToHexString (i: int) returns String
```

This function is passed an integer. It allocates a **String** on the heap with the characters giving the hex representation for this value, and returns a pointer to it.

The format of the result is shown by this example:

"0x000000001234abcd"

This function always allocates a new **String** on the heap. It is the responsibility of the caller to free this memory.

parseWhitespace

```
function parseWhitespace (str: String, startIndex: int) returns int
```

This function scans past any ' ' and '\t' characters. It starts scanning at **startIndex** and returns the position of the first non-white character. If nothing is found (i.e., there is only whitespace from **startIndex** on), it returns the size of **str**.

stringToInt

```
function stringToInt (str: String,
                      resultAddr: ptr to int)
                          returns bool
```

This function scans **str** and parses an integer. It stores the value at the address given by **resultAddr**.

These strings show examples of what is acceptable:

"12345"
" +12_345 "
"-9_223_372_036_854_775_808" *This is MIN_64 =* `0x8000_0000_0000_0000`
"-0x12_34_5_abc" *Hex is allowed*

This function returns true if everything was okay and false if there was any parsing problem with the input string.

Leading/trailing white space is allowed. A leading '+' or '-' sign is allowed. The value may be given in decimal or hex. Underscores may be used as separators, according the KPL conventions.

This function calls **stringToIntWithOptions** to do the work.

stringToIntWithOptions

```
function stringToIntWithOptions (str: String,
                                 startingPos: int,
                                 allowHex: bool,
                                 scanTrailingWhitespace: bool,
                                 nextPosAddr: ptr to int,
                                 resultAddr: ptr to int
                                     ) returns bool
```

This function parses a string which is assumed to begin with an integer. It attempts returns the corresponding integer. It returns true if and only if the parse was successful and the string contained no errors.

If the parse is successful, it stores the result at **resultAddr**.

Acceptable syntax is the same as KPL integer token, except a leading '+' or '-' sign is acceptable and leading/trailing white space is allowed.

These strings show examples of what is acceptable:

"12345"	
" +12_345 "	
"-9_223_372_036_854_775_808"	*This is MIN_64 =* `0x8000_0000_0000_0000`
"-0x12_34_5_abc"	*Assuming **allowHex** is true*
"123hello"	*Assuming **scanTrailingWhitespace** is false*

If **scanTrailingWhitespace** is true, then this function will ensure that nothing but whitespace follows the number. If false, it will stop on the first character beyond the number.

If there are any problems, this function returns false and does not modify **result**.

This function allows "+0", but "-0" is considered to be an error.

On success, this function sets **nextPos** to just beyond the end of the number (and whitespace if that was also scanned). If there was a problem, it sets **nextPos** to approximately where the problem was detected.

If **allowHex** is true, then the value can be specified in either decimal or hex, with a leading "0x" to differentiate. Hex values may also begin with "0X". If **allowHex** is false, the value must be specified in decimal.[3]

dump

```
dump (p: ptr to void, numberOfInts: int)
-- Dump memory
```

This function prints the contents of memory. It is passed a pointer **p**, which is assumed to be doubleword aligned, and **numberOfInts**, which is the number of 8 byte doublewords that are to be printed.

It prints the address and the value, both in hex and decimal. For example:

```
ADDRESS             VALUE
0x000000000fffffc8  0xffffffffffffffff      dec: -1
0x000000000fffffd0  0x0000000000000000      dec: 0
0x000000000fffffd8  0x0000000000000000      dec: 0
0x000000000fffffe0  0x7ff8000000000000      dec: 9221120237041090560
0x000000000fffffe8  0x0000000005fc9e10      dec: 100441616
```

The output is sent to the basic serial output.

memoryZero

```
function memoryZero (destPtr: ptr to void,
                     byteCount: int)
```

[3] If **allowHex** is false but the value begins with "0x", then zero will be returned and scanning will stop at the 'x' in "0x". If **scanTrailingWhitespace** is also true, there will be an error at the 'x' in "0x".

This function sets a block of memory to zeros. The starting address is **p** and the number of bytes is **byteCount**.

The **byteCount** may be zero or negative, in which case this function does nothing. This function performs no other error checking. Neither **destPtr** nor **byteCount** need be aligned.

NOTE: Whenever possible, use **MemoryZero8** instead. **MemoryZero8** is implemented in assembly code and optimized for an aligned address.

memoryZero8

```
external function memoryZero8 (destPtr: ptr to void,
                               byteCount: int)
```

This function sets a block of memory to zeros. The starting address is **p** and the number of bytes is **byteCount**.

The pointer **p** must be evenly divisible by 8. If **byteCount** is not a multiple of 8, it will be rounded up to the next multiple of 8.

The **byteCount** may be zero or negative, in which case this function does nothing. This function performs no other error checking.

This function is implemented in assembly code for efficiency.[4]

memoryCopy

```
function memoryCopy (destPtr: ptr to void,
                     srcPtr: ptr to void,
                     byteCount: int)
```

This function copies **byteCount** bytes from one address to another.

[4] Besides efficiency, there is no reason to implement this is assembly. At some future time, it is hoped that the KPL compiler optimizations will be good enough to move this function back into KPL.

If **byteCount** <= 0, then this is a nop. Neither the pointers nor the byteCount need be aligned.

This function is implemented in assembly code for efficiency.[5]

NOTE: **MemoryCopy8** is similar in functionality. However, it is only usable for aligned addresses. Since using **MemoryCopy8** instead of **MemoryCopy** will only save a few instructions per call, the gain becomes negligible with larger **byteCounts**.

memoryCopy8

```
external function memoryCopy8 (destPtr: ptr to void,
                               srcPtr: ptr to void,
                               byteCount: int)
```

This function copies **byteCount** bytes from one address to another.

Both pointers must be evenly divisible by 8. If **byteCount** <= 0, then this is a nop. If **byteCount** is not a multiple of 8, it will be rounded up to the next multiple of 8.

This function is implemented in assembly code for efficiency.[6]

Misc. Functions

RuntimeExit

```
external function RuntimeExit ()
```

[5] Besides efficiency, there is no reason to implement this is assembly. At some future time, it is hoped that the KPL compiler optimizations will be good enough to move this function back into KPL.

[6] Besides efficiency, there is no reason to implement this is assembly. At some future time, it is hoped that the KPL compiler optimizations will be good enough to move this function back into KPL.

This function is used to terminate a KPL program after a serious, fatal error.

By "fatal error", we mean there can be no return to the running program. There will be no return from this function.

This function prints the message

```
==================  KPL PROGRAM TERMINATION  ==================
```

and executes a BREAKPOINT machine instruction.

The emulator will execute the BREAKPOINT instruction by halting instruction execution and going into command-line mode in which commands are accepted. At this point, the emulator's debugging capabilities can be used.

Any attempt to resume execution (using the "go" command in the emulator), will cause this function to print

```
===== The KPL program has terminated; you may not continue. ======
```

and once again execute a BREAKPOINT instruction.

NOTE: It is possible that timer interrupts are enabled and thread switching is occurring. This function will first disable interrupts, to suspend thread switching. CSR_STATUS is modified to disable interrupts in order to prevent an interrupt from hijacking this function during the printing of the termination message.

EmulatorDebuggingRequested

```
external function EmulatorDebuggingRequested (codeAddress: int)
```

This function is called by KPL code whenever we want to temporarily suspend execution and throw the user into the emulator's debugger. There may be a return from this function.

This function executes a BREAKPOINT instruction, which will suspend emulation.

The **codeAddress** will be passed to the emulator which will display the message:

```
**********  EMULATOR DEBUGGING: Type 'stack' for more info. **********
```

After using the emulator's debugging capabilities, the programmer can resume program execution with the "go" command, and a return from this function will occur.

NOTE: It is possible that timer interrupts are enabled and thread switching is occurring. This function will not disable interrupts.

It is likely the programmer will look at the state of the registers and then resume full program execution. So modifying the CSR_STATUS to disable interrupts would likely mess up execution and be wrong to do.

EmulatorShutdown

```
external function EmulatorShutdown (exitCode: int)
```

This function is the normal way to terminate the execution of a KPL program which is expected to be run under the emulator. There will be no return from this function.

This function is called by KPL code whenever we want to permanently terminate program execution. If the "auto-go option" ("-g") was specified, the emulator will immediately terminate without any message and exit, returning the value of **exitStatus** as the Unix/Linux/POSIX exit code.[7]

If auto-go was not specified, the emulator will enter command mode.

This function works by executing the SLEEP2 machine instruction, which the emulator will interpret as above. Any attempt to resume execution will cause this function to print

```
===== The KPL program has terminated; you may not continue. ======
```

and once again terminate execution.

[7] This function places the **exitCode** in register **r1**. The emulator will retrieve this code from **r1** before itself terminating.

FatalError

```
external function FatalError (errorMessage: String)
```

This function should be invoked by any code that encounters an error and simply cannot proceed.

The behavior of this function is "thread-dependent": it may do different things in different threads within the same process.

A program can change the behavior of this function by modifying the thread preferences. **FatalError** is actually a function pointer, which is normally initialized to **KPLDefaultFatalErrorFunction**.

The default behavior is to print

```
*********  FATAL KPL SYSTEM ERROR: "errorMessage"  **********
```

and invoke **RuntimeExit**.

This function is invoked by code in packages such as System. By refining FatalError, an application program can catch and perhaps handle an error that might occur during execution.

addOk

```
function addOk (x, y: int) returns bool
```

In KPL all arithmetic is performed with signed values and any overflow will cause an "Arithmetic Exception", which will result in **ERROR_ArithmeticException** being thrown.

Sometimes it is useful to know whether a particular addition will cause an overflow without actually performing the addition. The **addOk** function can do this.

The two arguments are integer expressions and the result is a **bool** value. This function returns **true** iff the two values may be added without causing an exception.

```
if (addOk (x, y))
  sum = x + y
else
  print ("Problems")
endIf
```

This function will never cause an exception or throw an error.

This function is implemented by the compiler which uses the ADDOK Blitz-64 machine instruction ADDOK.

subOk

```
function subOk (x, y: int) returns bool
```

This function tests whether the subtraction "**x** - **y**" would overflow and returns true if and only if the subtraction is safe.

unsignedAdd, unsignedSub

```
function unsignedAdd (x, y: int) returns int
function unsignedSub (x, y: int) returns int
```

In KPL all arithmetic is performed with signed values and any overflow will cause an "Arithmetic Exception", which will result in **ERROR_ArithmeticException** being thrown.

However, it is occasionally necessary to perform arithmetic without signaling an exception. These functions can be used for that.

These functions perform an addition / subtraction and return the result. Overflow is ignored. The arguments and result are all 64 bit **int** values.

```
var i, x, y: int
```

```
...
i = unsignedAdd (x, y)    -- computes i = x + y
...
i = unsignedSub (x, y)    -- computes i = x - y
```

Note that an integer argument that is smaller (e.g., a **byte**, **halfword**, or **word** value) will be sign-extended to a 64 bit value first.

The result will be identical, regardless of whether the arguments are interpreted as signed or unsigned.

There are no error conditions and no errors will be thrown.

To implement this, the compiler makes use of the Blitz-64 machine instruction ADD3. These functions could have been implemented in KPL, but at a significant cost of efficiency.

min

```
function min (i, j: int) returns int
```

This function returns the smaller of the two arguments.

max

```
function max (i, j: int) returns int
```

This function returns the larger of the two arguments.

RandomNumber

```
function RandomNumber () returns int
```

This function returns a pseudo-random integer in the range 0 … 0x7FFF_FFFE, that is within 0 … 2,147,483,646.

RandomNumberBetween

```
function RandomNumberBetween (lo, hi: int) returns int
```

This function returns a random number between lo and hi, inclusive. The range is limited. We must have hi-lo <= 0x7FFF_FFFE in order to get numbers throughout the range.

endianSwapH
endianSwapW
endianSwapD

```
external function endianSwapH (arg: int) returns halfword
external function endianSwapW (arg: int) returns word
external function endianSwapD (arg: int) returns int
```

These functions can be used to transform data between “big endian” and “little endian” byte ordering.[8]

The **endianSwapD** function takes any 64 bit (**int**) value and swaps all bytes from one order to the other.

```
0x1122_3344_5566_7788  →  0x8877_6655_4433_2211
0x8877_6655_4433_2211  →  0x1122_3344_5566_7788
```

The **endianSwapW** function can be used to swap bytes in a 32 bit word.

```
0x........2244_6688  →  0x8866_4422
0x........8866_4422  →  0x2244_6688
```

[8] The same function can be used to go in either direction, either big→little or little→big.

The argument can be any 64 bit (**int**) value. The upper 32 sign bits are ignored and the result is a **word** value. Essentially, this function executes the ENDIANW machine instruction.

The **endianSwapH** function can be used to swap bytes in a 16 bit word.

```
0x............1289  →  0x8912
0x............8912  →  0x1289
```

The argument can be any 64 bit (**int**) value. The upper 48 sign bits are ignored and the result is a **halfword** value. Essentially, this function executes the ENDIANH machine instruction.

Basic Serial Printing and Reading

The functions described here provide basic printing and a way to read a line of input from the user.

Since these functions are implemented in package **System**, they will be available to any KPL program. The code in **System** deals with error handling and these basic print functions are used throughout **System**.

Blitz has two systems for printing, which we can call "**basic serial I/O**" and "**printf functionality**". In this section we describe the "basic serial" system.

The functionality of **printf** and **sprintf** is more complex than basic serial I/O. Printf functionality is not available within **System** and may not be needed by or available in simple programs. See **PrintPackage** for more information about printf functionality.

Typically, the basic printing described here goes go directly through the serial I/O communication port. However, in some user-level programs, the basic serial I/O may be redirected to print on **stdin** and read from **stdin**.

Kernel code will not have access to **stdin**, **stdout**, **printf** functionality. Kernel code will use only the basic serial I/O functions described here. Instead, basic serial I/O may be directed to a file, or "console" process, or some other mechanism, which can be used during kernel debugging.

The basic I/O functions described in this section are all implemented in terms of these 3 functions.

SimpleSerial_PrintString
SimpleSerial_PrintChar
SimpleSerial_ReadString

Exactly what these do is discussed elsewhere, but their interface and functionality is identical to the **print**, **printChar**, and **readString** functions described below.

print

```
function print (msg: String)
```

This function sends the bytes in **msg** to the basic serial I/O. As is the Blitz convention, only the characters up to the CURRENT length are output. This allows all bytes, including \0 to be output.[9]

printNL

```
function printNL ()
```

This function sends the newline character, that is '\n' which is ASCII 0x0a, to the output.

printInt

```
function printInt (i: int)
```

This function prints the argument in decimal. There will be no trailing \n character.

[9] The string will often contain multi-byte Unicode characters; these are sent without any checking, under the assumption that the output device will interpret them appropriately.

If negative, the output will begin with '-', but otherwise, no '+' will be printed. No separating commas are printed.

printDecimal

```
function printDecimal (i: int)
```

This function is identical to **printInt**.

printHex

```
function printHex (i: int)
```

This function prints the argument in hex. There will be no trailing \n character.

It will print "0x" followed by 16 hex characters, with 'a'...'f' in lowercase. For example:

```
"0x0000007890ABCDEF"
```

printPtr

```
function printPtr (p: ptr to void)
```

This function is identical to **printHex**, except the argument has a pointer type rather than **int**.[10] For example:

```
"0x00000000800001208"
```

[10] Recall that type **ptr to void** will match type **ptr to T**, where **T** is any type.

printBool

```
function printBool (b: bool)
```

This function will print either “TRUE” of “FALSE”.

printChar

```
function printChar (ch: int)
```

This function will send a single byte to the serial output. For example:

```
printChar (‘w’)
```

This function ignores all but the lower order 8 bits of **ch**.[11]

For multi-byte Unicode characters, this function should be called once for each byte. So calling **printString** once for an entire string of bytes is functionally equivalent to calling **printChar** once for each byte in the array.

printIntVar

```
function printIntVar (msg: String, i: int)
```

This function **prints** the message **msg**, followed by the integer value **i** in decimal, followed by newline (‘\n’). Here is a typical usage:

```
printIntVar ("MyVariable = ", myVar)
```

[11] Normally, the argument is an int in the range of either -128…+127 or 0…255; these are equivalent ways of saying the upper 7 bytes of the doubleword argument are simply ignored.

printHexVar

```
function printHexVar (msg: String, i: int)
```

This function **prints** the message **msg**, followed by the integer value **i** in hex, followed by newline ('\n').

printBoolVar

```
function printBoolVar (msg: String, b: bool)
```

This function **prints** the message **msg**, followed by "TRUE" or "FALSE" as determined by **b**, followed by newline ('\n').

printPtrVar

```
function printPtrVar (msg: String, p: ptr to void)
```

This function **prints** the message **msg**, followed by the pointer **p** in hex, followed by newline ('\n').

printStrVar

```
function printStrVar (msg: String, str: String)
```

This function **prints** the message **msg**, followed by the string **str**, followed by newline ('\n').

For example

```
printStrVar ("Filename: ", fname)
```

might print:

```
Filename: hello.txt
```

printBinaryVar

```
function printBinaryVar (msg: String, n: int)
```

This function **prints** the message **msg**, followed by **n** in binary, followed by **n** in hex, followed by newline ('\n'). Spaces are inserted to aid readability.

For example

```
printBinaryVar ("x = ", -1234)
```

will print:[12]

```
x = 1111 1111 1111 1111   1111 1111 1111 1111   1111 1111 1111 1111
          1111 1011 0010 1110      0xffff_ffff_ffff_fb2e
```

readString

```
function readString (buffer: ptr to array of byte)
```

This function is used for input from the basic serial I/O device. Presumably, the input is a stream of UTF-8 encoded Unicode characters typed by the human user.

For example, this function may invoke **SimpleSerial_ReadString** from the **HostInterface** package.

All input is in the form of complete lines, not individual characters. Echoing, backspacing, etc. happen outside of this function and (presumably) is done before the characters reach this function.

[12] In this document, this single long line of output wraps into two lines.

The **buffer** argument is a pointer to an array of bytes. This function will fill up the buffer, ignoring CURRENT size. The last character added will be the newline character '\n' and the array size will be adjusted.

If the **buffer** array is not large enough, the newline \n character will be dropped. Also, some of the final characters will be dropped if there is insufficient space in **buffer**.[13]

Heap-Related Functions

Heap management is discussed more fully in another chapter in this document.

heapInitialize

```
heapInitialize (version: int)
```

This function is passed a number and it initializes the heap management algorithm to:

version	
#0	The default heap management algorithm
#1	Same as #0, adding hidden **byteCounts** and error checking
...	

Anything previously allocated on the heap must never be accessed again.

This function is not thread-safe. It assumes we are running with only a single thread.

[13] The emulator implementation of the **SimpleSerial_ReadString** operation will always add a null \0 byte at the end, but it will be added just beyond the CURRENT array size so it will be inaccessible. So at most, the CURRENT array size will be set to MAXSIZE-1. You can probably ignore this information.

getHeapCurrentInUse

```
getHeapCurrentInUse () returns int
```

This function returns the amount of memory that has been requested but not yet freed. It is the total size of the non-free chunks in the heap.

getHeapTotalAllocation

```
getHeapTotalAllocation () returns int
```

This function returns the total amount of memory ever requested in calls to **MemoryAlloc**. This includes includes memory that was subsequently freed.

getHeapTotalFreed

```
getHeapTotalFreed () returns int
```

This function returns the sum of all calls to **MemoryFree**.

NOTE: With Heap Management Algorithms #0 and #1, free space is not reused, so this number has no relation to how much usable space remains.

getHeapRemaining

```
getHeapRemaining () returns int
-- Largest chunk that can be safely allocated
```

This function tries to returns the greatest amount of memory that may be requested without causing a heap-full error (i.e., an insufficient heap space error).

With Heap Algorithms #0 and #1 which don't re-use memory, this is simply the number of bytes remaining in the Heap Region. For other algorithms, it may mean something else, such as the amount remaining until a call to the OS for more pages will occur, or it may be a meaningless number.

One use of this function is in debugging. The programmer will need to verify that the program works correctly, even in the presence of a heap-full condition. The idea is to invoke this function and use the returned value in a call to artificially eat up all remaining heap space, as in:

```
junk = MemoryAlloc (getHeapRemaining ())  -- fill the heap
```

getHeapRegionSize

```
getHeapRegionSize () returns int
```

This function returns the amount of memory currently in use for the heap in bytes. This includes currently allocated chunks of memory, previously freed chunks, and additional space that is available for future calls to **MemoryAlloc**.

checkHeapConsistency

```
checkHeapConsistency ()
```

This function runs through the heap and performs any consistency checks it can. If errors are found, it will print details and invoke **Error: HeapViolation**.

NOTE: This function is not thread-safe. Any simultaneous use of the heap by another thread can result in failure.

runThruHeap

```
runThruHeap ()
```

This function will run through the heap and print the address of each non-free object. It is designed for use in chasing down memory leak bugs.

If the object looks like a string, it will print the string out. A typical memory leak bug is failure to free strings; seeing the string contents helps in fixing this.

MemoryAlloc

```
external function MemoryAlloc (byteCount: int) returns ptr to byte
```

This function is called to allocate memory on the heap and it returns a pointer to the newly allocated chunk of memory.

This function never returns null.

This function is invoked as a result of using the alloc construct in KPL or by calling **MemoryAlloc** explicitly.

If the allocation cannot be done, this function throws:

ERROR_HeapFull

NOTE: The memory is not zeroed. However, with any use of the **alloc** construct the KPL compiler will implicitly insert initialization, so the concern is only with explicit calls to this function.[14]

Consult the section on heap management algorithms for additional details.

MemoryFree

```
external function MemoryFree (p: ptr to byte)
```

[14] The kernel will always zero address spaces upon creation to prevent security compromises and data leakage from other processes. Therefore, newly allocated heap memory will almost always contain zeros, so the bug of "failure to initialize memory" may go unnoticed.

This function is called to return unused memory to the heap management system.

Of course, the pointer must always have been returned from a previous allocation, memory must be freed no more than once, and memory must never be used accessed after being freed.

Memory management is discussed more fully elsewhere, but we emphasize that the recommended strategy is to completely avoid using this function. This is the safest and most efficient approach for almost every application program.

remainingStackSpace

```
external function remainingStackSpace () returns int

-- The number of bytes left (see STACK_SIZE .equ)
```

This function looks at the current value of register r15 (**sp**, the "stack pointer"). It returns the number of free bytes remaining in the stack.

```
var i: int
if (remainingStackSpace () < 100)  -- if bytes left < 100
  printf ("Warning!")
  ...
```

The runtime stack is used to implement function and method invocation. Whenever a function or method is invoked, a stack frame may be pushed onto the stack and, when they return, that stack frame is popped. (Some small functions are able to get by without needing a stack frame.)

The stack space is a predefined region of memory allocated and set up whenever a new thread is created.[15]

At this time, KPL does not perform any runtime checking for stack overflow. The general approach of allocating a large stack region and using the virtual memory system to watch for overflow into "sentinel pages" is reasonable for most non-

[15] The variables STACK_START and STACK_SIZE describe the stack region. These variables are in the assembly **runtime.s** support code and are not directly accessible to KPL code.

critical applications. For applications that require high reliability, programmers are encouraged use KPL's **maxStackUsage** facility and determine statically whether any overflow is possible. With this approach, the programmer can specify limits at compile time and the compiler can verify the program respects those limits.

The **remainingStackSpace** function is implemented as an external assembly routine, and is not known to the KPL compiler.

Floating-Point Functions

isNegZero

```
function isNegZero (d: double) returns bool
```

This function returns true if and only if the argument is -0.0.

Recall that with floating point numbers, there are two representations of zero, +0.0 and -0.0. Normally, they are treated as equal:

```
var d: double = ...
if d == 0.0...          True if d is +0.0 or -0.0
if d >= +0.0...         True even if d is -0.0!!!
```

The **isNegZero** function is provided in case it is necessary to distinguish the two zero values. This function takes a **double** value and returns true if and only if it is -0.0.

Negative zero is represented as 0x8000_0000_0000_0000 so

```
if isNegZero (d) ...
```

is not equivalent to either of the following, because they use "floating point equality":

```
if d == copyBitsToDouble (0x8000_0000_0000_0000) ...
if d == -0.0 ...
```

However, **isNegZero** is equivalent to the following, since it uses comparison of bits:

```
if copyBitsToInt (d) == 0x8000_0000_0000_0000 ...
```

isnan

```
function isnan (d: double) returns bool
```

This function takes a **double** value and determines if it is "not-a-number" and returns **true** if so, and false if the value is a valid number, including infinities.

```
var d: double = ...
if isnan (d) ...
```

Several values can be interpreted as "non-a-number" values. This function ignores the mantissa bits; in other words, it ignores whether the NaN is quiet or signaling and ignores any payload.

There are no error conditions.

Note that the builtin keyword **nan** returns the canonical "not-a-number" value.

```
d = nan
d = copyBitsToDouble (0x7FF8_0000_0000_0000)  -- Equivalent
```

Note that comparing against NaN is problematic; you should alway use **isnan** instead. For one thing, there are multiple values that can be interpreted as NaN. But even assuming that "d" has the canonical value of NaN, the following un-intuitive results are mandated by IEEE 754:

```
if d == nan ...          This is false!
if d != nan ...          This is true!
```

Note that the capitalization follows POSIX standards; "isNaN" is not used.

This function is implemented by the compiler but they could also have been implemented as a normal function in KPL.

isinf

```
function isinf (d: double) returns bool
```

This function returns true if and only if the argument is +inf or -inf.

We chose the lowercase “isinf” and not “isInf” to follow the POSIX spelling.

Positive infinity is represented as 0x7FF0_0000_0000_0000 and negative infinity is represented as 0xFFF0_0000_0000_0000, so

```
if isinf (d)
...
```

is equivalent to

```
if (d == copyBitsToDouble (0x7FF0_0000_0000_0000)) ||
   (d == copyBitsToDouble (0xFFF0_0000_0000_0000))
...
```

which is equivalent to

```
if (d == inf) || (d == -inf)
...
```

floatingClass

```
external function floatingClass (arg: double) returns int
```

This function is passed a **double** value. It returns an **int** value in which individual bits indicate the type of value it is. The bits of the result are defined as:

bit	Set to 1 if “arg" is...
0	+0
1	-0
2	Subnormal (including +0 and -0)
3	Normal

4	+inf
5	-inf
6	NaN

All other bits are zero.

In either words, this function returns a value 0x0000...0x007f, where least significant bits having these meanings:

	nan	-inf	+inf	normal	denormal	-0	+0
...	x	x	x	x	x	x	x

This function executes the FCLASS machine instruction, and masks out all bits except those shown here.

resetFloatingStatus

```
external function resetFloatingStatus ()
```

IEEE 754 specifies five arithmetic exceptions that are to be recorded in the status flags[16]:

NX	Inexact	Set if the rounded (and returned) value is different from the mathematically exact result of the operation.
UF	Underflow	Set if the rounded value is "tiny" and inexact, returning a subnormal value including the zeros.
OF	Overflow	Set if the absolute value of the rounded value is too large to be represented. An infinity or maximal finite value is returned, depending on which rounding is used.
DZ	Divide-by-zero	Set if the result is infinite given finite operands, returning an infinity, either +∞ or −∞.
NV	Invalid	Set if a real-valued result cannot be returned e.g. sqrt(−1) or 0/0, returning a quiet NaN.

The Blitz-64 machine has the following bits in the status word:

[16] Source: Wikipedia

NX	Inexact
UF	Underflow
OF	Overflow
DZ	Divide-by-zero[17]
NV	Invalid

These bits are “sticky”, which means that once set to 1, they remain unchanged until explicitly cleared to 0.

The purpose of the **resetFloatingStatus** function is to reset all the floating point status bits to 0.

NOTE: The FLOAT_STATUS bits are in the CSR_STATUS register, which is protected. Therefore this function will execute differently depending on whether this is kernel code or user code. In Kernel Mode, this function will clear the bits directly; in User Mode, this function must invoke a system call to do the work.

floatingInexact
floatingUnderflow
floatingOverflow
floatingDivideByZero
floatingInvalid

```
external function floatingInexact      () returns bool
external function floatingUnderflow    () returns bool
external function floatingOverflow     () returns bool
external function floatingDivideByZero () returns bool
external function floatingInvalid      () returns bool
```

Each of these functions returns the status of the corresponding bit in the machine status word.

true	1	This condition has occurred
false	0	This condition has not occurred

[17] The DZ: divide-by-zero status bit is newly added.

This function executes the FCLASS machine instruction, and returns the appropriate bit.

setFloatingInexact
setFloatingUnderflow
setFloatingOverflow
setFloatingDivideByZero
setFloatingInvalid

```
external function setFloatingInexact      ()
external function setFloatingUnderflow    ()
external function setFloatingOverflow     ()
external function setFloatingDivideByZero ()
external function setFloatingInvalid      ()
```

Each of these functions sets the status of the corresponding bit in the machine status word to 1, to indicate the condition has occurred.

true	1	This condition has occurred
false	0	This condition has not occurred

These functions are provided for use in the functions which implement the basic floating point operations. For example, the floating multiply operation may be implemented in hardware by the FMUL machine instruction. On the other hand, this operation may be emulated in software instead. Either way, the IEEE spec defines which status bits are to be set under which conditions. (For example, OF-OVERFLOW or NV-INVALID may need to be set.) These functions are provided for code such as the code emulating FMUL.

NOTE: The FLOAT_STATUS bits are in the CSR_STATUS register, which is protected. Therefore these functions will execute differently depending on whether this is kernel code or user code. In Kernel Mode, this function will set the bits directly; in User Mode, this function must invoke a system call to do the work.

floatingSqrt
floatingAbs
floatingMin
floatingMax

```
external function floatingSqrt (d: double)      returns double
external function floatingAbs  (d: double)      returns double
external function floatingMin  (d1, d2: double) returns double
external function floatingMax  (d1, d2: double) returns double
```

These functions are used to access the corresponding machine instructions[18]:

FSQRT	return the square root of the argument
FABS	return the absolute value of the argument
FMIN	return the minimum (most negative) of the arguments
FMAX	return the maximum (most positive) of the arguments

When applied to not-a-number (NaN) or when the square root function is applied to a negative number, it is an invalid operation. The result will be not-a-number (NaN) an the **NV-Invalid** floating point status flag will be set.

setFloatingRound - ELIMINATED

```
function setFloatingRound () returns int
```

This function has been removed. At this time, Blitz-64 always uses "round to nearest, with ties to even".

This function will set the FLOAT_ROUND bits in the CSR_STATUS register, which will determine how floating point computations are to be rounded, if an exact value cannot be represented. This function is passed an integer with these meanings:

```
0    Nearest
```

[18] Instructions such as FSQRT will almost certainly be emulated, but this doesn't affect the operation of these functions, beyond their execution speed.

```
1    Toward zero
2    Down
3    Up
```

The argument is taken “mod 4”, so only the lower 2 bits are used.

NOTE: The FLOAT_ROUND bits are in the CSR_STATUS register, which is protected. Therefore this function will execute differently depending on whether this is kernel code or user code. In Kernel Mode, this function will modify the bits directly; in User Mode, this function must invoke a system call to do the work.

Error Handling in KPL

Errors can be expected and good code should check for and handle any and all error conditions.

There are a number of strategies for error handling which are discussed next. Different approaches make sense for different programs. In one place or another, Blitz uses all these approaches.

Set a flag or error variable In this approach, code encountering an error will set some global variable. Later, some code will check this variable and take appropriate action.

This approach is mandated for floating point by IEEE 754, which names flags such as **OF-Overflow**, **NV-Invalid**, and **NX-Inexact**. In Blitz, these flags are kept in dedicated register storage—in CSR_STATUS—so the flags are naturally thread-specific and thread-safe.

The Unix/Linux/POSIX file functions (e.g., “fopen”, “fread”, …) use this approach to set the global variable **errno**.

Return a special error value The idea is that under normal circumstances, a function will return a result, but with an error, the function will return a value that is distinct from all normal results.

For example, the function **digitValue** is passed a character code and returns 0…9 if the character is ‘0’…’9’. It returns -1, if the character is not a decimal digit.

This approach is clean and simple. It is recommended especially when errors are expected from time-to-time and the caller is expected to check for errors immediately.

Call an error function that does not return This approach is useful when errors are not expected to occur or at least expected to occur rarely. When the error is discovered, the code will call some error function, perhaps passing additional information about the error to the function.

The function will take action, but the code sequence—and most likely the entire thread with the error—will be abandoned forever. There will be no return from the error function.

The approach is common in Blitz, and is used when the most likely action is to begin debugging. The action taken by the error function is to suspend the thread and invoke a debugger to examine the state of the thread at the moment of the error.

Call an error function that cleans up and returns In some cases, the error can be repaired immediately and the code detecting the problem needs to continue. Obviously, if the error can be handled directly, without calling any function, this could be cleaner. But if the error could arise in several places, it may make sense to encapsulate the error handling in a single function.

In this case, the error handling function could be viewed as a helper function.

Throw an error KPL contains a **try-throw-catch** mechanism that makes it possible to for functions with problems to throw a specific error. This allows code within the program itself to catch the error and respond appropriately.

This is the approach recommended for fault-tolerant programs. When errors arise, the program will not lose complete control of the situation. Although there can never be a return to the code throwing the error, the application itself remains in control.

There are two important differences between the approach of calling an error function and the approach of throwing errors.

Benefits of throw Many error conditions can arise outside of the application. For example, the “heap-full” condition will occur within the heap management algorithm. In addition conditions related to arithmetic exceptions, null pointers,

array violations, and so on, are not specific to a particular application, and must be handled properly, regardless of where they occur and whether some application chooses to include code to handle these errors.

For such errors, the throw mechanism must be used. The application can choose to handle (i.e., to **catch**) the error or ignore it. But either way, the error will be dealt with.

With the approach of creating a function to handle the error, recall that there can only be one definition of a function. So system code that may generate the error must define some function to handle it, which precludes the possibility of application code handling it. But the **throw-catch** mechanism nicely overcomes this issue.

Benefits of error functions Unfortunately, with a **throw**, the calling stack is popped and all information about local variables and which function called which function is lost. This makes debugging almost impossible.

The Blitz Approach The primary approach to error handling in the Blitz system is this:

- The code detecting the error calls an error function.
- The error function throws the corresponding error.
- The throw occurs inside a try statement, so if it is not caught the function maintains control.
- If the error is caught, the code catching it is responsible.
- If the error is not caught, the error function invokes the debugger.

Typically, the error function has a name beginning "**RuntimeError**…" and the error to be thrown begins "**ERROR_**…"

Example: Heap Full Within the **MemoryAlloc** function in the Heap Management System, it may be that the heap is exhausted and the request cannot be satisfied. The code will invoke function

RuntimeErrorHeapFull

Error functions such as this are defined in the package in which they are called. Both **MemoryAlloc** and **RuntimeErrorHeapFull** are defined in the **System** package.

The error function does this:

```
try
  throw ERROR_HeapFull (…)
catch UncaughtThrowError (…):
  KPLPrintCatchStack (…)
  print "Out of memory" message
  invokeDebugger (…)
endTry
RuntimeExit ()
```

In the event that the application program catches **ERROR_HeapFull**, then this function ends with the **throw** statement. But if the error is not caught, this function will print some information that may be useful for debugging and it will then invoke the debugger. The debugger should never return, but if is does, the emulator is terminated.

At the time of writing, the **invokeDebugger** function will suspend execution and invoke the emulator's debugger by calling **EmulatorDebuggingRequested**. In the future, this function may invoke a debugging thread running on the Blitz processor itself.

Errors from the Heap Management System:

During memory allocation—i.e., within **MemoryAlloc**—it is possible the heap has become exhausted, in which case a "heap full" condition exists.

During memory allocation, memory freeing, and other heap functions, it is possible the data structures used by the heap system have become corrupted. The common cause of this is when an application program calls **MemoryFree** incorrectly.

These errors are handled by the two functions described next.

Error: HeapFull

```
function RuntimeErrorHeapFull (byteCount: int)
```

This function throws

```
error ERROR_HeapFull (byteCount: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
Within MemoryAlloc, out of memory!
```

Error: HeapViolation

```
function RuntimeErrorHeapViolation (message: String)
```

This function throws

```
error ERROR_HeapViolation (message: String)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
The heap management functions have detected an error!
```

The **message** is also printed, to give additional information.

Errors from Runtime Exceptions:

The Blitz processor checks for several type of error during the execution of machine instructions. If an error condition arises, exception handling will occur. This is described fully in the Instruction Set Architecture (ISA).

Ultimately, one of the following **RuntimeError...** functions will be invoked. The benefit of the Blitz approach is that the errors described here are always checked but the checking involves zero additional instructions. Therefore, there is no additional execution overhead.

These execution-related errors are expected to be rare and indicative of program bugs. But as bugs go, these are common errors. They must never be ignored.

Error: ArithmeticException

```
function RuntimeErrorArithmeticException (codeAddress: int,
                                          offendingInstr: int)
```

This function throws

```
error ERROR_ArithmeticException (codeAddress: int,
                                 offendingInstr: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
An "ARITHMETIC EXCEPTION" has occurred!
```

Error: UnalignedLoadStore

```
function RuntimeErrorUnalignedLoadStore (codeAddress: int,
                                         offendingInstr: int,
                                         unalignedAddr: int)
```

This function throws

```
error ERROR_UnalignedLoadStore (codeAddress: int,
                                offendingInstr: int,
                                unalignedAddr: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
An "UNALIGNED LOAD/STORE" exception has occurred!
```

Error: NullAddress

```
function RuntimeErrorNullAddress (codeAddress: int)
```

This function throws

```
error ERROR_NullAddress (codeAddress: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
A "NULL ADDRESS" exception has occurred!
```

Error: BadArrayIndex

```
function RuntimeErrorBadArrayIndex (index: int,
                                    ptrToArray: ptr to word,
                                    codeAddress: int)
```

This function throws

```
error ERROR_BadArrayIndex (codeAddress: int,
                           index: int,
                           ptrToArray: ptr to word)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an array index calculation, the index is either less than 0
or greater than or equal to the current array size!
```

Commentary The errors of "Null Pointer" and "Bad Array Index" are common and I have seen them too many times. However, the error reporting of Blitz—which gives the exact line number in the source code file as well as the calling stack and local variables— is a godsend. It makes locating these bugs trivial in most cases. I cannot stress how useful the Blitz exception mechanism has turned out to be, especially for the **Null Address Exception** and the **Bad Array Index Exception**.

The "Unaligned Load / Store" error is quite uncommon and I have to work to cause it. The compiler places variables on properly aligned addresses, so LOADs and STOREs never cause this error.

The "Arithmetic Exception" error is also quite uncommon. Blitz and KPL use 64 bit signed numbers almost exclusively, and these are large enough to accommodate just about everything.

Error Functions Inserted by the Compiler

The following functions are not called directly by KPL code. Instead, the compiler will insert calls to these functions when needed.

Each function will try to **throw** the corresponding error. If the error is not caught, the function will print a message and invoke the debugger.

The arguments to the **RuntimeError...** functions give additional information pertinent to the specific error type. In several cases, the additional information will be printed after the leading error message.

Error: WrongObject1

```
function RuntimeErrorWrongObject1 (actualDPT: ptr to void,
                                   expectedDPT: ptr to void,
                                   codeAddress: int)
```

This function throws

```
error ERROR_WrongObject1 (codeAddress: int,
                          actualDPT: ptr to void,
                          expectedDPT: ptr to void)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an assignment of the form '*ptr = x', the ptr does not
already point to an instance of the same class as x!
```

Error: WrongObject2

```
function RuntimeErrorWrongObject2 (actualDPT: ptr to void,
                                   expectedDPT: ptr to void,
                                   codeAddress: int)
```

This function throws

```
error ERROR_WrongObject2 (codeAddress: int,
                          actualDPT: ptr to void,
                          expectedDPT: ptr to void)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an assignment of the form 'x = *ptr', the ptr does not
already point to an instance of the same class as x!
```

Error: WrongObject3

```
function RuntimeErrorWrongObject3 (targetPtr: ptr to void,
                                   sourcePtr: ptr to void,
                                   codeAddress: int)
```

This function throws

```
error ERROR_WrongObject3 (codeAddress: int,
```

```
                                targetPtr: ptr to void,
                                sourcePtr: ptr to void)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an object assignment of the form '*ptr1 = *ptr2', the two
objects are not instances of the same class!
```

Error: BadClassDescriptor

```
function RuntimeErrorBadClassDescriptor (targetPtr: ptr to void,
                                         sourcePtr: ptr to void,
                                         size: int,
                                         codeAddress: int)
```

This function throws

```
error ERROR_BadClassDescriptor (codeAddress: int,
                                targetPtr: ptr to void,
                                sourcePtr: ptr to void,
                                size: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an object assignment or object equality test, something is
wrong with the dispatch table pointer, the dispatch table, or the
class descriptor!
```

Error: UninitializedObject

```
function RuntimeErrorUninitializedObject (p: ptr to void,
                                          codeAddress: int)
```

This function throws

```
error ERROR_UninitializedObject (codeAddress: int,
```

```
                                        p: ptr to void)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
Attempt to use an uninitialized object!
```

or

```
Something is wrong with the dispatch table pointer, the dispatch
table, or the class descriptor!
```

Error: UninitializedArray

```
function RuntimeErrorUninitializedArray (p: ptr to void,
                                         codeAddress: int)
```

This function throws

```
error ERROR_UninitializedArray (codeAddress: int,
                                p: ptr to void)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
Attempt to use an uninitialized array!
```

Error: InitializingArray

```
function RuntimeErrorInitializingArray (arrPtr: ptr to word,
                                        newSize: int,
                                        codeAddress: int)
```

This function throws

```
error ERROR_InitializingArray (codeAddress: int,
                               arrPtr: ptr to word,
                               newSize: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During initializeArray(), the previous array contained an
unexpected max size!
```

or

```
During initializeArray(), the previous array contained an
unexpected current size!
```

Error: SetArraySize

```
function RuntimeErrorSetArraySize (arrPtr: ptr to word,
                                   newSize: int,
                                   codeAddress: int)
```

This function throws

```
error ERROR_SetArraySize (codeAddress: int,
                          arrPtr: ptr to word,
                          newSize: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During setArraySize(), the new size is either negative or greater
than max size!
```

Error: CurrentArraySizeIsWrong

```
function RuntimeErrorCurrentArraySizeIsWrong (
                                     targetPtr: ptr to void,
                                     sourcePtr: ptr to void,
                                     srcCurrent: int,
                                     targetMax: int,
                                     codeAddress: int)
```

This function throws

```
error ERROR_CurrentArraySizeIsWrong (codeAddress: int,
                                     targetPtr: ptr to void,
                                     sourcePtr: ptr to void,
                                     srcCurrent: int,
                                     targetMax: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an array copy, the CURRENT size of the source array is not
within 0...TargetMaxSize!
```

Error: ArrayTooLarge

```
function RuntimeErrorArrayTooLarge (size: int, codeAddress: int)
```

This function throws

```
error ERROR_ArrayTooLarge (codeAddress: int, size: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During an array ALLOC, the initial size is not within 1 ...
2147483647!
```

Error: ArrayCountNotPositive

```
function RuntimeErrorArrayCountNotPositive (count: int,
                                            codeAddress: int)
```

This function throws

```
error ERROR_ArrayCountNotPositive (codeAddress: int,
```

```
                                        count: int)
```

If this error is not caught, this function prints the following message before invoking the debugger:

```
During the initialization of an array, a 'count' expression was
zero or less!
```

Objects, Classes and Dispatch Tables

Each object in KPL is represented in memory with one or more doublewords of memory. The first word of any object is an 8-byte header word containing the **Dispatch Table Pointer (DPT)**. All objects are positioned on doubleword aligned addresses and the size of every object is a multiple of 8 bytes. An object with no fields will have size 8, since it will have a dispatch table pointer.

Every object of a given class will have the same size. There are no variable-sized classes as you find in higher level OO languages, such as Smalltalk.

A pointer to an object will point to the dispatch table pointer. In other words, the dispatch table pointer is at offset 0. The first field will be at offset 8.

The fields of an object will appear in the order specified in the class definition. All fields from a superclass will precede the fields added by a subclass. The compiler will insert additional padding bytes in order to achieve the proper alignment for every field.[19]

The root superclass of all classes is named **Object** and this class is defined in the **System** package. Class **Object** has no fields. The dispatch table pointer is of course a sort of hidden field, but there are no directly named and accessible fields.

Each class will have a single dispatch table and a single class descriptor. The dispatch table and the class descriptor are placed in a read-only memory segment

[19] If the programmer is concerned about the padding bytes, they can reorder the fields in the class definition. To eliminate padding bytes, the programmer should place the all doubleword-aligned fields first; then all word-aligned fields; then all halfword-aligned fields; and finally all remaining, byte-aligned fields.

along with the executable code (the machine instructions). Thus, they can be accessed and read, but cannot be modified at runtime.

Dispatch Tables

For each class there is a "**dispatch table**" and a "**class descriptor**". Each dispatch table (DPT) is a block of memory with this format:

Offset	
0	ptr to class descriptor
8	ptr to method #1
16	ptr to method #1
24	ptr to method #1
…	…
…	null

The first field is a pointer to the class's class descriptor. This is followed by zero or more pointers to the code for the class's methods. The last entry is a null pointer.

The **System** package includes this definition:

```
type KPL_DISPATCH_TABLE =
  struct
    classDescriptor: ptr to KPL_CLASS_DESCRIPTOR
    firstMethodPtr: ptr to …
 endStruct
```

Sending a Message (Invoking a Method)

As is the case with other object-oriented programming languages, messages are sent to objects. By "send a message", we mean that a method is invoked using **dynamic dispatching**.

The compiler will know the name of the method, but determining which method code is to be executed cannot be determined until runtime, because the compiler cannot always know the class of the receiver object. Only at runtime can the class of the object be determined, and even then, objects of different classes can be used in a single given message-send statement.

During runtime, the program will access the dispatch table whenever a message is sent to an object. It will use the dispatch table of the object to select the correct method.

Each method is assigned an offset in the dispatch table by the compiler. Due to the possibility of overriding in subclasses, there may be several methods with the same name. However, methods with the same name will always be assigned to the same offset and that is the key to efficient implementation of dynamic dispatching.

The dispatch table contains an array of addresses, where each address points to a method. More precisely, each entry in the dispatch table is the address of the first instruction of a method.

Invoking a method is almost identical to calling a function. When calling a function, the first argument is placed in register **r1** and the remaining registers contain the other arguments. When invoking a method, a pointer to the receiving object is placed in register **r1** and the remaining registers contain the arguments. In either case, a CALL instruction is executed and, upon return, a RET instruction is used.

Given a pointer to an object at runtime, the compiler will generate code to follow the object's DPT to the dispatch table and then to call indirectly through the pointer corresponding to the method.

Consider invoking a function with a single argument, such as:

```
foo (p)
```

Here is the code to invoke a function:

```
loadd      r1,…
call       foo
```

Now consider sending a message, such as:

```
p.foo ()
```

Here is the code[20]:

[20] Recall that CALL is a synthetic instruction that is replaced by a JALR machine instruction, so the last instruction is effectively a CALL.

```
    loadd     r1,…
    loadd     s0,0(r1)
    bne       s0,r0,Label_continue
    call      _runtimeErrorUninitializedObject
Label_continue:
    jalr      lr,…offset of foo within Dispatch Table…(s0)
```

Sending a message adds two additional instructions over calling a function. One instruction is required to fetch the dispatch table pointer and one is required to make sure the object is initialized[21]. There is also a CALL to **RuntimeErrorUninitializedObject**, but that is not normally executed.

Commentary The JALR can be used as long as the offset is less than 32,768, which it almost always will be. In the extremely unusual case it is not, the compiler generates an additional instruction.

On the other hand, the CALL instruction will often be to a function that is further away than 32,768 bytes. CALL is a synthetic instruction and will expand to two instructions in such cases.

Since CALL is either 1 or 2 instructions, we can approximate its size as 1.5 instructions. Comparing this to the 3 instructions required for invoking a method (i.e., LOADD, BNE, JALR), we can estimate the overhead for invoking a method versus calling a function to be about 1.5 instructions.

I consider this overhead for dynamic dispatch to be near optimal.

Class Descriptors

For each class there is also a "**class descriptor**", in addition to the dispatch table. A class descriptor is represented with a block of memory with this format:

Offset	
0	magic number
8	ptr to class name
16	ptr to filename

[21] An uninitialized object will have a dispatch table pointer still set to zero.

24	line number
28	object size
32	hash value
40	ptr to my DPT
48	ptr to DPT of superclass #1
…	ptr to DPT of superclass #2
…	ptr to DPT of superclass #3
…	…
…	ptr to Interface Descriptor #1
…	ptr to Interface Descriptor #2
…	ptr to Interface Descriptor #3
…	…
…	null

The first field (at offset 0) is a "magic number", which will always contain the value 0x434c415353646573, which happens to be the ASCII codes for "CLASSdes". This field exists solely to try to catch bugs.

The second field (offset 8) points to a String giving the name of the class.

The third field (offset 16) contains a pointer to a sequence of bytes giving the source file name. This sequence is terminated with the null byte, \0.[22]

The fourth field (offset 24) is a word sized unsigned integer giving the line number within the above named file at which the definition of this class occurred.

The fifth field (offset 28) is a word sized unsigned integer giving the size of instances of this class, in bytes. Objects are limited in size to 4 GiBytes.

The sixth field (offset 32) is a pseudo-random number for this class, generated by the compiler.[23] This hash value is used for an efficient implementation of the **switchOnClass** statement.

The seventh field (offset 40) is a pointer back to the Dispatch Table.

After this, there will be a sequence of pointers, with one for each superclass of this class. Each of these pointers will point to the dispatch table for the superclass.

---

[22] This needs to be changed.

[23] This hash value is a function solely of the class name.

After this, there will be a sequence of pointers, with one for each interface that this class implements. Each of these pointers will point to an Interface Descriptor.

Finally, the last pointer is followed by null, i.e., a doubleword of zero.

The **System** package includes this definition:

```
type KPL_CLASS_DESCRIPTOR =
  struct
    magic:          int
    myName:         String
    fileName:       ptr to byte
    lineNumber:     word
    sizeInBytes:    word
    clHashValue:    int
    myDPT:          ptr to KPL_DISPATCH_TABLE
    firstSuperPtr: ptr to KPL_DISPATCH_TABLE
  endStruct
```

Interface Descriptors

For each interface there is an "**interface descriptor**". An interface descriptor is represented with a block of memory with this format:

Offset	
0	magic number
8	ptr to interface name
16	ptr to filename
24	line number
28	zero filler
32	zero filler
40	ptr to this interface descriptor
…	ptr to Interface Descriptor #1
…	ptr to Interface Descriptor #2
…	ptr to Interface Descriptor #3
…	…
…	null

This definition matches and overlays the class descriptors exactly. This is important for software that must handle both.

The **System** package includes this definition:

```
type KPL_INTERFACE_DESCRIPTOR =
  struct
    magic:          int
    myName:         String
    fileName:       ptr to byte
    lineNumber:     word
    filler1:        word
    filler2:        int
    firstSuperPtr: ptr to KPL_INTERFACE_DESCRIPTOR
 endStruct
```

The first field (at offset 0), the magic number, will always contain the value 0x494e545246646573, which happens to be the ASCII codes for “INTRFdes”.

The second field (offset 8) points to a String giving the name of the interface.

The third field (offset 16) contains a pointer to a sequence of bytes giving the source file name.

The fourth field (offset 24) is a word sized unsigned integer giving the line number within the above named file at which the definition of this interface occurred.

The fifth field (offset 28) and the sixth field (offset 32) are unused and will contain zeros. one overlays the objects size and the other overlays the hash value.

After this (beginning at offset 40), there will be a sequence of pointers, with one for each interface that this interface extends. Each of these will point to an interface descriptor. The first entry will point to this interface descriptor itself. The last pointer is followed by a null pointer to terminate the list.

Uses of Dispatch Tables, Class Descriptors, and Interface Descriptors

The primary need for the class hierarchy information is in the implementation of **isKindOf** operation.

The **isKindOf** operation is implemented by a function[24] which takes as arguments two pointers. One argument points to an object and the other points to a dispatch table or an interface descriptor. The second pointer identifies the class or interface we are asking about. The **isKindOf** function will search the object's dispatch table for a pointer to the given dispatch table or interface descriptor.

During this search, the class and interface descriptors are never actually accessed; all that is important is their identity, which is given by the pointers. However, we choose to generate the fields of the class and interface descriptors as described here. The descriptors are included along with the code in a read-only section, for future use by the debugger.

The error reporting functions and the debugger also access the dispatch tables and the class descriptors to obtain the class name.

At this time, there is no information kept in memory at runtime regarding the fields of a class. In order for a debugger to display the fields of an object in any meaningful form in the future, a debugger will have to access the executable or source files or use some other approach.

The Try-Throw-Catch Mechanics

Each time a **try** statement is entered, a "**catch record**" created for each **catch** clause in the statement.

Catch Records

The **System** package includes this definition:

```
type CatchRecord =
  struct
    next:          ptr to CatchRecord
    errorID:       String
    catchCodeAddr: int
```

[24] This is a hand-coded assembly function in **runtime.s**.

```
    prevSP:         int
    sourcefile:     String
    sourceLine:     int
  endStruct
```

Catch records are kept in a singly linked list. At any moment in time, this linked list will tell which errors are being caught. In other words, if an error is thrown, the catch list will tell whether it should be caught and, if so, all the information that is needed to perform the "catching".

The first field of a catch record (**next**) is a pointer to the next record in the linked list.

The second field (**errorID**) points to a **String** giving the name of the error. For example, one error which is discussed elsewhere is **ERROR_HeapFull**. The compiler will add a prefix giving the package name, and will represent this error with this string:

```
"System: ERROR_HeapFull"
```

The catch record will contain a pointer to a string such as this.

This field is called "errorID" because the strings are used to identify the error in the Blitz system. Errors are not given ID numbers because generating unique numbers across multiple compilations and assemblies is impossible. But the string will be placed in memory exactly once (by the package in which it was declared) and the linker will communicate this address to all separately compiled packages during the link phase.

If this error is being caught, then there is a block of code (the **catch** clause in a **try** statement somewhere) that must be executed when the error is thrown. The third field (**catchCodeAddr**) contains the address of the first instruction in this code block.

At the time the try statement is entered, there will be some stack frames on the activation stack. Later, other functions and methods may be called and, at the time of the throw, there will be additional frames on the stack. All these additional frames must be popped when the error is caught.

The fourth field (**prevSP**) contains the value of the stack top pointer, i.e., the **sp** register, at the time the try statement was entered.

The fourth and fifth fields (**sourcefile** and **lineNumber**) are used by the debugger to indicate where the corresponding **catch** clause is located. When an error is uncaught, the debugger will print out the existing catch stack and the **sourcefile** and **lineNumber** fields will be used in the printing.

Catch records are not allocated on the heap. Instead, each catch record is a local variable and stored in the stack frame for the function or method containing the **try** statement. When the function is entered, the catch record is created along with the other local variables of the function. When the function returns, the catch record along with the other local variables are popped off the activation stack. There is one catch record in the stack frame for every **catch** clause appearing in the function.

The "**catch stack**" is a linked list of catch records, linked on the **next** field in the catch records. Upon entering a **try** statement, the compiler will produce code that will add one catch record to the catch stack for each **catch** clause in the statement. Upon exiting the **try** statement (at the **endTry)**, the code will pop the catch stack back to what it was upon entry to the **try** statement.

If a function contains any **try** statements, instructions at the very beginning of the function will save the value of the catch stack upon entry. It is possible to return from within a try statement, so whenever a **return** statement is executed, the compiler will generate code to restore the catch stack to this saved value.

When a **throw** statement is executed, the catch stack is searched for the first matching catch record. If one is found, execution will transfer to the statements in the **catch** clause. The catch record is used to pop the activation stack back to what it was at the time the **try** statement was entered. Also, the catch stack is restored to what it was before entering the try statement. These actions have the effect of exiting the body of the **try** statement.[25]

Because of all this, the entering and exiting a **try** statement and the execution of a **throw** statement is fairly expensive.[26]

---

[25] Of course, any objects allocated on the stack are not freed. The use of **throw** for normal processing is discouraged since it may introduce subtle memory leaks.

[26] Just as a very rough guide, the cost of entering and leaving a **try** statement is approximately (13 × n) + 5 instructions, where n is the number of **catch** clauses in the **try** statement. The cost of performing a **throw** is roughly (5 × k) + 11 instructions, where k is the number of catch records that must be searched, plus whatever is required for passing and retrieving arguments.

The catch stack is local to a thread. In other words, each thread will have its own catch stack. The pointer to the head of the current catch stack, i.e., to the first catch record in the linked list of catch records, is kept in the **ThreadData** object.

Thread-Specific Functionality

For each thread there will be an object of type ThreadData, which will contain the information pertinent to the thread. For application programs running in user-space, this object will be in the virtual address space and so it will be accessible by thread's code.

ThreadData

Here is the layout of the **ThreadData** objects:

```
class ThreadData
  superclass Object
  fields
    catchStack:    ptr to CatchRecord
    memAllocFun:   ptr to ALLOC_FUNCTION_TYPE
    memFreeFun:    ptr to FREE_FUNCTION_TYPE
    fatalErrorFun: ptr to FATAL_ERROR_FUNCTION_TYPE
    threadName:    String
    threadNumber:  int
    threadPrefs:   ptr to ThreadPreferences
endClass
```

The fields of **ThreadData** objects are "fixed" in the sense that the KPL compiler and language system rely on them. While fields might be added, any changes to existing fields risk introducing subtle bugs.

This is a class so the first field (**catchStack**) will be at offset 8. Each object has a dispatch table pointer at offset 0.

The **catchStack** field will point to the first catch record—the top—of the catch stack.

Each thread can potentially use a different heap and a different heap allocation algorithm. While this would be unusual, it is supported with the second and third fields (**memAllocFun** and **memFreeFun**). These fields point to the functions that are to be executed whenever **MemoryAlloc** or **MemoryFree** is called.

MemoryAlloc is an assembly function defined in **runtime.s** as:

```
MemoryAlloc:    load.d    t,OFFSET_OF_memAllocFun(tp)
                jr        t
```

Recall that register **tp** points to the **ThreadData** object. Whenever **MemoryAlloc** is called, the first instruction will get the address of the function to invoke from the known offset (i.e., 16) of the **memAllocFun** field within **ThreadData**. The second instruction is an indirect jump to that address.

The same scheme is used for functions **MemoryAlloc**, **MemoryFree**, and **FatalError**. This flexibility imposes a two instruction overhead whenever one of these functions is invoked.

Every thread also has a function called **FatalError** which is configurable in the same way as **MemoryAlloc** and **MemoryFree**. The **fatalErrorFun** field points to the address of the function to be invoked whenever **FatalError** is called.

The **threadName** field is used in debugging. The **threadID** is a small integer, used to distinguish threads with the same name. The initial thread in an application is named “Main Thread” and given a number of 0. This can be changed, if desired.

The **threadPrefs** field points to an object of type **PrintPreferences**. This object contains information used in printing. In particular, these preferences are used in the **PrintPackage** and the **Number** package.

The **PrintPreference** object contains, for example, information about whether to use ‘.’ or ‘,’ for the decimal point in floating point numbers, whether or not to include ‘+’ for positive numbers, etc.

In a multi-threaded application, all threads may share a single **PrintPreference** objects. Alternatively, each might have a unique object, with different settings.

Although not implemented at this time, other localization / preference data may be handled similarly.

threadPtr

```
function threadPtr () returns int
```

At all times, register **tp** ("thread pointer register") is dedicated to point to the current thread's **ThreadData** object.

A pointer to the **ThreadData** object is occasionally needed and the function **threadPtr** is used to retrieve the value of register **tp**.

This function returns the value of the register as an **int**, so it would normally be cast to type "**ptr to** ThreadData", as shown next.

Here is an example of the use of this function.

Each thread has a number of "preference" values, which might be changed with code like this. See the system types **ThreadData** and **PrintPreferences** for more detail.

```
var
  threadDataPtr: ptr to ThreadData
  prefs: ptr to PrintPreferences

threadDataPtr = asPtrTo (threadPtr (), ThreadData)
prefs = threadDataPtr.threadPrefs.printPrefs

prefs.integerSeparator    = ','
prefs.floatPosInf         = "+∞"
prefs.floatNegInf         = "-∞"
prefs.floatNaN            = "NaN"
```

While the **threadPtr** function could easily be implemented as an external assembly routine, it is implemented directly by the KPL compiler for efficiency reasons.

initializeThreadPtr

```
external function initializeThreadPtr (p: ptr to ThreadData)
```

This function initializes register **tp**. (The "thread pointer register" is **r12**).

Register **tp** normally points to an object of class **ThreadData**. Register **tp** is initialized whenever a thread is created and does not change.

The **ThreadData** object is where information particular to a thread is kept. An important field is **catchStack** which is used in the processing of the try-throw-catch mechanism.

Programmers should not modify the **catchStack** field or the **tp** register, except during thread initialization.

```
var
  mainThreadData: ThreadData
...
mainThreadData = new ThreadData {
            catchStack = null,
            threadName = "Main Thread",
            threadNumber = 0,
            threadPrefs = & threadPrefs_0 }
initializeThreadPtr (& mainThreadData)
```

The argument to **initializeThreadPtr** should have type "**ptr to** ThreadData".

This function is implemented as an external assembly routine, and is not known to the KPL compiler.

FatalError

```
function FatalError (errorMessage: String)
```

This function is configurable. The default implementation is to throw

ERROR_FatalError

and, if that is not caught by the application, it will print

```
**********  FATAL ERROR: "errorMessage"  **********
```

to invoke **RuntimeExit**, which will terminate execution of all threads and enter the emulator so debugging can commence.

The default implementation can be overridden with code like this:

```
var threadDataPtr: ptr to ThreadData

threadDataPtr = asPtrTo (threadPtr (), ThreadData)
threadDataPtr.fatalErrorFun = MyFatalFunct
```

The reason that **FatalError** exists and is configurable in this way is that errors can arise within code in packages like **System** and **Number**.[27] This code must do something, but the appropriate error handling really depends on the application. While the default behavior will be adequate for most programs, some programs may wish to deal with the error in application-specific ways.

[27] For example, the **appendStrings** function checks for insufficient space and calls FatalError if there is a problem.

Chapter 2: Built-in KPL Functions

Introduction

The KPL compiler recognizes a number of predefined functions, which are sometimes called "built-in functions". These are discussed more fully in

An Introduction to KPL: A Kernel Programming Language

but they are mentioned here. Consult that document for more information.

Some of these functions are built-in to the compiler because they somehow require extra-legal handling.

Some built-in functions can handle many different types. For example, **arraySize** can take an argument of any array type. Other built-in functions take a *Type* — not a *Value* — as an argument. For example, **isInstanceOf** takes a *Type* as its second argument.

Type Casting and Conversions

asByte
asHalfword
asWord

```
function asByte     (i: int) returns byte
function asHalfword (i: int) returns halfword
function asWord     (i: int) returns word
```

These functions check that the argument is within range and return the same value. An Arithmetic Exception will occur is the argument is output range.

forceToByte
forceToHalfword
forceToWord

```
function forceToByte     (i: int) returns byte
function forceToHalfword (i: int) returns halfword
function forceToWord     (i: int) returns word
```

These functions will sign-extend the lower bits to create a value within the given type. If the argument is already within range, it will be unchanged. If not, the value will be altered. There are no possibilities for failure.

forceToDouble
forceToInt

```
function forceToDouble (i: int)    returns double
function forceToInt    (d: double) returns word
```

These functions attempt to represent the same value in a different form. For example, **forceToInt (123.1)** will yield **123**. Likewise, **forceToDouble (123)** will yield **123.0**.

copyBitsToDouble
copyBitsToInt

```
function copyBitsToDouble (i: int)    returns double
function copyBitsToInt    (d: double) returns word
```

These functions change the type of a value without altering the bits. For example, **copyBitToInt (1.75)** will yield **4610560118520545280** (i.e., **0x3ffc_0000_0000_0000**),

asInteger
asPtrTo

```
function asInteger (p: ptr to AnyType) returns int
function asPtrTo   (x:…; Type) returns ptr to Type
```

The **asInteger** function allows the programmer to convert a pointer into a 64 bit integer. The bits are not altered.

With **asPtrTo**, the second argument is a *Type*, not a *Value*. The **asPtrTo** function allows the programmer to convert an integer or a pointer of any type into a pointer to the specified *Type*.

isKindOf

```
function isKindOf (p: …, Type) returns bool
```

The first argument **p** must be either an object or a pointer to an object. The second argument *Type* must name a class or an interface.

At runtime, this function looks at the class of **p** and returns true if it is a subclass of the given *Type*. If the given *Type* is an interface, it checks to see if the class of p implements that interface.

In this example, assume that **Student** is a subclass of **Person**:

```
var p: ptr to Object
…
p = alloc Student { … }
…
if isKindOf (p, Person)
  printf ("This is a Person or Student")
```

```
endIf
```

Implementation: For each occurrence of **isKindOf**, the compiler will insert a call to the assembly function **_isKindOf**.[28] This function will look at the object and follow its DPT to its dispatch table. From there, it will locate the object's class descriptor. It will then run through the table to see if the given class or interface is present, in a fairly tight loop. Thus, there is not much overhead[29] for a use of **isKindOf**.

isInstanceOf

```
function isInstanceOf (p: …, ClassName) returns bool
```

The first argument **p** must be either an object or a pointer to an object. The second argument must name a class.

At runtime, this function looks at the class of **p** and returns true if it is the same as the class named by the second argument *ClassName*.

Implementation: The compiler inserts code directly inline to obtain the DPT of **p**. The code then compares this to a constant, which is the address of the dispatch table for the named class. So this operation is very fast.

sizeOf

```
function sizeOf (Type) returns int
```

This function returns the number of bytes required for a value of the given type.

This value is mandated by the KPL specification and is not implementation-dependent. The sizes of the various types in KPL are:

---

[28] The package **System** contains a second implementation of **isKindOf**, written in KPL, which may be of interest. This function is named **KPLIsKindOf**, and is now commented out.

[29] The **_isKindOf** function requires about 5×**n**+11 machine instructions, where **n** is the number of superclasses and interfaces implemented by the class.

Type	sizeOf (in bytes)
byte	1
halfword	2
word	4
int	8
double	8
bool	1
pointer	8
array	8 + n * **sizeof** (element), rounded up to a multiple of 8
object	8 + …fields…, rounded up to a multiple of 8
struct, union	…fields…

Each array has an 8 byte header field containing the maximum size and the current size of the array. The size of the array is always a multiple of 8, so for arrays with small elements (such as Strings), there will often be padding bytes inserted after the last element.

Each object has an 8 byte header, which is a pointer to the dispatch table for the object's class. For each class, there is only one dispatch table so all objects of a given given point to the same dispatch table. The fields are always laid out in memory in the same order they appear in the class definition. Padding bytes are inserted to make sure that every field is properly aligned, and padding bytes are added to make sure the object's size is a multiple of 8.

With a **struct**, the fields are always laid out in memory in the same order they appear in the **struct** definition. With a **union**, the fields are overlaid. Padding bytes are inserted to make sure that every field is properly aligned. The size of a **struct** or **union** will be either 1 byte, 2 bytes, 4 bytes, or a multiple of 8 bytes, and padding bytes will be added as necessary to achieve this.

initializeArray

```
function initializeArray (arr: ArrayType)
```

This function is used to initialize an array, and every array must be initialized before use. More precisely, this function will initialize the header word that every array has. The maximum size will be initialized according to the type definition, and the

current size will be set to the maximum size. The elements of the array will not be modified.

The argument may be either an array or a ptr to an array, as in:

```
var
  a: array [100] pf int
  p: ptr to array [100] of int
initializeArray (a)
initializeArray (p)
```

In KPL variables are initially set to zeros. An array which has not been properly initialized with **initializeArray** will have zeros in this header. This uninitialized header will cause an error if the array is accessed.

The KPL compiler implements all multi-dimensional arrays as arrays of arrays. In other words, only singly-dimensioned arrays exist. In the example below, the two arrays have the same type and the following is all legal KPL.

```
var
  a: array [10, 20] of int
  b: array [10] of array [20] of int
…
a = b
a [i,j] = b [i,j]
a [i] [j] = b [i] [j]
a [i] = b [i]
```

setArraySize

```
function setArraySize (arr: ArrayType, newSize: int)
```

This function sets the current size of the array **arr** to **newSize**. The array **arr** must already be initialized and the **newSize** must be within 0 … maxSize.

arraySize

```
function arraySize (arr: ArrayType) returns int
```

This function returns the current size of the array. If the array is uninitialized, this function will return 0.

arrayMaxSize

```
function arrayMaxSize (arr: ArrayType) returns int
```

This function returns the maximum size of the array. If the array is uninitialized, this function will return 0.

initializeObject

```
function initializeObject (p: ptr to ClassName)
```

This function will initialize an object. This consists of setting the header field of the object to the address of the class's dispatch table. This function merely stores a pointer to the dispatch table at the address given; it does not affect the other fields of the object in any way.

It is often the case that objects are initialized with the **alloc** or **new** constructs and the **initializeObject** function will not be needed. But there are other situations in which this function is useful.

Normally, objects are allocated on the heap by using the **alloc** construct, which will not only initialize the object's header, but will also initialized the object's fields.

If the heap and pointers are not involved, the programmer will typically use the **new** construct to create objects. The **new** construct will also initialize both the object's header and the fields.

CPUControl
CPUControlUserMode

```
function CPUControl         (expr: int, Integer) returns int
function CPUControlUserMode (expr: int, Integer) returns int
```

The Blitz-64 hardware contains two machine instructions which are intentionally left "undefined". Individual implementations of the Blitz-64 core will elect to use these instructions differently. In some implementations, these instructions may not be used at all.

Here is the form of these machine instructions:

```
CPUControl          RegDest,RegSource,Value16
CPUControlUserMode  RegDest,RegSource,Value16
```

Each instruction takes two register operands and a 16 bit immediate value. The **Value16** field acts as a 16 bit "operation code" that indicates which operation is to be performed. Its precise meaning is left as "implementation dependent" and up to individual core designers. Consult

Blitz-64 Instruction Set Architecture Reference Manual

for details and examples of how these instructions might be used.

These **CPUControl** and **CPUControlUserMode** functions are provided in order to allow programmers to use these instructions.

The first argument may be any integer-valued expression and will be used for the **RegSource** operand. The second argument must be a value within -32,768 … +32,767. This integer will be used as the **Value16** operation code in the instruction. The result placed in **RegDest** will be returned by the function.

Implicitly Inserted Functions

The functions in this section are inserted automatically by the compiler. Normally, they would not be coded by the programmer, but they could be.

ptrToBool

```
function ptrToBool (p: ptr to AnyType) returns bool
```

The **ptrToBool** function returns true if the pointer is not null and false if it is null. It is inserted automatically.

This function allows the following code—which follows a common C/C++ pattern—to work as expected:

```
if p
  printf ("not null")
else
  printf ("null")
endIf
```

upcastToHalfword
upcastToWord
upcastToInt

```
function upcastToByte     (i: byte)                returns halfword
function upcastToHalfword (i: byte/halfword)       returns word
function upcastToWord     (i: byte/halfword/word) returns int
```

These functions are inserted automatically when needed by the compiler and would not normally be used explicitly by the programmer.

They allow an integer value of 8, 16, or 32 bits (i.e., a **byte**, **halfword**, or **word**) to be used in a context requiring a 64 bit integer (i.e., an **int**).

For example, in the following code the value -123 has a type of byte:

```
var i: int
i = -123
```

The compiler implicitly inserts an upcast function, as in:

```
i = upcastToInt (-123)
```

These functions will sign-extend the value. Thus, the value—when viewed as a signed number—will never be altered.

These functions do not require additional machine instructions. For example, consider:

```
i = x
```

If x has type **int**, the compiler might generate:

```
loadd    r1,x
stored   i,r1
```

Whereas, if x has type **byte**, the compiler might generate:

```
loadb    r1,x     # loadB used, not loadD
stored   i,r1
```

This efficiency is achieved because LOADB will sign-extend. All **byte**, **halfword**, and **word** values are kept in sign-extended form when in registers.

In KPL, integer values are normally signed, but occasionally the programmer must work with unsigned numbers of various sizes. In such cases, the upper bits must be explicitly cleared, as in:

```
var
  i: int
  b: byte
…
i = b & 0x00000000000000ff
```

This might be compiled into these instructions:

```
loadb    r1,b
andi     r1,r1,0xff
stored   i,r1
```

upcastToDouble

```
function upcastToDouble (p: word) returns double
```

This function is inserted automatically when needed by the compiler and would not normally be used explicitly by the programmer.

This function is used in code such as:

```
var d: double
…
d = d * 7
```

The value "7" is an integer. It must be converted to a floating point number before the multiplication can occur. The compiler automatically inserts **upcastToDouble** in the above code.

Any 32 signed integer value (i.e., any **byte**, **halfword**, or **word** value) can be represented exactly as a double precision floating point number. So there is never any loss of accuracy or possibility for error.

However, the same is not true of 64 integer values. For an arbitrary **int** value, there is a possibility that accuracy will be lost if the value is either too large or too negative[30]. For this reason, the programmer must explicitly invoke the **forceToDouble** function in code like the following:

```
var
  d: double
  i: int
…
d = d * forceToDouble (i)
```

[30] More precisely, loss of accuracy will not occur if the **int** value is within -9,007,199,254,740,992 ... +9,007,199,254,740,992. Outside this range, loss of accuracy usually happens.

Chapter 3: MiscLib Package

Introduction

The functions in this package are for programs that must read input from the user.

These functions rely on a global variable called **InputBuffer**, which makes them not reentrant. Thus, these functions are *not thread safe!*

All output is direct to stdout and all input is obtained by calls to **SimpleSerial_ReadString**.

At some point in the future, this code needs to be rewritten to

- Read the input using calls to the file system.
- Be made reentrant.
- Allow arbitrary UTF-8 input.

InputBuffer

```
const MAX_INPUT_BUFFER_SIZE = 1000
var InputBuffer: array [MAX_INPUT_BUFFER_SIZE] of byte
```

The **InputBuffer** is a shared variable, which is used for reading in the user's input.

GetInputLine

```
function GetInputLine (prompt: String) returns bool
```

This function prints the **prompt** and reads a line of input. It removes the trailing newline character ('\n').

The result will be in the global variable **InputBuffer**. This function will initialize this array before use.

It prints to **stdout** and reads by calling **SimpleSerial_ReadString**.

This function returns "all okay", i.e., true if there were no errors and false otherwise. The following are considered errors:

- Too many characters are typed, overflowing the input buffer.
- The input contains something besides ASCII characters.

NOTE: This function is not reenterant.

GetYesNo

```
function GetYesNo (prompt: String) returns bool
```

This function prints the **prompt** and reads either "yes" or "no". It returns true for "yes" and false for "no".

This function will accept

"y"	"yes"	"Y"	"YES"
"n"	"no"	"N"	"NO"

This function retries until it gets something correct.

This function calls **GetOneLine**, so it indirectly calls **SimpleSerial_ReadString**.

NOTE: This function is not reenterant.

GetInt

```
function GetInt (prompt: String) returns int
```

The function prints the **prompt**, reads in an integer, and returns it.

The input can be in decimal or hex and whitespace is ignored. If the input is invalid, it will repeat the **prompt** and keep trying until it gets a valid integer.

Hex numbers are indicated with a leading "0x" or "0X" and capitalization of the hex digits is ignored.

This function calls **GetOneLine**, so it indirectly calls **SimpleSerial_ReadString**.

NOTE: This function is not reenterant.

GetOneChar

```
function GetOneChar (prompt: String) returns int
```

This function prints the **prompt** and then reads in a single character, followed by newline and returns the character. It retries until it gets something correct.

This function calls **GetOneLine**, so it indirectly calls **SimpleSerial_ReadString**.

Since it calls GetOneLine, the only valid input is an ASCII printable characters (i.e., " " … "~") followed by a newline "\n".

NOTE: This function is not reenterant.

AppendIntToString

```
function AppendIntToString (prefx: String, i: int) returns String
```

This function is passed a **prefix** and an **int**. It converts the **int** to a decimal string, appends it to the **prefix**, and returns the result as a new String. For example:

Call	Result

```
AppendIntToString ("Label_", 123)     "Label_123"
AppendIntToString ("abc",   -123)     "abc-123"
```

NOTE: This function always allocates a new string on the heap. It frees any other memory it allocates.

Indent

```
function Indent (i: int)
```

This function prints **i** blanks on **stdout**.

PadTo

```
function PadTo (str: String, fieldWidth: int)
```

This functions is passed a string and an integer. It assumes the string has just been printed; it prints as many padding blanks as necessary to ensure that **fieldWidth** characters have been printed.

The blanks are printed on **stdout**.

Chapter 4: Print Package

Introduction

KPL provides an output system similar to C / C++. The following two functions are built in to the language:

```
printf
sprintf
```

In order to use these functions in a package, the package must use "PrintPackage":[31]

```
header MyPack
  uses System, PrintPackage
  ...
endHeader
```

The **printf** and **sprintf** functions are unusual in that they can take a variable number of arguments. All other KPL functions have a fixed number of arguments.

The general form of **printf** is

```
printf (FileID, FormatString, arg1, arg2, arg3, ...)
```

The *FileID* argument must be an identifier. It is optional and defaults to "**stdout**" if missing. These are equivalent:

```
printf (FormatString, arg1, arg2, arg3, ...)
printf (stdout, FormatString, arg1, arg2, arg3, ...)
```

The output can be directed elsewhere, as in:

```
printf (stderr, FormatString, arg1, arg2, arg3, ...)
printf (myFile, FormatString, arg1, arg2, arg3, ...)
```

[31] If a package doesn't invoke **printf** or **sprintf**, then there is no need to use **PrintPackage**.

KPL does not have the "fprintf" function from C / C++; instead the **printf** function is used.

The general form of **sprintf** is

```
sprintf (ID, FormatString, arg1, arg2, arg3, ...)
```

For **sprintf**, the first argument is required and must be an identifier. Furthermore, this ID must have type String, i.e., **ptr to array of byte**. This is where the "output" will be directed to.

```
var
  stringBuffer: ptr to array [1000] of byte = ...
sprintf (stringBuffer, FormatString, arg1, arg2, arg3, ...)
```

For both **printf** and **sprintf**, the *FormatString* argument must be a string constant. It may not be any other expression.

```
printf ("Hello", ...)          -- OK

myString = "Hello"
printf (myString, ...)         -- Compile-time error
```

The **printf** function will send output to the given output channel, with **stdout** being the default output channel.

The **sprintf** function will add bytes to the end of a byte array. The output target (a "string buffer") should be a previously initialized array. It may contain some elements, but should have additional space. In other words, its CURRENT size should be less than its MAXIMUM size. The **sprintf** function will add bytes to the end of the string buffer, increasing its CURRENT size. If the end of the string buffer is reached and an attempt is made to write beyond it, then an error will be thrown.

The output system writes all Unicode characters using the UTF-8 encoding.

The Format String

The *FormatString* is similar to C / C++. It may contain character data (which is written out) and format codes (which are used to control the writing of data values).

```
printf ("val1 = %d  val2 = %d\n", i+4, foo(j))
```

The format code begins with % and continues through the code letter.

Just as in C / C++, for each format code (such as "%d") there must be exactly one argument. The arguments following the FormatString are matched up, in order, with the format code.

KPL also allows a second form, which may be unfamiliar to C / C++ programmers. If the argument is a simple identifier, then the argument may be embedded directly in the string. Parentheses are used for this.

The following two are equivalent:

```
printf ("val1 = %d  val2 = %d  val3 = %d\n", i, foo(j), k)
printf ("val1 = %d(i)  val2 = %d  val3 = %d(k)\n", foo(j))
```

Here are the rules for this second form: The embedded ID form can be be mixed with the traditional form, as shown in the example above. To use the embedded form, the argument being printed must be an ID, not a complex expression, so it can only be used to print constants, parameters, local variables, and global variables. The opening parenthesis must immediately follow the format code character; no intervening spaces are allowed. The ID may be preceded by or followed by any number of blanks. Tabs and newlines are not allowed. The ID and any trailing blanks should be followed by the closing parenthesis. The (and) parentheses are not sent to the output.

The following format codes are recognized:

```
%d     print an int in decimal
%s     print a String
%c     print a single character
%x     print an int in hex
%e     print a double number
%f     print a double number
%g     print a double number
%b     print a bool value
%h     print a halfword value in binary
%w     print a word value in binary
```

```
%i      print a int value in binary
%o      print an object's class name
%%      print %
%(      print (
```

The compiler will check the *FormatString* and each of the format codes. For each, it will verify that the format code is specified correctly. The compiler will also verify that for each format code, there is a corresponding argument of the correct type.

The general form of a format code is:

```
%  [ Flags ]  [ Width  ]  [ . Precision ]  FormatCharacter
```

For example

```
printf ("  %d   %-#20.8x   ", ...)
```

The following *Flag* characters are recognized:

```
-   0   #
```

The *Flag* characters are optional. They may be specified in any order, but there must be at most one occurrence of each. The meaning of "-" is "left-justify within the field". The meaning of the other characters depends on the *FormatCharacter*.

The *Width* is an integer. More precisely, the *Width* is a sequence of decimal digits and it may not begin with a 0. This integer specifies the **field width** in which the data will be printed. Generally speaking, padding bytes are added as necessary to fill out to the full field width.

The *Precision* is an integer, i.e., a sequence of decimal digits. The meaning of *Precision* is dependent on the *FormatCharacter*.

The *FormatCharacter* must be one of the following. We discuss each of these format codes in turn in the following sections.

```
d  s  c  x  X  e  f  g  b  B  h  w  i  o  %  (
```

%d Decimal

Flags:

0 Not allowed

Insert comma separators

- Left-justify the digits within the field

Field Width:

Print the value within a field, adding padding bytes if necessary. If the field is insufficiently large enough, then ignore the width and print the full value.

Precision:

Not allowed

Print the integer value in decimal. If the field width is specified, then the number of characters will be at least that, but may be more.

If the # flag is present, then commas will be added to numbers larger than 999. The actual character to be used is determined by

PrintPreferences.integerSeparator

The default is comma (,) but this may be changed to, for example, a period (.) for European style numbering. The preferences may also indicate no character, in which case the # flag is ignored.

```
         Example          Example          Example
         ==============   ===============  ============================
%d       "0"              "123"            "-9223372036854775808"
%#d      "0"              "123"            "-9,223,372,036,854,775,808"
%12d     "           0"   "     1234567"   "-9223372036854775808"
%#12d    "           0"   "   1,234,567"   "-9,223,372,036,854,775,808"
%-12d    "0           "   "1234567     "   "-9223372036854775808"
```

%s String

This format code is used to print a String. The argument must be a pointer to an array of bytes which contains a sequence of UTF-8 encoded characters.

Flags:

0 Not allowed
Not allowed
- Left-justify the characters within the field

Field Width:

Print the string of characters within a field, adding padding bytes if necessary. If the field is insufficiently large enough, then ignore the width and print the full string.

Precision:

The "precision" number indicates that the string should be truncated. Only print this many of the leading characters in the string.

```
           Example              Example
           ==================   ============================
%s         "hello"              "Now is the time for all good"
%10s       "     hello"         "Now is the time for all good"
%-10s      "hello     "         "Now is the time for all good"
%10.3s     "       hel"         "       Now"
%-10.18s   "hello     "         "Now is the time fo"
```

%c Character

This format code is used to print a single character. The argument must be an integer and this integer is interpreted as a Unicode codepoint. (Note that this is not a UTF-8 encoding of a character.)

```
var i: int = '€'
printf ("char = %c", i)         -- Ok, will print "char = €"

var str: String = "€"
printf ("char = %c", str[0])    -- Wrong; str is UTF-8 encoded
```

Flags:

0 Not allowed
Not allowed
- Left-justify the digits within the field

Field Width:

Print the character within a field, adding padding bytes as necessary. Since the field width must be at least 1, there will never be any truncation needed.

Precision:

Not allowed

```
            Example
            ============
%c          "a"
%5c         "    a"
%-5c        "a    "
```

If the integer value is not a Unicode codepoint, then this error will be thrown:

Not_A_Unicode_Codepoint

If the integer is any value representable as a byte, i.e., within the range -128 ... +127, then it will be sent "as is" to the output. Therefore, any ASCII control character (such as \t, \n, etc.) can be sent to the output.

%x Hex

This format code is used to print an integer value in hexadecimal.

Flags:
- 0 Print leading 0 digits (and F's for negative values).
- # Add a "0x" or "0X" prefix, depending on the format character.
- - Left-justify the digits within the field

Field Width:

Print the value within a field, adding padding bytes as necessary.

Precision:

Truncate the value to this number of characters.

Format Character:
- x Print in lowercase, i.e., 0x0123abcd
- X Print in uppercase, i.e., 0X0123ABCD

Here are examples of the most useful cases:

```
            Decimal 4,779              Decimal -4,779
            ========================   ========================
%#x         "0x12ab"                   "-0x12ab"
%016x       "00000000000012ab"         "ffffffffffffed55"
%04.4x      "12ab"                     "ed55"
```

The # flag adds “0x” in front of the number:

```
             Example                    Example
             ========================   ========================
%x           "123abc"                   "-123abc"
%#x          "0x123abc"                 "-0x123abc"
```

If the format code is “X” instead of “x”, then the number will be in uppercase:

```
             Example                    Example
             ========================   ========================
%X           "123ABC"                   "-123ABC"
%#X          "0X123ABC"                 "-0X123ABC"
```

If a field width is present, the number will be padded as necessary:

```
             Example                    Example
             ========================   ========================
%16x         "           123abc"        "          -123abc"
%#16x        "         0x123abc"        "        -0x123abc"
%-16x        "123abc          "         "-123abc         "
%#-16x       "0x123abc        "         "-0x123abc       "
```

If the field width is inadequate for the value, then the width will be ignored and the full value will be output:

```
             Example                    Example
             ========================   ========================
%3x          "123abc"                   "-123abc"
```

If the 0 “zero fill” flag is present, then the number will be sign-extended by adding leading 0’s to non-negative numbers and F’s to negative numbers. If the flag is not present, negative values will be printed with a “-” sign, followed by the absolute magnitude.

```
             Example                    Example
             ========================   ========================
%16x         "         1234abcd"        "        -1234abcd"
%016x        "000000001234abcd"         "ffffffffedcb5433"
```

If the 0 “zero fill” flag is present, then the number will be sign-extended to 16 digits. Thereafter, padding blanks will be added as needed to reach the field width. If the field width is too small, the width will be ignored and the full value will be printed:

```
             Example                    Example
```

```
             ========================   ========================
%020x        "    000000001234abcd"     "    ffffffffedcb5433"
%-020x       "000000001234abcd    "     "ffffffffedcb5433    "
%#020x       "  0x000000001234abcd"     "  0xffffffffedcb5433"
%#-020x      "0x000000001234abcd  "     "0xffffffffedcb5433  "
%04x         "000000001234abcd"         "ffffffffedcb5433"
```

If a *Precision* value is given, the number will be truncated to that number of digits. The 0 "zero fill" flag must be present.

```
             Example                    Example
             ========================   ========================
%0.11x       "0001234abcd"              "fffedcb5433"
%016.11x     "     0001234abcd"         "     fffedcb5433"
%-016.11x    "0001234abcd     "         "fffedcb5433     "
%04.4x       "0123"                     "ff80"
```

When truncation occurs, only sign-extension digits may be removed. In other words, the value as printed must be unchanged.

```
printf ("%0.4x", 0xffffffffffff8003)  – Ok, prints "8003"
printf ("%0.4x", 0x0000000000007fff)  – Ok, prints "7fff"
printf ("%0.4x", 0x0000000000008003)  – Error
printf ("%0.4x", 0x0012340000007fff)  – Error
```

The reason the third line throws an error is that the argument is a positive number, but "8003" is a negative number, when the 16 bits are interpreted as a two's complement, signed value.

```
8003 = 0x0000000000008003 =  32,771
       0xFFFFFFFFFFFF8003 = -32,765
```

If problems arise, the error **TruncationProblemsWithHex** will be thrown.

%f Floating Point

This format code is used to print a **double** value.

Flags:

0 Not allowed.

Add the prefix and postfix to the output string.

- Left-justify the output string within the field

Field Width:

Print the output string within a field, adding padding bytes as necessary. If the width is too small, then it is ignored and the entire output string is printed.

Precision:

Print exactly this number of digits.

The formatting algorithm is complex, but if the simplest formatting is specified:

```
%f
```

the following outputs are possible:

```
"12.35"
"123.0"
"1.2340000000000000e+20"
"1.234e-10 (inexact)"
```

A *Field Width* may be specified and the "-" *Left-Justification Flag* may be used. If the *Field Width* is inadequate, it will be ignored and the print string will be as long as necessary.

```
            Example
            ====================
%10f        "     12.35"
%-10f       "12.35     "
%10f        "1.2345678901234e+10"
```

The formatting is guided by the following preferences in **threadPrefs.printPrefs**:

```
PrintPreference       Default value       Reasonable options
==================    =================   ==================
numberPrefix          null                "$ "
numberPostfix         null                " USD"
floatDecimalPoint     '.'                 ','
floatSeparator        ','                 '.', '_', 0
floatSeparator2       '_'                 ',', ' ', 0
floatPosInf           "<pos infinity>"    "+∞"
floatNegInf           "<neg infinity>"    "-∞"
floatNaN              "<not-a-number>"    "NaN"
floatExp              "e"                 " × 10 ^"
floatInexactPostfix   " (inexact)"        null
floatPositiveSign     0                   '+'
```

```
floatExpPlusSign      '+'                  0
```

The programmer can change these values with code such as this:

```
var prefs: ptr to PrintPreferences
prefs = asPtrTo (threadPtr (), ThreadData).threadPrefs.printPrefs
prefs.numberPrefix  = "$ "
```

If the # flag is present, then the **numberPrefix** string will be prepended to the front of the output string and the **numberPostfix** string will be appended to the end of the output string. These strings default to nothing, but, if the preference **numberPrefix** value is changed to "$ " for example, the following output could be produced:

```
"$ 12.35"
```

If the preference **numberPostfix** value is changed to " € ", the following output could be produced:

```
"12.35 €"
```

Given the default values for **floatDecimalPoint** and **floatSeparator**, the following output could be produced:

```
"12,345,678.0"
```

These could be changed to produce European-style output:

```
"12.345.678,0"
```

Or the separator could be eliminated:

```
"12345678.0"
```

The **floatSeparator2** character is used to the right of the decimal point. If changed to a blank, numbers would be formatted like this:

```
"1,234.567 890 12"
"3.000 000 001 58e-34"
```

The **floatExp** string could be changed to " × 10 ^ ", giving output like this:

```
"6.022 × 10 ^ -23"
```

The **printf** functions will append the **floatInexactPostfix** to a number whenever the output produced by the algorithm is not the exact same value as the **double** value. This preference value can be changed to **null** to avoid this.

```
"6.022e-23 (inexact)"   -- The default style
"6.022e-23"             -- After changing floatInexactPostfix
```

By default, the leading plus sign is not printed, but this can be changed by modifying the **floatPositiveSign** preference.

```
"123.45"         -- The default style
"+123.45"        -- After changing the pref
```

The printing of not-a-number (NaN) values, positive infinity and negative infinity can be changed. The default output strings are:

```
"<not-a-number>"
"<pos infinity>"
"<neg infinity>"
```

The zero values will printed as:

```
"-0.0"
 "0.0"       -- Default
 "+0.0"      -- After changing floatPositiveSign to "+"
```

Otherwise, the output string will be produced in one of these two forms:[32]

```
"123.456"        -- Will use this form if reasonable
"1.23456e+23"    -- Will use this otherwise
```

If *Precision* is present, then exactly that many digits will be given in the print string. Otherwise, the **printf** algorithm will choose the number of digits based on the value to be printed.

[32] The printing algorithm will make the determination of which form to print the number in, based on how many zeros would be printed.

%e Floating Point - Exponential Form
%g Floating Point - General Form

This format code is used to print a **double** value.

Flags:
- 0 Not allowed.
- # Add separators.
- - Left-justify the output string within the field

Field Width:
Print the output string within a field, adding padding bytes as necessary. If the width is too small, then it is ignored and the entire output string is printed.

Precision:
If present, round to this number of digits. If absent, default to 16 for **%g** and 17 for **%e**.

If **%e** is used, the output will always be in exponential form, such as:

```
1.234e5
6.25e-2
1.23e-100
```

If **%g** is used, the output of many values will be rendered in a more human-friendly form, while others will still be in exponential form. For example:

```
123400.0
0.0625
1.23e-100
```

If *Precision* is specified, then the number will be rounded to a value with exactly that many digits. Trailing zeros to the right of the decimal point will be removed and not printed.

The default *Precision* with **%e** is to round to 17 digits, which will cause each floating point value to be printed differently.

With floating point, a difficulty is often caused by the fact that some numbers cannot be represented exactly with **double** values and when rounded, all 17 digits are needed. For example:

```
printf ("%e", 0.3)
```

will print:

```
2.9999999999999999e-1
```

The default *Precision* with **%g** is to run to 16 digits, which gives more readable output, and eliminates this problem. For example:

```
printf ("%g", 0.3)
```

will print:

```
0.3
```

There are a number of options that are controlled by the current values in "print preferences". Here are the things that can be changed:

- Is a leading "+" printed for positive numbers?
- Is a leading "+" printed after the "e" for positive exponents?
- What character is used for the decimal point?
- What character or string is used to indicate "times 10 to the power"?
- Are separators included in the output and what characters are used?
- How are these values rendered?
  - +inf
  - -inf
  - nan
- Will inexact values be flagged with a postfix, such as " (inexact)"?

The formatting is guided by the following preferences in **threadPrefs.printPrefs**:

PrintPreference	Default value	Reasonable options
numberPrefix	***<ignored>***	
numberPostfix	***<ignored>***	
floatDecimalPoint	'.'	','
floatSeparator	','	'.', '_', 0
floatSeparator2	'_'	',', ' ', 0
floatPosInf	"<pos infinity>"	"+∞", "inf"
floatNegInf	"<neg infinity>"	"-∞", "-inf"
floatNaN	"<not-a-number>"	"NaN"

```
floatExp                "e"                 "E" or "×10^"
floatInexactPostfix     " (inexact)"        null
floatPositiveSign       false               true
floatExpPlusSign        true                false
```

The programmer can change these values with code such as this:

```
var prefs: ptr to PrintPreferences
prefs = asPtrTo (threadPtr (), ThreadData).threadPrefs.printPrefs
prefs.floatPosInf  = "+∞"
```

If **%e** is used, then the **floatInexactPostfix** will be appended to the number whenever the output produced by the algorithm is not the exactly the same value as the **double** value. This preference value can be changed to **null** to avoid this.

```
"6.022e-23 (inexact)"   -- The default style
"6.022e-23"             -- After changing floatInexactPostfix
```

If **%g** is used, then the **floatInexactPostfix** will be ignored and will never be appended.

By default, the leading plus sign is not printed, but this can be changed by modifying the **floatPositiveSign** preference.

```
"1.23e+2"          -- The default style
"+1.23e+2"         -- After changing the pref
```

The printing of not-a-number (NaN) values, positive infinity and negative infinity can be changed. The default output strings are:

```
"<not-a-number>"
"<pos infinity>"
"<neg infinity>"
```

The zero values will printed as:

```
"-0.0"
"0.0"       -- Default
"+0.0"      -- After changing floatPositiveSign to "+"
```

A *Field Width* may be specified and the "-" *Left-Justification Flag* may be used. If the *Field Width* is inadequate, it will be ignored and the print string will be as long as necessary.

```
                Example
                ======================
    %10e        "   1.23e+2"
    %-10e       "1.23e+2   "
    %10e        "1.2345678901234567e+10"
```

%b Boolean

This format code is used to print a **bool** value. Basically, the value is printed as either "true" or "false".

Flags:

0 Not allowed.
Suppresses the printing of **false**.
- Left-justify the output within the field

Field Width:

Print the value within a field, adding padding bytes as necessary. If the width is too small, then it is ignored and the entire value is printed.

Precision:

Truncate the value to this number of characters.

Format Character:

b Print in lowercase, e.g., "true"
B Print in uppercase, e.g., "TRUE"

Here are some examples:

```
                Example         Example
                =============   =============
    %b          "true"          "false"
    %#b         "true"          ""
    %8b         "    true"      "   false"
    %-8b        "true    "      "false   "
    %2b         "true"          "false"
    %8.1B       "       T"      "       F"
    %-8.1B      "T       "      "F       "
    %#8.1B      "       T"      "        "
    %#8b        "    true"      "        "
    %#-8b       "true    "      "        "
    %#2b        "true"          "  "
```

%h Halfword
%w Word
%i Integer

These format codes are used to send bytes directly to the output. The bytes are uninterpreted. That is, the binary value is sent directly.

These codes can be useful in writing output to a non-text file. In many cases, it is easier to read and write arbitrary integer values as 8 byte binary values rather than as decimal values of uncertain length and format. However, such files are not human-readable.

The %h (halfword) format code will send 2 bytes (16 bits) to the output. The corresponding argument should be an integer within the following range, i.e., any 16 bit value.

```
0x8000 ... 0x7fff
-32,768 ... +32,767
```

The %w (word) format code will send 4 bytes (32 bits) to the output. The corresponding argument should be an integer within the following range, i.e., any 32 bit value.

```
0x80000000 ... 0x7fffffff
-2,147,483,648 ... +2,147,483,647
```

The %i (integer), format code will send 8 bytes (64 bits) to the output. The corresponding argument should be an integer. Any **int** value is acceptable.

Flags:
- 0 Not allowed.
- # Not allowed.
- - Not allowed.

Field Width:
Not allowed.

Precision:

Not allowed.

For example:

```
printf (myFile, "%h", 0x6162)
printf (myFile, "%h", 24930)   -- Equivalent
```

will send 16 bits to the output. If interpreted as a text file, this output will print as:

```
"ab"
```

There is no corresponding format code for 1 byte (8 bits), because the %c (character) format code will work fine. To send any 8 bit value, i.e., any value within the following range

```
0x80 ... 0x7f
-128 ... +127
```

use something like:

```
printf (myFile, "%c", 0x0f)
```

%o Object

This format code is used to print an object's class. The corresponding argument must be a pointer to an object.

Flags:

0 Not allowed.

Prefix the class name with "a" or "an".

- Left-justify the output within the field

Field Width:

Print the class name within a field, adding padding bytes as necessary. If the width is too small, then the width is ignored and the entire class name is printed.

Precision:

Not allowed.

Here are examples:

```
                Example
                =========================
%o              "Person"
%15o            "          Person"
%-15o           "Person          "
%3o             "Person"
```

The # prefix flag will cause “a” to be added to the class name. If the class name starts with a vowel then “an” is used instead.[33] No space is added.

```
Person → aPerson
Employee → anEmployee

                Example                     Example
                ======================      ========================
%#o             "aPerson"                   "anEmployee"
%#15o           "         aPerson"          "      anEmployee"
%#-15o          "aPerson         "          "anEmployee      "
```

If the pointer is **null**, then “< null >” is printed.

This format code might be adequate for some tasks like debugging, but often the programmer will want to know more than just the object’s class. If the class implements a “printString” message, something like this might be more useful:

```
printf ("obj = %s", p.printString ())
```

%% Percent
%(Parenthesis

Sometimes it is desirable to output the “%” character. Since % is used to signal a format code, it must be doubled to avoid this interpretation. We can view %% as a special format code which just prints “%”.

For example:

[33] Vowels are defined here as a, e, i, o, u, A, E, I, O, U. This works reasonably well. However some people prefer to say things like “an heir” or “a Opossum”, since they do not pronounce the first letter.

```
printf ("Percent = \"%%\"\n")
```

Sometimes it is desirable to print a value and immediately follow it with a "(" parenthesis. In order to prevent the compiler from interpreting the "(" as indicating an embedded identifier, the "(" must be escaped:

```
printf ("value = %d(i)")              -- OK; embedded ID is used.
printf ("value = %d(decimal)", i)     -- Error
printf ("value = %d%(decimal)", i)    -- Must escape the "("
```

The %(is not widely used, since inserting an extra space will usually be acceptable if not preferable.

```
printf ("value = %d (decimal)", i)    -- Probably preferable
```

Flags:

0 Not allowed.
Not allowed.
- Not allowed.

Field Width:

Not allowed.

Precision:

Not allowed.

Implementation

The compiler will parse **printf** and **sprintf** statements and transform them into calls to simpler functions. These simpler functions are implemented in **PrintPackage** and you can have a look at the code of that package for more detailed documentation.

For example, consider this KPL code:

```
printf ("example  %s  %d  \n", str, i)
```

The KPL compiler will transform this, as if the programmer had written this instead:

```
f_print_begin (stdout)
f_print_s (stdout, false, -1, -1, "example  ")
```

```
f_print_s (stdout, false, 10, -1, str)
f_print_s (stdout, false, false, -1, "  ")
f_print_d (stdout, true, false, -1, i)
f_print_s (stdout, false, -1, -1, "  \n")
f_print_end (stdout)
```

The purpose of the functions **f_print_begin** and **f_print_end** is to perform low-level locking and unlocking in order to make the entire sequence atomic.

The **f_print_s** function is defined as

```
function f_print_s (fileID: ptr to FILE,
                    leftJustify: bool,
                    fieldWidth: int,
                    truncateTo: int,
                    str: String)
```

The **f_print_d** function is defined as

```
function f_print_d (fileID: ptr to FILE,
                    addCommas: bool,
                    leftJustify: bool,
                    fieldWidth: int,
                    value: int)
```

There are other similar functions for the other format codes.

In the course of parsing the *FormatString*, the compiler will perform error checking to make sure things like the *Flags* and *Precision* are appropriate for the given *FormatCode* and that, for each *FormatCode*, there is a corresponding argument of the correct type.

For the *FormatCodes* **%e**, **%f**, and **%g** — which handle floating point values of type **double** — the corresponding functions (such as **f_print_f**, …) are not located in **PrintPackage**. Instead, they are found in the package named "**Number**".

If *FormatCodes* **%e**, **%f**, or **%g** are used, the programmer must include **Number** in the "uses" clause. If these *FormatCodes* are not needed, then the **Number** package need not be used.

Unicode and UTF-8

Recall that with Unicode, each character has a numerical **codepoint**. The codepoint is an integer greater than or equal to 0. For example, 'a' is 97 and '😀' is 128512.

In KPL, individual characters are represented with **int** values, so the following are both legal KPL statements and set the variable to the exact same value:

```
i = '😀'
i = 128512
```

Using a 64 bit integer for each character is inefficient for long strings, so UTF-8 is widely used. With UTF-8 each character is represented with 1 to 4 bytes.

PrintPackage provides two useful functions for converting a codepoint to/from its UTF-8 encoding.

MAX_ALLOWABLE_CODEPOINT

```
const MAX_ALLOWABLE_CODEPOINT = 0x10FFFF
```

This number comes from Unicode and is 1,114,111 in decimal.

Note that:

2^{21} = 2,097,152 = 0x200000

Although the UTF-8 scheme can naturally encode any 21-bit binary number, Unicode codepoints are limited to the above range.

ToUTF8

```
function ToUTF8 (p: ptr to byte, codepoint: int) returns int
```

This function translates a single Unicode character into its UTF-8 encoding. It is passed the codepoint to be translated and a pointer to memory where the encoded bytes will be placed. This function will store 0-4 bytes at this memory address and return the number of bytes stored.

The codepoint must be a legal Unicode codepoint. In other words, it must be in the range

hex	decimal
0x0 … 0x0010_FFFF	0 … 1,114,111

The translation will be 1 to 4 bytes in length. This function will return the number of bytes in the UTF-8 encoding. If the codepoint is not valid, this function returns 0.

FromUTF8

```
function FromUTF8 (ch: byte, soFar: int) returns int
```

This function can be used to decode a UTF-8 string and extract codepoints. It is passed:

ch the next byte of input
soFar information accumulated from previous bytes

Initially, it should be called with **soFar** = 0 and is called successively on the bytes in the UTF-8 string.

This function returns:

≥ 0	The Unicode codepoint, if this byte completes a valid encoding.
-1	A UTF-8 encoding error has occurred
< -1	The value of **soFar** to be used for the next call

Here is how to use this function:

```
soFar = 0
while
  ch = …Get the next byte…
  soFar = FromUTF8 (ch, soFar)
```

```
    if soFar >= 0
      ...Got a codepoint in soFar; do something with it...
    elseIf soFar == -1
      ...Got an error; print a message...
      soFar = 0
    else
      -- In the middle of encoding; keep going.
    endIf
  endWhile
```

NOTE: This function is reentrant. (The entire state is contained in **soFar**.)

NOTE: Codepoints can be encoded with more bytes than necessary. For example, codepoint=0 is normally encoded in 1 byte, but it can also be encoded with 4 bytes. (It would be `\xf0\x80\x80\x80`.) This function will not flag this as an error, although it might be considered an error by some people/programs.

NOTE: The encoded bit pattern must be within 0 ... MAX_ALLOWABLE_CODEPOINT, or else this function will consider it an error.

Chapter 5: HostInterface Package

Introduction

The **System** package contains many core functions needed by any KPL program and it must be included in every KPL program. Among other useful helper functions, **System** contains functions related to error handling and the printing of messages.

However, the handling of input/output and communication with the outside world will be different on different computers. For example, a laptop might display messages on a screen, a computer in a robot might use a small led display for messages, and an embedded computer might send output over a debugging cable to a diagnostic support machine.

Each Blitz program is running on a different machine in a different environment with different ways of interacting with the outside world. The purpose of the **HostInterface** package encapsulate the details of communication with the particular environment and the ways in which the I/O is processed for that environment.

For the Blitz system running on laptop, one version of **HostInterface** is used. For the Blitz system running within a robot, a different version of **HostInterface** would be used. And for the system running in an embedded device, yet a third version of the **HostInterface** package would be needed. The benefit of isolating host/envornment dependencies in the **HostInterface** package is that all these systems can use the same version of the **System** package.

The idea is that if a function is unchanged, regardless of the environment, it should be included in **System**. But if the function's implementation depends on environment-specific details, it should be placed in **HostInterface**.

For example, the function to append two Strings is identical, regardless of the environment, so it is placed in **System**. On the other hand, the function to print or display a String will be implemented differently for different environments, so it will be placed in **HostInterface**.

Another benefit of this division of labor between **System** and **HostInterface** is that porting the Blitz system from one environment to another—that is, implementing Blitz on new hardware—reduces to the task of modifying or rewriting **HostInterface**. We strive to remove all hardware dependencies from **System**, so that all Blitz programs can share an identical copy of **System**.

Basic Environments

Initially, we envision these environments:

- Emulated, running under Mac OS X
- Standalone, running on a single board computer

The initial software development was done within the traditional, Unix-based environment.[34] All Blitz code was executed by the Blitz emulator, which is a program named "**blitz**". The emulator runs as a normal Unix program and, as such, makes Unix system calls.[35]

Before developing the Blitz kernel, it was important to make sure the Blitz programming system (the compiler, assembler, linker, and emulator) were up to the task of serious software development. The biggest Blitz program in existence is the KPL compiler, a Unix program written in C++. If this program could be ported to Blitz

[34] This would be my Mac laptop.

[35] Perhaps I should use the term "POSIX-complaint OS" rather than "Unix". Mac OS X is POSIX-compliant, while Android and Linux are mostly POSIX-compliant. Within Windows there are several options for POSIX compatibility. The subject of POSIX-compliance is complicated and filled with technicalities. Simply put, the Blitz software has not been run on any other POSIX-compliant OS. I prefer the term "Unix system" so that I don't suggest a greater degree of precision than exists.

—that is, if the KPL compiler can be re-written in KPL, compiled, and executed on the emulator—then we can conclude that the Blitz program development approach works. At this date, the KPL compiler exists in two versions. One version is written in KPL and this compiler is called **kpl**. The other version is written in KPL and, for the time being to avoid confusion, is called **kpl2**. The version written in KPL runs under the emulator and functions identically to the version written in C++. The assembler has also been ported to KPL and can be run with the emulator.

The compiler and other tools interface with the OS in the following ways:

- The source files are opened and read.
- The output files are opened (and created and destroyed), and written to.
- File error information from the host OS is retrieved.
- The date and time are retrieved.
- Output to stdout is used for listings, etc.
- Output to stderr is used for error messages.
- Command line arguments are retrieved.

By encapsulating this functionality in HostInterface, programs like the compiler and other tools need not know what OS they are running under.

Ultimately, the compiler and other development tools will run under a Blitz-based OS, which will provide these capabilities. But until then, HostInterface will simply pass these requests through to the Unix host OS, which will provide this functionality.

Although POSIX-compliance provides a myriad of ways for a program to communicate with the OS, the above list pretty much covers what is required in order to support Blitz code development. These are the sorts of things that, at a minimum, must be handled in **HostInterface**.

By the way, the emulator has these additional interactions with the OS:

- Control-C interrupts need to be accommodated.
- Raw serial I/O needs to be accommodated.
- Access to floating point exception flags (overflow, invalid, …) is required.

The emulator runs on the host Unix system and provides the illusion of real Blitz hardware. But since the emulator does not run under or within the Blitz system, there is no need to accommodate those functions within **HostInterface**.

So we envision two versions of the HostInterface package:

- Version 1 passes the requests through to the Unix-based host OS
- Version 2 passes the requests though to the Blitz OS.

In this chapter, we are discussing only the version of **HostInterface** that runs under a Unix host OS.

At this writing, the Blitz OS does not exist. Only version 1 of **HostInterface** exists. Only the bare minimum of functionality needed to support the programming tools (such as the **kpl2** compiler, the assembler, and the linker) are implemented.

Below, we first document the functionality provided by **HostInterface**. Then we discuss the mechanism by which these requests are passed through to the host OS.

The Environment for HostInterface

The **HostInterface** package does not use any other packages. As such, the code within it cannot call any of the functions provided by **System** or **PrintPackage**. This includes many of the familiar functions that are used in KPL code almost without thinking.

initializeHostInterface

```
function initializeHostInterface ()
```

The application's **main** function should call this function before using any other functions. Currently, this function does nothing, since there is nothing to do.

Upon program startup, the function **KPLSystemInitialize** within **System** will be invoked before the **main** function is called. This will initialize the heap mechanism, although nothing will be allocated on the heap. Then, **KPLSystemInitialize** will invoke **initializeHostInterface**, which is a function within **HostInterface**. Finally,

when this returns, **KPLSystemInitialize** will terminate and the **main** function will begin executing.

The application program may subsequently decide to change the heap management algorithm. Therefore, **initializeHostInterface** must not allocate anything on the heap. If the application programs changes the heap management algorithm, any chunk of memory allocated within **initializeHostInterface** will be lost.

If the **main** function is going to change the heap management algorithm, it must do it before any files are opened or the command line arguments are examined. The following functions from **HostInterface** allocate memory, so the application *must not call these before changing the heap management algorithm*:[36]

fopen
hostArgs
hostDate

Since these functions must allocate space on the heap, the function **MemoryAlloc** must be accessible in **HostInterface**. **MemoryAlloc** is implemented in assembly code in **runtime.s** and performs an up-call to a heap function from the **System** package. None of the functions in **HostInterface** free any memory, so there are no calls to **MemoryFree**. It is the responsibility of the application code to free objects, if that is required.

Normally, **HostInterface** does not do any printing or user interaction, so there is no need for access to functions like the following and they are not available within **HostInterface**:

print
printString
printIntVar
printDecimal
readString

HostInterface provides two pathways for its users to communicate with the outside world:

[36] In the future, other functions that allocate memory may be added to **HostInterface**. Any application program that changes heap management any place other than at the beginning of **main** better be careful to check about this.

SimpleSerial I/O
Host OS file system (stdin, stdout, stderr)

The **SimpleSerial** device is documented as part of the Instruction Set Architecture, and specifies a minimal connection to a serial device. Even the most minimal Blitz system can use this pathway to communicate with the outside. Typically the **SimpleSerial** device correspond to “Tx" and “Rx" digital I/O pins, similar to what exists on many single board computers like the Arduino. No file system or advanced OS kernel is required for use of the **SimpleSerial** pathway.

The second pathway will be familiar to anyone who already uses the Unix file system.

The SimpleSerial Functions

The **HostInterface** packages provides these functions:

SimpleSerial_PrintChar
SimpleSerial_PrintString
SimpleSerial_ReadString

These functions are implemented in assembly code and are found in **runtime.s**.

There is no additional code in the version of **HostInterface** discussed here. In a future version intended to run under the Blitz OS, the situation may be different. Calls to these functions may be re-routed to the file system in such a future version of **HostInterface**.

SimpleSerial_PrintChar

```
external SimpleSerial_PrintChar (ch: int)
```

This function is passed a character and it sends it to the output. Only the low-order 8 bits are used; the upper bits are ignored. For UTF-8 encoded Unicode text, this function should be called once for each byte, i.e., 1 to 4 times for each “character”.

SimpleSerial_PrintString

```
external SimpleSerial_PrintString (msg: String)
```

Recall that a variable of type String contains a pointer to an array of bytes.[37] This function sends these bytes to the output. It is functionally equivalent to calling **SimpleSerial_PrintChar** repeatedly, but presumably faster.

Recall that arrays in Blitz have both a current size and a maximum size. This prints characters up to the current size. Often, the string will end with the newline character, which is \n.

SimpleSerial_ReadString

```
external SimpleSerial_ReadString (buffer: String)
```

This function is passed a pointer to an array of bytes. The current size of the array is ignored and the maximum size limits how how many bytes can be read in.

Presumably, the user will type a line of characters and hit ENTER/RETURN. Regardless of where the characters come from, the buffer will be filled with UTF-8 encoded characters, with the newline \n as the last byte added.

The bytes beyond the newline, up to the maximum size of the array, are undefined and may be modified by this function.[38]

If the buffer is not large enough to accommodate the input, the final \n and as many characters as necessary will be dropped.

---

[37] Presumably, this is a UTF-8 encoded strings of characters, but not necessarily.

[38] Actually, the emulator will always add \0 after the last character, but it will be added just beyond the CURRENT array size. So at most, the CURRENT array size will be set to MAXSIZE-1, rather than MAXSIZE as you might expect. Normally, Blitz code respects the CURRENT size, so this \0 will be ignored by application code.

Misc Functions

hostArgs

```
function hostArgs () returns String
```

This function returns a String containing the command line arguments. For example:

```
"-p -o filename"
```

When run under the emulator, a Blitz program will get these arguments from the command line that was used to start the emulator. For example:

```
% blitz -g MyProgram.exe -args "-p -o filename"
```

hostDate

```
function hostDate () returns String
```

This function returns the current date and time in the form used in this example:

```
"Thu Apr 30 17:10:25 2020"
```

Accessing the Host File System

The general goal is that the Unix/Linux/POSIX functionality is mirrored. Each of the functions described in this section simply invokes the corresponding function on the host system.

One important difference with Unix is that if the **errno** variable may be set, then it will be set in Blitz. With Unix, the **errno** variable is often set if there is an error but unchanged if there was no error.

errno

```
var errno: int
```

The package defines this global variable. Upon errors in any of the file functions, errno will be set to a numeric code that corresponds to the error.

Unlike Unix, **errno** is always set, even if there is no error. Any function that may set **errno** will always set **errno**. The condition of "no error" is assigned the value of 0.

The following constants are defined in this package and should be used instead of the integer values.[39]

[39] The intent is that the error names, the numeric codes, and the meanings/descriptions of the error conditions are the same as in the underlying host OS. For example, EINVAL is defined as 22.

EPERM	Operation not permitted
ENOENT	No such file or directory
EIO	Input/output error
EBADF	Bad file descriptor
ENOMEM	Cannot allocate memory
EACCES	Permission denied
EFAULT	Bad address
EEXIST	File already exists
ENOTDIR	Not a directory
EINVAL	Invalid argument
EMFILE	Too many open files
ENAMETOOLONG	File name too long

FILE

This package defines

```
type FILE = struct … endStruct
```

Objects of type **FILE** are used to identify files to be read from and written to and are normally referred to with pointers, as in:

```
var myFile: ptr to FILE
…
myFile = fopen ("hello.txt", "r")
if myFile == null
  perror ("Problems")
else
  ch = fgetc (myFile)
  …
```

stdin, stdout, stderr

This package defines the following, which can be used without being opened first:

```
var
```

```
stdin:  ptr to FILE = …
stdout: ptr to FILE = …
stderr: ptr to FILE = …
```

useful constants

This package defines the following useful constants:

```
const
  EOF = -1

  EXIT_FAILURE = 1
  EXIT_SUCCESS = 0
```

and the following values, which are used by the **fseek** function:

```
const
  SEEK_SET = 0
  SEEK_CUR = 1
  SEEK_END = 2
```

fopen

```
function fopen (filename: String,
                mode: String)
                    returns ptr to FILE
                        [ Max_Stack_Usage = 184 ]
```

Every file must be opened before being read or written. A new **FILE** object is allocated on the heap and a pointer to it is returned. This **FILE** object is then used in the read and write operations.

The **mode** argument follows the Unix conventions:

"r" The file must already exist. The initial position will be at the beginning. Only reading is allowed.

"r+" The file must already exist. The initial position will be at the beginning. Both reading and writing are allowed.

"w" Create file or truncate it to zero length if it already exists. Only writing is allowed.
"w+" Create file or truncate it to zero length if it already exists Both reading and writing are allowed.
"a" Create file if necessary, otherwise position it at file end. Only writing is allowed.
"a+" Create file if necessary, otherwise position it at file end, Both reading and writingare allowed. Note with "a+": Check host differences on initial position for reading.

If there is any error, null is returned and **errno** is set to indicate the error.

The following things cause errors:

- The filename is null.
  ENOENT: “No such file or directory”
- The filename is longer than HOST_DEVICE_BUFFER_SIZE.
  ENAMETOOLONG: “File name too long”
- The **mode** is bad.
  EINVAL: “Invalid argument”
- The host OS had an error.
  ENOENT: “No such file or directory”
  EACCES : “Permission denied”
  ENOMEM: “Memory alloc failed”

This function allocates space on the heap and may also throw heap-related errors.

fclose

```
function fclose (fileID: ptr to FILE)
                      [ Max_Stack_Usage = 40 ]
```

All I/O operations are first completed and buffers are flushed. Then the file is closed and the FILE object is freed.

The **errno** variable will be set.

The **fileID** argument must not be null..

remove

```
function remove (filename: String) returns int
                    [ Max_Stack_Usage = 48 ]
```

The given filename is removed and deleted.

The **errno** variable will be set. On success this function returns 0 and **errno** = 0. On failure, this function returns -1 and **errno** ≠ 0.

If the file doesn't exist, then **errno** is set to

ENOENT: "No such file or directory"

feof

```
function feof (fileID: ptr to FILE) returns bool
                    [ Max_Stack_Usage = 56 ]
```

This function returns true if at EOF.

NOTE: This functions follows the host, so true may not be returned until after the program tries to read past the end.

fgetc

```
function fgetc (fileID: ptr to FILE) returns int
                    [ Max_Stack_Usage = 48 ]
```

This function returns the next byte from the file/stream, as a value in the range 0 ... +255.

Any attempt to read beyond the end of the file/stream returns EOF, which is -1.

Continuing to call this function after EOF is not an error.

The **errno** variable will be set. On success this function returns 0 … 255 and **errno** = 0. On failure, this function returns EOF and **errno** will indicate the cause.

fputc

```
function fputc  (ch: int, fileID: ptr to FILE) returns int
                      [ Max_Stack_Usage = 168 ]
```

Writes the byte **ch** to the file/stream. Only the least significant 8 bits are used, so it can be viewed either as a number in the range 0 ... +255 or the range -128 ... +127.

The **errno** variable will be set. On success this function returns the byte in the range 0 … 255 and **errno** = 0. On failure, this function returns EOF and **errno** will indicate the cause.

ungetc

```
function ungetc (ch: int, fileID: ptr to FILE)
        [ Max_Stack_Usage = 40 ]
```

This will clear the EOF condition, and future "getting" will return the pushed back character(s) before resuming getting bytes from the file/stream.

Any attempt to push EOF (i.e., -1) back onto the file/stream is ignored. It is legal to push back bytes, even after you have reached EOF.

This function follows POSIX semantics. Per POSIX, this will not modify the underlying file/stream. Only one character of pushback is guaranteed, but usually more can be pushed back.

Do not to rely on multiple pushbacks!

The **errno** variable will be set. An attempt to pushback something besides -1, 0 ... 255 will set **errno** to EINVAL.

perror

```
function perror (str: String)
                      [ Max_Stack_Usage = 176 ]
```

This function checks **errno** and prints

str*: *message

on **stderr** where **message** is several words describing the error.

If **str** is null, then it just prints the message without "**:** ".

If **errno** is 0, it prints

***str*: Okay, no error**

Any errors from the printing itself will be ignored and **errno** will be unchanged by this function.

fread1

```
function fread1  (buffPtr: ptr to void,
                  byteCount: int,
                  fileID: ptr to FILE)
                      returns int
                          [ Max_Stack_Usage = 56 ]
```

This function reads bytes from the file/stream and stores them at the location in memory given by **buffPtr**, up to the number of bytes specified by **byteCount**.

This function returns the number of bytes successfully read (even if errors happen) and the count may be zero. When EOF is reached, **errno** will be zero; the returned count may be less than the requested **byteCount**.

There may be several reasons that the bytes are not all read, and one reason is that there was an address mapping problem. If there would be a TLB fault, an exception will not occur. Instead, the I/O will just stop. This could be before any bytes were read or it could be mid-way through the read, yielding a non-zero return value. Perhaps this can be cured by just storing a byte into the next address in the buffer, which will cause the fault directly. But beware! Even if a byte could be stored ok, there might be interrupts and/or thread switching and the call might still fail again on the next attempt without processing any bytes.

A **byteCount** of 0 is allowed.

fwrite1

```
function fwrite1 (buffPtr: ptr to void,
                  byteCount: int,
                  fileID: ptr to FILE)
                      returns int
                          [ Max_Stack_Usage = 56 ]
```

This function writes the number of bytes specified by **byteCount** to the file/stream from the location in memory given by **buffPtr**.

This function returns the number of bytes successfully written (even if errors happen) and may be zero. The returned value will be nonzero if and only if the returned value is not equal to the desired **byteCount**.

There may be several reasons that the bytes are not all written, and one reason is that there was an address mapping problem. If there would be a TLB fault, an exception will not occur. Instead, the I/O will just stop. This could be before any bytes were written or it could be mid-way through the write, yielding a non-zero return value. Perhaps this can be cured by just loading a byte from the next address in the buffer, which will cause the fault directly. But beware! Even if a byte could be loaded ok, there might be interrupts and/or thread switching and the call might still fail again on the next attempt without processing any bytes.

A **byteCount** of 0 is allowed.

fseek

```
function fseek  (fileID: ptr to FILE,
                 offset: int,
                 whence: int)
                     [ Max_Stack_Usage = 40 ]
```

The whence argument should be either:

1 = SEEK_SET
2 = SEEK_CUR
3 = SEEK_END

This function changes to the current position in the file. The new position can be specified relative to the beginning of the file, the current position, or the end of the file. The **whence** argument determines which.

A positive **offset** moves forward in a file, and a negative **offset** moves back toward the file beginning.

This function may indirectly cause reads and/or writes to the file to occur.

It is an error to seek to before the beginning of the file.

It is not an error to seek to beyond the end of the file. If this happens and subsequent writes occur at this new position, the skipped over bytes will be written with zeros.

The EOF indicator will be cleared. Any bytes pushed back with **ungetc** will be discarded.

ftell

```
function ftell  (fileID: ptr to FILE) returns int
                     [ Max_Stack_Usage = 48 ]
```

This function returns the current position in the file, i.e., 0 ... fileSize.

Upon an error, **errno** will be set and -1 returned. Otherwise **errno** will be set to zero.

When the file is positioned before the first byte, it returns 0.

When the file is positioned at EOF (after the last byte), it returns the file size in bytes.

fputs

```
function fputs  (src: String,
                 fileID: ptr to FILE)
                     returns bool
                         [ Max_Stack_Usage = 240 ]
```

This function writes the bytes in **src** to the file/stream. As is the convention in Blitz, all bytes in the string up to the current size are written.[40]

[40] This differs from Unix, in which everything up to, but never including, the \0 byte is written.

If all is ok this function returns true. If an error occurs, it returns false and **errno** indicates the nature of the error.

fread

```
function fread  (buffPtr: ptr to void,
                 size: int,
                 count: int,
                 fileID: ptr to FILE)
                     returns int
[ Max_Stack_Usage = 8 ]
```

This function returns the count of items successfully read.

NOTE: Not yet implemented.

fwrite

```
function fwrite (buffPtr: ptr to void,
                 size: int,
                 count: int,
                 fileID: ptr to FILE)
                     returns int
[ Max_Stack_Usage = 8 ]
```

This function returns the count of items successfully written.

NOTE: Not yet implemented.

fgets

```
function fgets  (dest: String,
                 fileID: ptr to FILE)
                     returns bool
```

```
[ Max_Stack_Usage = 8 ]
```

This function reads bytes from the file/stream up to EOF, a '\n' character, or the maximum array size.

If all is ok this function returns true. If an error occurs, it returns false and **errno** indicates the nature of the error.

NOTE: Not yet implemented.

Implementation

A Blitz program runs as machine code on a Blitz hardware core, which is intended to be a bare machine running nothing else. When emulated, the Blitz emulator must faithfully behave exactly the way a Blitz hardware core is suppose to act.

Of course the emulator runs on a Unix/Linux/POSIX host operating system. The question is how can the Blitz program gain access to the underlying host OS if all it can do it execute machine instructions.

The answer is that there is a sort of "backdoor". The emulator is tasked with implementing a number of devices, such as the "SimpleSerial" device. To provide the backdoor to the host, another I/O device has been added, called the "HostDevice".

From the point of view of running Blitz machine code, the host OS is a device. The Blitz code can communicate with it by controlling this "device" in order to send and receive information.

Within the emulator, which is charged with emulating devices, the HostDevice is "emulated" by simply invoking system calls to the underlying host OS.

Currently, the "HostDevice" occupies a single page in the memory-mapped I/O space at address 4_0010_8000. The running Blitz code sends commands and data to the host by writing to predefined addresses on this page. The Blitz code gets status and information from the host by reading from addresses in this page.

The details can be found in the following functions within the Blitz emulator, i.e., within blitz.c:

```
void putHostDevice (int64_t addr, int64_t value) {…}
int64_t getHostDevice (int64_t addr) {…}
```

You can consult these functions for details, but essentially it works like this.

Several memory locations are used to transfer arguments to the host function. The Blitz code will store values into these addresses as the first step. One location in particular (0x4_0010_8020) is used to store a function code. This code tells which host function is needed. (1=**fopen**, 2=**fclose**, …)

To perform the call to the host function, the Blitz code will read from a particular location called "do_it" (0x4_0010_8050). The read operation will invoke the host function. Like most C programs, the emulator will be suspended for the duration of the system call and, when it completes, the emulator will once again gain control. The emulator will then determine a status code — basically errno — and this will be the result of the read to the "do_it" address.

From the viewpoint of the executing Blitz code, reading from the "do_it" location will both cause the operation to occur and return the status from the operation. Some file functions also return a result as well as setting **errno**. For example, **fgetc** returns a character of data. For this, there is a second location in the HostDevice page called RET_VAL (0x4_0010_8048) in which the additional data is stored byte the emulator.

The **fopen** and **remove** operations must pass a pathname to the host OS. The HostDevice page also contains a 1,024 byte chunk of space which is set aside for storing that pathname. The Blitz code will store the pathname in that region before reading from "do_it".[41]

The command line arguments to the Blitz application program are specified on the command line of the Blitz emulator. For example

```
% blitz -g MyProgram.exe -args "-a -b -c"
```

will pass the following string to the Blitz program:

```
-a -b -c
```

[41] It is done this way, rather than having the emulator read the pathname from Blitz memory, since there could be a page fault if the pathname happens to lie within an unmapped page. But according to the ISA, page faults do not occur when accessing memory-mapped addresses. This assumes that backdoor to the host might in the future be used in user-level application code.

The argument string “-a -b -c” is passed from the emulator to the running Blitz code through the HostDevice page. Two addresses used. The first address is 0x4_0010_8000 (ARGS_SIZE) and can be queried by the Blitz code to determine the number of bytes in the string. The second address is 0x4_0010_8008 (ARGS_NEXT) and can be queried to retrieve a single byte of the string. The bytes are returned in order, terminating with a final null \0 byte. The Blitz code will allocate a new String and read from this address repeatedly, placing the bytes in this new String.

The transfer of the date-and-time string is done in a similar way, with the Blitz code reading from address 0x4_0010_8010 (DATE_SIZE) to determine the number of bytes in the string and reading from address 0x4_0010_8018 (DATE_NEXT) repeatedly to fetch the bytes. A new date and time are obtained from the host OS each time the DATE_SIZE address is queried, and the current position (i.e., the next byte to be retrieved from DATE_NEXT) is reset to the beginning of the string whenever DATE_SIZE is queried.

Chapter 6: Number Package

Introduction

The **Number** package contains several functions useful when working with values of type **double**.

For information about KPL's implementation of floating point, consult the chapter in this document titled "Floating Point".

If the programmer uses the **%e**, **%f**, or **%g** *FormatCodes* in a **printf** or **sprintf** statement, then this package must be included. The code that performs the conversion into a string of decimal characters is also available directly to the programer as functions like **doubleToString**.

The **doubleToString** and **doubleToStringWithOptions** functions convert a **double** value into a decimal representation, analogously to "strfromd" in Unix/Linux/POSIX.

The **stringToDouble** and **stringToDoubleWithOptions** functions work in the opposite direction. They parse a string looking for a floating point number, analogously to "strtod" or "atod" in Unix/Linux/POSIX.

The functions **isfinite**, **isnormal**, and **isdenormal**, which are used to classify a double value, are also included in this package.

There are a number of useful constants (such as **FP_PI** and **FP_MAX_NORMAL**) defined here.

This package also include a "big number" class. In Blitz and other 64-bit computers **int** values are limited to the range

-9,223,372,036,854,775,808 ... +9,223,372,036,854,775,807

While **ints** are adequate for all but most unusual applications, there is occasionally a need to work with values that exceed this range. The "big number" classes and methods provide this capability.

doubleToString

```
function doubleToString (d: double) returns String
```

Given a **double**, allocate and return a string representing the value. Here are examples of what might be returned:

```
0.0
-0.0
inf
-inf
nan
456.125
-5.0
1.23456e+123
```

Short for "**doubleToStringWithOptions** (d, true, false, 15, null)".

NOTE: This function always allocates a new string on the heap. It frees any other memory it allocates.

doubleToStringWithOptions

```
function doubleToStringWithOptions (d: double,
                                    wantNice: bool,
                                    precision_: int,
                                    isInexactAddr: ptr to bool
                                        ) returns String
```

Given a **double**, this function will allocate and return a string representing the value. Here is an example of a string that might be returned:

```
1.23456e+123
```

If **wantNice** is true, then some numbers will be printed in a more human-friendly way, like this:

```
123.456
100000.0
0.00001
```

Here is a typical usage:

```
str = doubleToStringWithOptions (d, true, false, 17)
```

The following usage is recommended to fix "4.999...999" to “5.0":

```
str = doubleToStringWithOptions (d, true, false, 15)
```

The returned value for special cases is determined by **PrintPreferences**. The defaults are:

`<pos infinity>`	*as determined by **floatPosInf***
`<neg infinity>`	*as determined by **floatNegInf***
`<not-a-number>`	*as determined by **floatNaN***
`-0.0`	
`0.0` *or* `+0.0`	*depending on **floatPositiveSign***

For the basic case (**wantNice**=false) the result will always be in exponential form. There will always be exactly one digit to the left of the decimal point and there will always be at least one digit to the right of the decimal point. Trailing zeros will be removed. Here are some examples:

```
-1.23125e+123
1.23456e+2
4.5e-10
1.0e+0
```

The number is printed with exactly **precision** digits of accuracy, although trailing (insignificant) zeros are removed. In other words, the exact value will be rounded to **precision** digits and then any trailing zeros to the right of the decimal place will be removed. However, if the number is an integral value, it will be printed with as, e.g., “`123.0`” and not “`123.`”.

A precision of ≤ 0 will default to 16 if **wantNice** is true, and 17 otherwise. A precision of 16 is nice, since it turns “3.9999999999999999” into “4.0”. A precision

of 17 is good since every double value will print differently. A precision of 1074 will give the exact decimal value of any double; values above 1074 will have no effect, since trailing zeros are removed.

NOTE: Some strings with 17 digits of precision will map to the same value. For example, 1.2345678901234567 and 1.2345678901234568 both map to `0x3ff3c0ca428c59fb`, which prints as the former.

The exponent will have all leading zeros removed, which C doesn't do.

If **wantNice** is true, we will print some numbers in a friendlier form. Some numbers will be printed without any exponent. There will always be at least one digit to the right of decimal point. Integers below 100,000,000,000,000,000.0 will print without an exponent, regardless of precision. Otherwise, values will be printed in exponential form.

Here are some examples:

```
-0.00000000045
123.456
45.0
123000000000.0
1.23125e+123
```

If any significant zeros were removed, then **isInexact** will be set to true, otherwise to false. By this we mean that true will be stored at **isInexactAddr**. If **isInexactAddr** is null, it is ignored.

NOTE: This function always allocates a new string on the heap. It frees any other memory it allocates.

printDoubleLikeC

```
function printDoubleLikeC (d: double) returns String
```

Given a double, this function allocates a string of digits representing the value. The intent is to produce exactly the same characters that "%.17g" in C/C++ would produce.

This functionality is useful in the KPL compiler. There are two versions of the compiler, one coded in C++ and the other in KPL. The goal is that both versions should produce exactly the same output. To test the compiler, there is a large test suite with many programs. In order to run the test suite and compare the results, we need to have the output match exactly, in order to avoid lots of distracting output.

There are a couple of minor discrepancies — concerning whether the output is in 999.999 form or in 9.999e99 form — but the match is exact in almost all cases.

NOTE: This function always allocates a new string on the heap. It frees any other memory it allocates.

doubleToAllDigits

```
function doubleToAllDigits (d: double,
                            timesTenToThePowerAddr: ptr to int)
                                returns String
```

Given a **double**, this function allocates a string of digits containing a decimal integer. It also sets **timesTenToThePower** to the position of the decimal point. Taken together, this will be an exact decimal representation of the value.

For example, d = 470000.0 would yield "47" with **timesTenToThePower** = 4. For example, d = -793.125 would yield "-793125" with **timesTenToThePower** = -3. If the number is negative, the string will begin with a "-" character. But "+" will never be added. For +0.0, -0.0, posInf, negInf, or NaN, it will return null, with **timesTenToThePower** = 1, 2, 3, 4, or 5 respectively.

stringToDouble

```
function stringToDouble (str: String) returns double
```

This function will scan a number and return the value. It will throw the following if there is a parse error:

ERROR_stringToDoubleError

The following examples show what can be parsed:

"123.456"
" +123.456e+10 "
" +inf "
"-inf"
"nan"
"-0.0"

A decimal point or "e" is required. Underscores are allowed.

NOTE: This function will set flags UF-UNDERFLOW, OF-OVERFLOW, NX-INEXACT if these conditions arise.

stringToDoubleWithOptions

```
function stringToDoubleWithOptions (str: String,
                                    startingPos: int,
                                    scanTrailingWhitespace: bool,
                                    simpleIntegerIsOk: bool,
                                    resultAddr: ptr to double,
                                    nextPosAddr: ptr to int
                                        ) returns bool
```

This function will scan the string **str** which is expected to contain a floating point number and it will store the value as a **double** at **resultAddr**.

It will return true if there were no problems and false if the input string was in error.

Here are some examples of what it can handle:

" 123.456 "
" 123 " — *depends on **simpleIntegerIsOk***
" 123.456 xx " — *depends on **scanTrailingWhitespace***
"+123e+10"
"123E7"
"12_345_678.0000001"
"12345678.000_000_1"

“-123.456e-10”
“+0.0”
“-0.0”
“inf”
“-inf”
“nan”
“ \t\t 123.456 \t\t ”

For numbers, there are two options:

If **simpleIntegerIsOk** is false, then either a decimal point or “e” is required.

If **simpleIntegerIsOk** is true, then inputs like “123” are acceptable.

A leading “+” or “-” sign is optional and the exponent character can be either “e” or “E”. A sign on the exponent is optional. Infinity can written as either “inf” or “infinity” and a leading “+” or “-” sign can be provided. For “inf”, “infinity” “nan”, case is ignored.

The scan for a number begins at **str[startingPos]**. Leading whitespace (‘ ’ and ‘\t’) is ignored.

If **scanTrailingWhitespace** is true, the string will be checked to ensure that only whitespace follows the number. If **scanTrailingWhitespace** is false, the scanning will stop on the first character past the number. In either case, **nextPos** will be set to the index of where the scanning stopped.

A number may optionally contain underscores. If present, they must be spaced correctly.[42] For example:

“12_345_678.000_000_1”

Regardless of the number of digits provided, this algorithm will round correctly, using “round-to nearest, with ties to even”.

---

[42] More precisely, if any underscores are present to the left of the decimal point, there must be exactly one “_” separating each group of 3 digits to the left of the decimal point; the number must not begin with “_” and there must not be an “_” between the ones digit and the decimal point. If any underscores are present to the right of the decimal point, there must be exactly one “_” separating each group of 3 digits; the number must not end with “_” and there must not be an “_” directly after the decimal point.

If the number has a small number of digits (roughly 16 or fewer) and an exponent within -22 ... +22, then the conversion will use an efficient "fast-path" algorithm. For other numbers, an algorithm involving **BigInts** will be used[43].

NOTE: This function always allocates a new string on the heap. It frees any other memory it allocates.

NOTE: This function will set flags UF-UNDERFLOW, OF-OVERFLOW, NX-INEXACT if these conditions arise. A parser error or "nan" will not set NV-INVALID.

parseFloat

```
   function parseFloat (str: String,
                        startingPos: int,
                        scanTrailingWhitespace: bool,
                        simpleIntegerIsOk: bool,
                        nextPosAddr: ptr to int,
                        resultAddr: ptr to int,
                        mantissaAsIntAddr: ptr to int,
                        mantissaAsBigIntAddr: ptr to ptr to Number,
                        decimalExpAddr: ptr to int)
-- Want to scan trailing whitespace?
-- Is a number without decimal or exponent acceptable?
-- RESULTS: next position
--       code (-1,1,2,3,4,5: error,+inf,-inf,nan,+float,-float
--       mantissa (if mantissaAsBigNum is null
--       mantissa if the value requires more than 18 digits.
--       exponent, adjusted so mantissa interpreted as xxxx.0
```

This function parses a floating point number and returns the digits (as the **mantissa**) and the **exponent**.

For example, the input "123.400e10" will be returned as:

mantissa = 123400

[43] This is a complex, time-consuming algorithm. Everything allocated on the heap will be freed. The algorithm used here is described at:
https://www.exploringbinary.com/correct-decimal-to-floating-point-using-big-integers/

exponent = 7

Note that 123.400e10 = 1234000000000.0 = 123400.0e7

This function will also accept

"inf"
"+inf"
"-inf"
"infinity"
"+infinity"
"-infinity"
"nan"

These are case-insensitive, is things like "+InFiniTY" are accepted.

The **result** will always be set to indicate what was seen:

-1	Parse Error
1	+inf
2	-inf
3	nan
4	+number
5	-number

The mantissa value will be returned as a 64-bit **int**, unless it is very close to **MAX_64**. Otherwise, it will be returned as a **BigInt**. If the mantissa is returned as an **int**, **mantissaAsBigInt** will be set to null.

The exponent is returned as an **int**. If the user should ever provide a string with an extremal exponent (such as "1.0e+12345678901234567890123456789O"), it will be treated as an error.

The "**...Addr**" parameters must not be null.

The input can be required to contain either a decimal point or an exponent. However, if **simpleIntegerIsOk** is true, then the input need not contain either. If a decimal point appears, then there must be at least one digit before it and at least one digit after it.

This function will return the stopping position as **nextPos**. With **scanTrailingWhitespace**, the user can elect to have the function either ensure that only whitespace follows the number, or to simply stop after a complete number has been parsed. If we do not scan trailing whitespace, then **nextPos** will be the index of the character following the number (or the string length, if there are no more characters).

If we ignore trailing whitespace, things like "info**"** will be accepted as "inf", stopping at the "o". This means that "infiniti…" (which is misspelled) will be accepted as "inf" with "initi…" remaining to be scanned. Likewise "nanXXX" will be accepted as legal input if we ignore trailing whitespace. The caller may wish to check to make sure the next character is not a letter/digit/underscore, and flag such inputs as errors.

If there is a parse error, **nextPos** will be updated. It will be set to some value within 0..**stringSize** (i.e., just just beyond the last character). It will point to some spot that is approximately where the problem occurred.

MEMORY USAGE: This function may allocate space on the heap. It will free everything allocated, except the **BigInt** it returns in **mantissaAsBigInt**. If there is a parse error, it will set **mantissaAsBigInt** to null and free everything it allocated.

parseInteger

```
function parseInteger (str: String,
                       startIndex: int,
                       newStartAddr: ptr to int)
                            returns ptr to Integer
```

This function creates and returns either a **SmallInteger** if possible or a **BigInt** object, if the value is not within the range of **SmallInteger**.

The string is scanned from **str[startIndex]** until the end of the string. The parsing will stop with the first character after the integer. If successful, the stopping location will be stored at **newStartAddr**. If unsuccessful, **newStartAddr** will not be modified.

If no valid integer was found, then this function returns null.

Upon success, this function will scan the following:

```
(' ' | '\t')*   ( + | - | ε )    DecimalDigit+
```

NOTE: This function accepts 0, 00, 000..., +0, +00, +000... but not -0, -00, -000…

NOTE: Upon success, this function will allocate a new object on the heap. It frees any other memory it allocates.

fpclassify

```
function fpclassify (d: double) returns int
```

This function returns one of the following values:

```
1     FP_NAN
2     FP_INFINITE
3     FP_ZERO
4     FP_NORMAL
5     FP_SUBNORMAL          — Not including +0.0 or -0.0
```

NOTE: Zero values are usually considered subnormal numbers, but this function doesn't classify them as such. Instead, it follows the Unix/Linux/POSIX pattern.

isfinite

```
function isfinite (d: double) returns bool
```

This function returns true for zero, normal, and subnormal values; and false for **nan**, **+inf**, **-inf**.

isnormal

```
function isnormal (d: double) returns bool
```

This function returns true for any “normal” floating point number and false for any **nan**, **+inf**, **-inf**, zero, and subnormal value.

isdenormal

```
function isdenormal (d: double) returns bool
```

This function returns true for a “denormal” (i.e., “subnormal”) floating point number and false for any **nan**, **+inf**, **-inf**, +0.0, -0.0, and normal value.

NOTE: Zero values are usually considered subnormal numbers, but this function doesn’t classify them as such. Instead, it follows the Unix/Linux/POSIX pattern followed in **fpclassify**.

Chapter 7: Floating Point

Introduction

This chapter assumes you are familiar with floating point numbers. There are many introductions to floating point arithmetic, including:

> *"An Overview of Floating Point Numbers"*, by Harry H. Porter III

Single and Double Precision

Blitz implements:

> Double Precision Floating Point

Blitz does not implement single precision floating point.

Perhaps some implementations will add support for single precision, but this would be an extension of the Blitz-64 ISA (Instruction Set Architecture) specification.[44]

The floating point instructions are all Format-A instructions and there is plenty of additional room in the op-code space for more instructions and there are no major barriers to adding it.

Single precision was not included because it adds (1) complexity to the ISA, (2) complexity to the KPL language, (3) effort to any implementation, and (4) size and power consumption to any circuit. Any single precision calculation can be done with

[44] This statement applies to the version documented here. Refer to the section titled "About This Document" for details.

greater accuracy using double precision, so there is no arithmetic advantage to using single precision.

Of course single precision require half the memory and can thus be moved around faster. And some single precision operations can be performed faster than double precision.

Performance optimization is best addressed with a very different architectural design. Double precision is included for doing the relatively small, quick calculations that are occasionally required for programs, such as computing performance or "percentage utilization of some resource". For intense numerical computation, such as neural simulations or image rendering, special purpose engines are clearly the better choice.

IEEE 754

The IEEE 754 standard defines how computers shall implement floating point numbers. It is long and complex, but modern computers follow it.

Blitz-64 follows the IEEE 754 floating point standard.

The IEEE standard has some optional aspects and variations. The IEEE standard states that if double precision is implemented, the single precision must also be implemented. So right away, it is clear that Blitz is not technically in conformance.

Nonetheless, the intent of Blitz is to follow IEEE 754 correctly and accurately for those aspects that are implemented.

Floating Point Instructions

To access floating point, you write code like and the KPL compiler will translate it into the appropriate floating point machine instructions. For example:

```
var
  d: double
...
```

```
d = d * 123.456e-10
...
printf ("d = %g\n", d)
```

Here are the machine instructions for floating point:

FADD	RegD,Reg1,Reg2	RegD ← Reg1 + Reg2
FSUB	RegD,Reg1,Reg2	RegD ← Reg1 - Reg2
FMUL	RegD,Reg1,Reg2	RegD ← Reg1 × Reg2
FDIV	RegD,Reg1,Reg2	RegD ← Reg1 / Reg2
FMIN	RegD,Reg1,Reg2	RegD ← MIN (Reg1, Reg2)
FMAX	RegD,Reg1,Reg2	RegD ← MAX (Reg1, Reg2)
FNEG	RegD,Reg1	RegD ← -Reg1
FABS	RegD,Reg1	RegD ← ABSOLUTE_VALUE (Reg1)
FSQRT	RegD,Reg1	RegD ← SQUARE_ROOT (Reg1)
FEQ	RegD,Reg1,Reg2	RegD ← (Reg1 = Reg2) ? 1 : 0 (float compare)
FLT	RegD,Reg1,Reg2	RegD ← (Reg1 < Reg2) ? 1 : 0 (float compare)
FLE	RegD,Reg1,Reg2	RegD ← (Reg1 ≤ Reg2) ? 1 : 0 (float compare)
FGT	RegD,Reg1,Reg2	RegD ← (Reg1 > Reg2) ? 1 : 0 (float compare)
FGE	RegD,Reg1,Reg2	RegD ← (Reg1 ≥ Reg2) ? 1 : 0 (float compare)
FCVTFI	RegD,Reg1	Convert: floating-point ← int
FCVTIF	RegD,Reg1	Convert: int ← floating-point
FMADD	RegD,Reg1,Reg2,Reg3	RegD ← (Reg1 × Reg2) + Reg3
FNMADD	RegD,Reg1,Reg2,Reg3	RegD ← (-(Reg1 × Reg2)) + Reg3
FMSUB	RegD,Reg1,Reg2,Reg3	RegD ← (Reg1 × Reg2) - Reg3
FNMSUB	RegD,Reg1,Reg2,Reg3	RegD ← (-(Reg1 × Reg2)) - Reg3
GETFSTAT	RegD	RegD ← CSR_STATUS [floating bits]
PUTFSTAT	Reg1	CSR_STATUS [floating bits] ← Reg1
FCLASS	RegD,Reg1	RegD ← classify(Reg1) || FLOAT_STATUS

The **FGT** and **FGE** instructions are synthetic and implemented in terms of **FLT** and **FLE**.

The **FCLASS** instruction is going away and will be replaced by **GETFSTAT** and **PUTFSTAT**.

The double precision values are stored in memory and registers. When in memory, they are always stored at doubleword-aligned addresses. The general purpose registers are used for floating point; there are no “floating point registers”. Floating values are moved with the same instructions that are used to move 64 bit signed integers, namely **LOADD**, **STORED**, **MOV**, ...

To compare floating point values, the programmer can write code like this:

```
if d1 == d2 ...
if d1 != d2 ...
if d1 <  d2 ...
if d1 <=  d2 ...
if d1 >  d2 ...
if d1 >= d2 ...
```

Equality comparison of two floating point values must the **FEQ** instruction, since there are special rules concerning the not-a-number (NaN) value. There is no “FNE” instruction; the compiler will generate an **FEQ** instruction instead. To compare the actual bits, the programmer can use this:

```
if copyBitsToInt (d1) == copyBitsToInt (d2) ...
```

Implementing floating point in silicon is expensive and it is envisioned that some implementations will choose to avoid this complexity. So these instructions may actually be emulated. If emulated, the running program will be none-the-wiser. It will produce the same results, although it will of course run slower, to the extent it uses any emulated instructions.

The **GETFSTAT** and **PUTFSTAT** are mandatory and must be included in any implementation. Any or all of the remaining instructions may be either implemented or cause an **Emulated Instruction Exception**.

It is envisioned — but not mandatory — that if any of the following instructions are emulated, they are all emulated, and if any are implemented in silicon, then they are all implemented in silicon:

FADD
FSUB
FMIN
FMAX
FNEG

FABS
FEQ
FLT
FLE
FGT
FGE

It is envisioned that, if the above instructions are emulated, then **FMUL** will also be emulated. However, **FMUL** is significantly more complex and even if the above instructions are implemented in silicon, **FMUL** may be emulated.

Likewise, it is envisioned that, if the **FMUL** is emulated, then **FDIV** will also be emulated, but a hardware implementation of **FMUL** does not imply a hardware implementation of **FDIV**. Division is more complex and used less often than multiplication, so it may make sense to emulate it.

Square root is even more complex and used less often than division, so it may make sense to emulate **FSQRT**, even when **FDIV** is implemented in hardware.

It is envisioned that the following instructions are either all emulated or all implemented in silicon:

FMADD
FNMADD
FMSUB
FNMSUB

Likewise, these instructions may group together:

FCVTFI
FCVTIF

The overhead to invoke a function is less than the time to invoke an emulation routine. Calling a function may require no more than a CALL and RET instruction. Because of the calling conventions, the function may avoid any saving of registers on the stack. But when we emulate an instruction, we have the overhead of saving registers, dispatching to the right code, dissecting the instruction and obtaining the arguments, restoring the registers, and returning to the interrupted code. Emulation of instructions should be avoided if at all possible and only included when necessary

for deficient implementations. Even an implementation in which the pipeline is inoperative and

If an operation will always be implemented in software, it is far better to avoid adding it to the instruction set. It is better to make it a first-class function and require all programs to call it.

The square root operation is really complex and almost certainly would be emulated. For this reason there is no "FSQRT" instruction.

We chose the prefix "F" for "floating" rather than "D" for "double" to prevent confusion with "doubleword" integers. If, in the future implementation, single precision instructions are added, we propose that the new instructions add "**.S**" for "single":

double precision	*single precision*
FADD	**FADD.S**
FSUB	**FSUB.S**
FMUL	**FMUL.S**
FDIV	**FDIV.S**
…	…

Double Precision Numbers

Here is the representation of a 64-bit double precision floating point value:

The interpretation of the "exponent" bit patterns is:

Bit Pattern	**Meaning of Exponent Field**
000 0000 0000	-1022 — *Denormalized Numbers, including zero*
000 0000 0001	-1022

```
   ...              ...
011 1111 1110    -1
011 1111 1111    0
100 0000 0000    +1
   ...              ...
111 1111 1110    +1023
111 1111 1111    Infinity, Not-a-Number
```

Infinity If the exponent is all ones (i.e., 111_1111_1111), then the value of the fraction matters. If the fraction is all zeros, then the value is +∞ or –∞ depending on the sign bit.

+∞:

```
0x7FF0000000000000 =
   0 11111111111 0000000000000000000000000000000000000000000000000000
```

-∞:

```
0xFFF0000000000000 =
   1 11111111111 0000000000000000000000000000000000000000000000000000
```

Not-A-Number If the exponent is all ones (i.e., 111_1111_1111) and the value of the fraction is not all zeros, then NaN is represented. There are multiple representations that are to be interpreted as NaN values. Here are two common representations for NaN:

NaN:

```
0xFFFFFFFFFFFFFFFF =
   1 11111111111 1111111111111111111111111111111111111111111111111111
0x7FF8000000000000 =
   0 11111111111 1000000000000000000000000000000000000000000000000000
```

Normalized Numbers If the exponent field is not all zeros and not all ones, then the value represented is:

$$(-1)^{\textbf{sign}} \times 1.\textbf{fraction} \times 2^{\textbf{exponent}}$$

where **fraction** represents the 52 bit pattern in bits [51:0] as a binary fixed point number.[45]

[45] For example, "110.1" is the binary fixed point representation for the number we write in decimal as 6.5 and "1000.01" is the number 8.25. As you can see, "1.fraction" ranges from 1.0000…0 to binary 1.1111…1, which (when expressed in decimal) is from 1.0 to something like 1.9999…9.

The **exponent** ranges from -1022 to +1023.

For double precision, the "**bias**" is defined as +1023.

The meaning of the **exponent** bits [62:52] can be determined by interpreting them as an integer within 1 … 2046 (i.e., within 000_0000_0001 … 111_1111_1110) and subtracting the **bias**, to give an exponent in the range -1022 … +1023.[46]

The most significant bit indicates the **sign**, with 1=negative.

Denormalized Numbers If the exponent field is all zeros (i.e., 000_0000_0000), then the value is a "denormalized" number[47]. The value of the number is:

$$(-1)^{\mathbf{sign}} \times 0.\mathbf{fraction} \times 2^{\mathbf{-1022}}$$

The leading implicit "1" bit is no longer assumed; it is now "0". Also the exponent is always -1022, which happens to be the smallest exponent for normalized numbers.

Examples: The **largest normalized number** is:

In binary: `1.11111111...11111111111` × 2^{+1023} (There are 1+52 ones)
Representation: `0x7FEF_FFFF_FFFF_FFFF`
Decimal approximation: 1.7976931348623157 × 10^{+308}

The **smallest positive normalized number** is:

In binary: `1.00000000...00000000000` × 2^{-1022} (There are 52 zeros)
Representation: `0x0010_0000_0000_0000`
Decimal approximation: 2.2250738585072014 × 10^{-308}

The **largest denormalized number** is:

In binary: `0.11111111...11111111111` × 2^{-1022} (There are 52 ones)
Representation: `0x000F_FFFF_FFFF_FFFF`
Decimal approximation: 2.2250738585072009 × 10^{-308}

[46] The following names are sometimes used: **emin** =-1022, **emax** = +1023

[47] The term "subnormal" is synonymous with "denormal" and "denormalized".

The **smallest positive denormalized number** is:

In binary: `0.000000000...00000000001` × 2^{-1022}
Representation: `0x0000_0000_0000_0001`
Decimal approximation: 4.9406564584124654 × 10^{-324}

Zero is represented as a denormalized number:

In binary: `0.000000000...00000000000` × 2^{-1022}
Representation: `0x0000_0000_0000_0000`
Exact value: +0.0

There are two zero values. The zeros are generally considered to be "denormalized numbers".

+0.0:

```
0x0000000000000000 =
   0 00000000000 0000000000000000000000000000000000000000000000000000
```

-0.0:

```
0x8000000000000000 =
   1 00000000000 0000000000000000000000000000000000000000000000000000
```

Decimal Representation

All floating point numbers can be represented exactly in decimal. Of course all integer values can be represented exactly, and the decimal representation is always more concise[48]:

Binary	Decimal Equivalent
11000000111001	12,345

While every binary floating point value can be represented exactly in decimal, some values require a lot of digits to represent exactly.

[48] To be precise, it is equally concise for 0 and 1.

For example, the smallest denormalized number (0.000000000...00000000001 × 2^{-1022}, which equals 2^{-1074}) has 1074 digits to the right of the decimal point, 751 of which are significant. It is exactly:

```
0.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000494065645841246544176568792868221372365059802614324764425585682500675507270
20875186529983636163599237979656469544571773092665671035593979639877479601078187
81263007131903114045278458171678489821036887186360569987307230500063874091535649
84387312473397273169615140031715385398074126238565591171026658556686768187039560
31062493194527159149245532930545654440112748012970999954193198940908041656332452
47571478690147267801593552386115501348035264934720193790268107107491703332226844
75333572083243193609238289345836806010601150616980975307834227731832924790498252
47307763759272478746560847782037344696995336470179726777175851256605511991315048
91101451037862738167250955837389733598993664809941164205702637090279242767544565
22908753868250641971826553344726562 5
```

One useful property of double precision floating point numbers is this:

> *When expressed in decimal and reduced to 17 significant digits, every double precision floating point value has a unique representation.*

Thus, it is often useful to display no more than 17 significant digits of a number. This allows you to distinguish two different numbers. And precision beyond 17 digits is not really useful, since nothing in the physical world can be measured accurately to that precision.

But note that not every 17 digit number can be represented uniquely.

> *Some decimal values that differ in the 17th significant digit will map into the same double precision floating point value.*

For example, the following decimal numbers:

```
0.40000000000000000
0.40000000000000001
0.40000000000000002
```

all map into:

```
0x3fd999999999999a
```

Rounding

When we go from decimal to floating point, we may have to round, since not all numbers can be represented exactly. When we go from floating point to decimal, we may have to round, since we often want a small, human-friendly number of digits.

Also, when an operation such as addition or multiplications is performed, we need to round, since the exact result often cannot be precisely represented as a double precision value.

The IEEE 754 standard defines several ways to round:

- Round toward zero (truncation)
- Round up, toward + infinity
- Round down, toward - infinity
- Round to nearest, with ties toward zero
- Round to nearest, with ties toward the even value

Blitz implements only "**round to nearest, with ties to even**".

"**Round-toward-zero**" is also called "**truncation**" and is especially easy to implement: just ignore any extra bits. Ignoring difficult data is rarely the correct way to do things. The situations in which "**round-up**" and "**round-down**" are mathematically more justifiable than "**round-to-nearest**" are unknown to me.

From a mathematical perspective, "**round-to-nearest**" is the most reasonable. Blitz avoids the complexity and implementation cost of supporting multiple rounding methods.

There is a question of how to round when the exact value is exactly half way between the two nearest values that can be represented.

For example, if we wish to round 3.5 to the nearest whole number, we have a "tie" since 3.5 is exactly half way, so there are two "nearest" values.

With decimal integers, an even number ends in 0, 2, 4, 6, or 8. With binary numbers, an even number ends with 0 and an odd number ends with 1. We can extend this

definition to double precision values. The bit in the least significant place determines whether the number is even or odd.

For example:

even:

$1.0110010110010010100110011110010100110101011010110110 \times 2^{-123}$

odd:

$1.0110010110010010100110011110010100110101011010110111 \times 2^{-123}$

"**Round-to-nearest-with-ties-to-even**" means to round exact ties to the value with a zero bit in the least significant place.

In terms of implementation cost, rounding ties "to even" as opposed to rounding ties "toward zero" is not significantly different. The biggest difficulty is in detecting an exact tie, but this is not significantly harder than detecting whether a value is a tiny bit above or a tiny bit below a tie. The mechanism for rounding a value up is required regardless of how exact ties are dealt with. Basing the direction on the least significant bit is not particularly difficult.

Things To Remember About Floating Point

We can make the following statements about floating point numbers:

- Every floating point numbers has a sign. Every number is either positive or negative.
- There are two representations for zero: positive zero (i.e., +0.0) and negative zero (i.e., -0.0).
- There are two representations of infinity: positive infinity (+∞ or +inf) and negative infinity (-∞ or -inf)
- The exponent may be positive or negative, allowing both very large numbers and very small numbers.
- There is a special representation called "not a number" ("NaN"). This value can represent a missing value or the result of an undefined operation, such as

divide-by-zero. In some implementations there are two variations, called “quiet NaN” and “signaling NaN”.

- Every 32-bit integer (i.e., every integer in the range -2,147,483,648 to +2,147,483,647) can be represented exactly with a 64 bit double precision floating point number, but not with a single precision float. In fact, the integer range is a little greater: every 54-bit integer can be represented exactly in double precision floating point. Almost all larger integers will get rounded.

Floating point numbers have these limitations:

The range of the exponents is limited.

The amount of precision is limited.

The limited range of exponents means that we cannot represent very large numbers or numbers very close to zero. However, 10^{+308} is a truly enormous number and 10^{-308} is really small, infinitesimally close to zero. It is difficult to imagine any physical quantity that would be measured with numbers outside of this range.

Likewise, there is a limit on precision, although it is unlikely that any value can be measured with this accuracy or that such accuracy will be inadequate.

The danger is this:

Complex calculations, particularly involving numerous arithmetic operations, will introduce errors in accuracy.

These errors may tend to combine and accumulate, causing the final results to be grossly inaccurate. This is a particular risk when a computed value is the product of many operations, as in an iterative algorithm.

With scientific notation, humans often express the amount precision or accuracy of a value, showing the accuracy by the number of digits. Measurements that are more accurate have more digits, and less accurate values are rounded to values with fewer digits.

For example, these two values are exactly equal, but they suggest different confidences in their accuracy:

7.25×10^6
7.2500×10^6

With floating point, these two numbers are represented identically. After all, they are really the same number.

Each floating point value is nothing more than a value. There is no information about the accuracy of that value.

Since there is a finite number of bits available for each number, there are only a finite number of values that can be represented. Consequently:

Many values cannot be represented.

Instead, we must make-do with numbers that are nearby and approximately equal to the desired value. The value that the bits of a floating point value represent will be the closest approximation to the true, correct value. At least we hope so!

Floating point representation is fundamentally a binary representation, not a decimal representation. As a consequence:

Many simple decimal values cannot be represented.

For example, the following commonly used number can be represented simply and exactly in decimal, but cannot be represented exactly in floating point:

0.3

The closest we can come with double precision is:

0.2999999999999999888977697537484345957636831909179687 5

Because of the limitations of floating point, almost every operation (such as +, -, ×, and ÷) will introduce errors. And the more operations that are performed, the greater the inaccuracy of the final result.

Arithmetic operations are almost always inexact and introduce errors.

Errors tend to get larger as more operations are performed.

Some rational numbers require an infinite number of digits in their decimal representation. For example:

1/3 = 0.33333…

Likewise, some rational numbers may require an infinite number of bits in their binary representation.

Regardless of whether we represent a rational number in decimal or binary, the infinite strings of digits/bits will settle into a simple repeating pattern. This is true of all rational numbers, but irrational numbers (e.g., π, $\sqrt{2}$) do not have such simple decimal or binary representations. Neither their decimal nor their binary expansions will ever exhibit a repeating pattern.

Some numbers many have a finite representation in decimal but require an infinite sequence in binary. For example, the following number:

4.3

requires an infinite binary expansion to represent it, namely:

100.01001100110011… = 100.01(0011)*

It turns out that every binary number without a repeating part can be represented with a finite number of decimal digits. Furthermore, the number of digits to the right of the decimal point will never exceed the number of places to the right of the binary point. For example:

101.1101 *(binary)* = 5.8125 *(decimal)*

Turning to floating point representation, we have limited number of bits available, which means we cannot accommodate arbitrary precision.[49] Not every number is representable, so we must round numbers to a nearby number that is representable.

For example, the number 6.022×10^{23} can only be represented approximately, even though it appears not to have a great amount of precision. The closest number that can be represented using a double precision floating point is slightly larger:

[49] Recall there is a countable infinity of rational numbers between any two numbers, yet with only 64 bits, we only have a small number of unique representations.

$$6.022000000000000 27262976 \times 10^{23}$$

The next-closest number is a little less:

$$6.021999999999999 60154112 \times 10^{23}$$

No number between these two values can be represented exactly.

The underlining shows 17 digits, which is why we say that doubles can represent values with "about" 17 digits of accuracy.

Programs with floating point involve inaccuracy. To predict the expected accuracy of a computation is not at all trivial, yet should not be ignored.

Floating point arithmetic is meant to mimic mathematical arithmetic, but it must be remembered that they are only approximately the same:

- The exact value or result of an operation is **not** always representable, so the computed answer is often **not** mathematically correct.
- Floating point addition is **not** always associative, due to rounding errors. That is, (x + y) + z is not always equal to x + (y + z).
- Floating point multiplication is **not** always associative. That is, (x * y) * z is not always equal to x * (y * z).
- Floating point multiplication does **not** always distribute over addition with the exact same results. That is, x * (y + z) is not always equal to (x * y) + (x * z).

However, we can say this:

- Floating point addition and multiplication are commutative, like math. For example, x+y = y+x, so you don't have to worry about the order of operands for a single operation.

Compile-Time v. Runtime Computation

The KPL compiler will simplify expressions when it can, in order to avoid performing the computation at runtime. In this code:

```
const
  N = 123.456
...
d = d + (10.0 * N)
```

the compiler will perform the multiplication at compile-time. The compiler will perform the operations in the same order and in the same way, so that the result will be exactly the same as if it had been performed at runtime.[50]

The compiler will never re-order arithmetic computations, either floating point or integer, if this could possibly result in a different numerical result or in different overflow/error/rounding behavior.

> ***The programmer has complete control over the order of arithmetic computations.***

However, this means that certain optimizations that might be desired must be made explicitly by the programmer.

The programmer can control the order of operations with parentheses. For example, these two statements will yield different code. They execute the additions in a different order:

```
a = (b + c) + d
a = b + (c + d)
```

Exceptions and Error Terminology

IEEE 754 defines the following five error conditions:

[50] That is, assuming the program compiles. The compiler will flag some operations as errors. For example, "0.0 / 0.0" will be flagged since this operation is invalid and yields not-a-number. If the programmer truly wants a "not-a-number", then **nan** must be used explicitly.

Invalid
Overflow
Underflow
Divide-by-zero
Inexact

In IEEE 754, these are called "*exceptions*", but we avoid this terminology. In Blitz, the term "*exception*" is used differently to mean hardware exception processing. Here are a few of the Blitz exceptions:

Illegal Instruction Exception
Unaligned LOAD / STORE Exception
Syscall
… etc …

None of the floating point error conditions will cause exception processing in Blitz. Instead, the occurrence of an error will set a status bit, which can be checked by software later.

In Blitz, there is an important status register called **CSR_STATUS**, which controls the processor execution. Among other things, the register contains these fields:

FLOAT_ROUND (2 bits)
FLOAT_STATUS (5 bits)

The **FLOATING_ROUND** bits are meant to determine which rounding technique will be used.

The 5 bits of the **FLOATING_STATUS** fields are given these names:

NV - Invalid
OF - Overflow
UF - Underflow
DZ - Divide-by-zero
NX - Inexact

These bits correspond one-to-one to the error conditions required by IEEE 754. When the "exception" (as IEEE calls it) arises, the corresponding bit is set to 1. The bits are sticky and remain set. These bits are not cleared by the floating point

instructions; instead they are cleared by an explicit operation. Typically, these 5 bits are cleared when a program begins execution. Subsequently, they can be be examined, if desired, to determine whether an error has arisen during computation.

In the design of the Blitz ISA, creating an exception for floating point errors was considered, but rejected.[51]

Here is the meaning of these error conditions:

NV-Invalid operation
The result is mathematically undefined, e.g., the square root of a negative number. The result is **NaN**.

DZ-Division by zero
An operation on finite operands gives an exact infinite result, e.g., 1.0/0.0 or log(0.0). The result is **±inf**.

OF-Overflow
A result is too large to be represented correctly (i.e., its exponent with an unbounded exponent range would be larger than **emax**). The result is **±inf**.

UF-Underflow
A result is very small (outside the normal range) and is inexact. The result is a subnormal or zero.

---

[51] The proposal was to have a single exception type called "FLOATING_INVALID_EXCEPTION". Recall that the NV-Invalid bit in FLOAT_STATUS is set to 1 whenever an operation that does not have NaN as arguments produces a NaN result. The proposal would add a single bit to CSR_STATUS called "FLOATING_EXCEPTION_ENABLED", which would be set and cleared by software. Whenever a NaN is produced, i.e., whenever an operation is found to be invalid and the NV bit is set, and when FLOATING_EXCEPTION_ENABLED is true, a FLOATING_INVALID_EXCEPTION would occur. The flow of control would be interrupted and a trap handler would be invoked, in the same way as for the other Blitz exceptions.

Such an exception would mirror the ARITHMETIC_EXCEPTION which applies to integer arithmetic. In general, this fits with the Blitz philosophy to check all operations for errors and immediately stop and report an error when it occurs. One problem is that floating instructions can be expected to require many cycles. In any pipelined architecture, it seems there is a choice between allowing imprecise interrupts or imposing a large implementation overhead, especially if we wish to keep the pipeline full with unrelated instructions that happen to follow a floating instruction. Neither choice is palatable. But most importantly, we already have a mechanism — the NaN itself — for detecting errors, whereas with integer arithmetic there is no other way to detect whether an error has occurred.

While this decision is subjective, we note that nothing precludes adding this mechanism in future versions of the ISA.

NX-Inexact
The exact, unrounded result is not representable exactly. The result is the rounded value.

Invalid Operation

If there is a problem with some floating point operation, the result will be not-a-number (**NaN**) and the **NV-Invalid** bit in **FLOATING_STATUS** will be set.

There are three general reasons for such an error:

- One of the operands was non-a-number to start with.
- The result is mathematically undefined. For example, 0/0.
- The result is a complex number. For example, "square root of a negative".

When the Result is Undefined

The following operations will result in **NaN** and the **NV-Invalid** bit will be set. Each of these is associated with either a FADD, FSUB, FMUL, or FDIV machine instruction.

- $+\infty + -\infty$
- $-\infty + +\infty$
- $+\infty - +\infty$
- $-\infty - -\infty$
- $\pm 0 \times \pm\infty$
- $\pm\infty \times \pm 0$
- $\pm 0 \div \pm 0$
- $\pm\infty \div \pm\infty$

The IEEE standard discusses other floating point operations which may generate **NaN** such as:

- The **powr** function defines 0^0, 1^∞, and ∞^0 as **NaN**.[52]

In Blitz, these functions will be implemented in software in the future and will be expected to follow the IEEE specifications and to set the **NV-Invalid** bit**.**

When the Result is a Complex Number

Here are some operations which result in a complex number:

- The square root of a negative number
- The logarithm of a negative number
- The inverse sine or cosine of a number that is less than −1.0 or greater than +1.0

The IEEE spec says that these operations should yield a result of **NaN** and the "invalid operation" flag should be set.

In Blitz, square root is implemented by the **FSQRT** machine instruction. If the argument is negative, the result will be **NaN** and the **NV-Invalid** bit will be set.

The other functions may be implemented in software in the future. Of course, they will be expected to follow the IEEE specification and set the **NV-Invalid** bit when the result is a complex number.

Conversion Between Integers and Floating Point

Floating Point → Integer

The following Blitz-64 machine instruction converts from double precision to a 64-bit integer:

```
fcvtif   r4,r5     # Floating ConVerT to Integer from Floating
```

[52] Traditionally, the **pow** function and the integer exponent **pown** function define 0^0, 1^∞, and ∞^0 as 1.

This instruction will be executed by the KPL built-in function **forceToInt**, which is used to convert a **double** value to an **int**.[53]

What if the argument is **NaN**? The IEEE principal that "**NaN**s must always be propagated" cannot be followed since the result — a 64-bit signed, twos complement integer — cannot represent a **NaN** value.

Instead, the **fcvtif** instruction will set the **NV-Invalid** flag in the **CSR_STATUS** register. The integer result will be "0".

What happens when the floating point number exceeds the range of the integers? The integer result will either be largest positive integer or the most negative integer and the **OV-Overflow** flag will be set.

Result if too large...
MAX_64 = +9,223,372,036,854,775,807 = 0x7FFF,FFFF,FFFF,FFFF
Result if too negative...
MIN_64 = -9,223,372,036,854,775,808 = 0x8000,0000,0000,0000

Here are the values around the largest signed integer (0x7FFF,FFFF,FFFF,FFFF = 9,223,372,036,854,775,807) that can be represented exactly with double precision floats:

+9,223,372,036,854,774,784.0
+9,223,372,036,854,775,808.0
+9,223,372,036,854,777,856.0

Note that the integer 9,223,372,036,854,775,808 is representable exactly as a double precision float since it is 1.0×2^{63}. This integer can be expressed as the unsigned number 0x8000,0000,0000,0000, which is one greater than we can represent as a signed 64 bit integer. (Of course, we can represented its negation -9,223,372,036,854,775,808 exactly as a signed integer as 0x8000,0000,0000,0000.)

Using double precision floats, the adjacent values to this large number differ by a substantial amount, due to the loss of precision at this magnitude. The integer representation has 63 zero bits while the double precision representation has only 52 zero bits. We lost 11 bits. Note that 2^{11} = 2,048 and binary 1000_0000_0000 is

[53] This function is named "force..." rather than something like "doubleToInt", since there is not always an exact conversion and accuracy may be lost. There is the possibility that the **OV-Overflow**, **NV-Invalid**, and **NX-Inexact** bits will be set. Hopefully, the name will remind the programmer of these error possibilities.

decimal 2,048. Thus, going to the next greatest number, will add 2,048 to the value. Going down instead, we must decrement the exponent and go to:

$$1.1111111111111111111111111111111111111111111111111111 \times 2^{62}$$

which is 1,024 less.

For Blitz-64, the test for overflow is this:

If the floating value is greater than +9,223,372,036,854,775,807.0 (including **+inf**) then the **OV-Overflow** and **NX-Inexact** bits will be set and the result will be +9,223,372,036,854,775,807 (i.e., 0x7FFF,FFFF,FFFF,FFFF).

If the floating value is more negative than -9,223,372,036,854,775,808.0 (including **-inf**) then the **OV-Overflow** and **NX-Inexact** bits will be set and -9,223,372,036,854,775,808 (i.e., 0x8000,0000,0000,0000) will be used.

If the floating point value is not an integer (i.e., if it has non-zero digits to the right of the decimal point, as in 4.5) then the **NX-Inexact** bit will be set and the value will be rounded to the nearest integer, with ties to even.

The **UF-Underflow** and **DZ-Divide-by-zero** bits in **FLOAT_STATUS** will be unchanged.

Note that in the "C" language, casting a "double" to an "int64_t" handles error cases differently. Basically, the most negative value (-9,223,372,036,854,775,808 = 0x8000_0000_0000_0000) is used as an indication of any overflow or error. The conversion is done using truncation, not rounding. The flags that IEEE says are to be set (FE_INEXACT, FE_INVALID, FE_OVERFLOW) are apparently ignored.

Integer → Floating Point

The following Blitz-64 machine instruction converts from double precision to a 64-bit integer:

```
fcvtfi   r4,r5      # Floating ConVerT to Float from Integer
```

This instruction will be executed by the KPL built-in function **forceToDouble**, which is used to convert an **int** value to a **double**.[54]

The range of floating point numbers is much greater than integers, so there is no possibility of overflow. However, not all integers can be represented exactly. In some cases, the value must be rounded to the nearest floating point value.

All integers in this range can be represented exactly as floating point numbers:

Decimal		64-bit Integer	Double Precision
-2^{53}	-9,007,199,254,740,992	0xFFE0,0000,0000,0000	0xC340,0000,0000,0000
	…		
$+2^{53}$	+9,007,199,254,740,992	0x0020,0000,0000,0000	0x4340,0000,0000,0000

Most integers outside this range must be rounded, and the rounding rule (i.e., round-to-nearest, with ties to even) will be used.[55]

If the integer cannot be represented exactly, then the **NX-Inexact** bit in **FLOAT_STATUS** will be set. The other bits (**OF-Overflow**, **UF-Underflow**, **NV-Invalid**, and **DZ-Divide-by-zero**) will be unchanged.

A word-sized integer (i.e., a 32-bit signed number) is within the range:

-2,147,483,648 … +2,147,483,647

Every integer in this range can be represented exactly with a double precision floating point number, so rounding will never be necessary. The 52 bits of mantissa are more than enough to represent all possible 32 bit integers with perfect accuracy.

---

[54] This function is named "force…" rather than something like "intToDouble", since there is not always an exact conversion and accuracy may be lost. There is the possibility that the **OV-Overflow**, **NV-Invalid**, and **NX-Inexact** bits will be set. Hopefully, the name will remind the programmer of these error possibilities.

[55] Note that 2^{53} is represented as a binary integer as a "1" followed by 53 "0"s. In other words, it requires 54 bits to represent. In a double precision number , the leading "1" bit is implicit and there is enough room for up to 52 additional bits. All numbers smaller than 2^{53} can be represented with only 53 bits. But if we can accommodate only 52 bits after the leading "1", how can we accommodate 2^{53}, a 54 bit number?

Recall that after the explicit bits, we assume all bits to the right are implicitly "0"s. The value 2^{53} is an even power of 2 so, in binary, it is "1" followed only by "0"s. Although we can represent this number exactly, we cannot represent all 54 bit numbers exactly; we can only represent the even values. The next value — $2^{53}+1$ — cannot be represented exactly.

In KPL, if the programmer uses data of type **byte**, **halfword**, or **word**, code like this will work fine:

```
var
  w: word
  d: double
...
d = w
```

The compiler will use the **FCVTFI** instruction. There is no need for the programmer to use the **forceToDouble** function, since there is no possibility for inexact results or errors.

Relational Operations with NaN

There are some strange and unexpected behaviors when one of the operands to a relational comparison is NaN.

Normally, an operation in which one operand is NaN is required to yield NaN as a result. For relational operations, the result is normally a "boolean" or a branch, so it is not possible to yield a NaN.

So what happens? Here is the rule:

Every NaN shall compare unordered with everything, including itself.

In particular, we have:

Operation	Result	Invalid Operation
NaN < x	false	yes
NaN <= x	false	yes
NaN > x	false	yes
NaN >= x	false	yes
NaN == x	false	no
NaN != x	true	no

Note that this implies the following very **unexpected results**:

Operation	Result	Invalid Operation

NaN	==	NaN	false	no
NaN	!=	NaN	true	no

Normally, we accept these equivalences:

This:	is the same as:
x < y	NOT (x >= y)
x <= y	NOT (x > y)
x > y	NOT (x <= y)
x >= y	NOT (x < y)

However, these are not true when one of the operands in NaN!

Interesting Behaviors with Zero and Infinity

Zero

The bit pattern representation of zero is:

+0.0	0x0000_0000_0000_0000
-0.0	0x8000_0000_0000_0000

Note that the floating point representation for +0.0 is bit-for-bit identical to the representation for 0 in binary integer representation.

It happens to be true that -0.0 is represented identically to the most negative signed integer, but this is less useful.

Using the "==" or "!="operators in KPL will use the **FEQ** machine instruction, which follows the IEEE spec for comparing floating point values. So the following is true:

```
+0.0 == -0.0
```

If you want to distinguish these values, the use code like this:

```
if copyBitsToInt (d1) == copyBitsToInt (d2) ...
```

```
if copyBitsToInt (d1) == copyBitsToInt (-0.0) ...
```
See footnote[56]

Although +0.0 and -0.0 compare as equal, they may also result in different outcomes in some computations. This challenges our understanding of the meaning of "equal", to say the least.

Infinity

There are two infinities which are represented as follows:

+∞	`0x7FF0000000000000`
-∞	`0xFFF0000000000000`

Dividing by zero will cause the **DZ-Divide-by-zero** bit to be set and the result to be infinity. Note that:

1.0 / +0.0 yields +∞
1.0 / -0.0 yields –∞

Also:

±0.0 / ±0.0 yields **NaN** and sets the **NV-Invalid** bit

Dividing by infinity will yield zero. For example:

-0.0/-∞ yields +0.0

Not-a-Number (NaN)

There is a special value called "not-a-number", which is often abbreviated "NaN". In KPL, the predefined keyword **nan** is used. Some arithmetic operations are considered to be "undefined" and, when attempted, will result in a NaN result, to indicate that the result is undefined. Here are some examples of operations that with yield "not-a-number".

[56] You could code something like "`if copyBitsToInt (d1) == 0x8000_0000_0000_0000`..." but the compiler will generate the same code, and it seems a lot less clear.

0 / 0
∞ / ∞
0 * ∞

Other operations are mathematically defined but give a complex result, which cannot be represent. Complex numbers are not handled by floating point, so operations such as the following will return NaN.

Square root of a negative number
Log of a negative number

Another use of NaN is to represent an uninitialized or missing value. If a variable is used before it is initialized, spurious incorrect results might occur, but this can be avoided if the variable contains NaN.

There are several bit patterns that can be used to represent NaNs, so there is not a single bit pattern for NaN.

A value is defined to represent NaN if (1) the exponent field is all 1's, and (2) the bits of the fraction field are not all zero. (If the fraction bits are all zero, then the value is either +∞ or –∞.) The sign bit of a NaN value is ignored.

The KPL programmer may use the keyword **nan** in program code. The representation used here for NaN is **0x7ff8000000000000**, so the following is always true:

```
copyBitsToInt (nan) == 0x7ff8000000000000
```

The floating point operations that yield a NaN result will always use this value. However, since there are multiple bit patterns that should be interpreted as NaN, the programmer should always test for NaN with the **isnan** function, as in:

```
if isnan (d) ...
```

Signaling and Quiet NaNs

Blitz does not implement “signaling” NaNs. This section is educational and does not apply to Blitz.

Although we normally just talk about NaN, the IEEE spec actually describes two kinds of NaN: “signaling NaN” and “quiet NaN”.

A “signaling NaN” is supposed to cause a break in the flow of execution when it is encountered in a computation. That is, a trap or exception of some sort will occur, and the normal instruction sequence will be interrupted immediately. Signaling NaNs might reasonably used for uninitialized values: their use may represent a program bug which needs attention. In theory, signaling NaNs might also be used as placeholders for values (such as complex numbers) which require special handling.

The idea with a “quiet NaN”, is that it can be used as an operand in arithmetic operations and it will simply be propagated during computation. That is, if one of the operands to some operation is a quiet NaN, the result will also be a quiet NaN. This allows a lengthy sequence of operations to be performed quickly with no special testing for problems. Once a NaN appears, as a result of some error, it will persist in the chain of computations. Each subsequent operation will complete normally, without causing an exception or trap even though some sort of error occurred earlier in the sequence. If any problems occur at any step of the computation, the final result will be a quiet NaN. Therefore, it is sufficient to perform only a single test for NaN after the entire computation to see if any errors arose at any stage of the computation.

The IEEE spec does not require signaling NaNs; they are optional.

The Blitz approach is for the hardware to interpret all NaN values identically, basically as quiet NaNs.

As mentioned previously, there are several bit patterns that are interpreted as NaN; there is no single bit pattern for NaN. A value is defined to represent NaN if (1) the exponent field is all 1’s, and (2) the bits of the fraction field are not all zero. The sign bit is ignored.

If a distinction between quiet and signaling NaN is implemented — in Blitz, it is not — then one of the bits in the fraction field will be used to distinguish between quiet and signaling.

The exact bit patterns for NaN are not fully specified and can vary between implementations.

We can say that a value with all bits set to one (i.e., the representation for the signed integer -1 which is 0xFFFF FFFF or 0xFFFF FFFF FFFF FFFF) will definitely represent a NaN and will almost certainly represent a quiet NaN. For example, the all-ones pattern will be a quiet NaN for Intel, AMD, SPARC, ARM, RISC-V, etc.

Mixing Single and Double Precision Using NaN

There are many bits in the fraction field, and the only requirement for NaN is that they cannot all be zero. Thus, there is room to store some additional data within the NaN. So a NaN can carry a sort of "payload" value in the fraction bits. This capability may or may not be used in a particular implementation of IEEE 754-2008.

For example, the fraction field in a double is 52 bits. Assume that one bit is reserved to be always set to indicate that this is a NaN, and assume that a second bit is reserved and used to distinguish between a quiet NaN and a signaling NaN. This leaves 50 bits that can be used to store an arbitrary value. Notice that this is enough room to store an entire single precision floating point number.

Imagine a machine that implements double precision arithmetic and uses 64-bit registers to store floating point values. How might this machine store 32-bit single precision values in these same registers?

Any 64-bit value in which the high order 32 bits are set, will be always recognized as a NaN. One approach to storing a single precision value in a 64 bit register is to store the single precision value in the least significant bits 32 bits and all 1s in the most significant 32 bits.

All single precision operations will only look at the least significant 32-bits of the operands and, for the result value, will always set the most-significant 32 bits to 1s.

Any accidental attempt to perform a double precision operation on a register containing a single precision value, will interpret that operand as a NaN.

Normalized and Denormalized Numbers

Not every number is representable and the representable numbers are spaced out on the number line. So each possible floating point value is separated by a numerical distance from the next smallest number and from the next largest number. As the numbers get smaller and closer to zero, the spacing gets smaller and the numbers are closer together. As the numbers get larger, the spacing is farther apart.

For example, the following numbers differ by a very small amount:

4.567×10^{-25}
4.568×10^{-25}

On the other hand, these two numbers differ by a very large amount:

$4.567 \times 10^{+25}$
$4.568 \times 10^{+25}$

However in both examples above, the accuracy is the same: 4 digits of precision.

However, there is only a limited number of bits available to represent the exponents. Exponents cannot continue to get more negative and we cannot represent smaller and smaller numbers, ever more close to zero. Therefore, this pattern of the floating point numbers becoming spaced ever more closely as they get closer and closer to zero cannot continue. Something has to change as the numbers get smaller and approach zero.

What happens is that below some size, the representable values are simply spaced uniformly all the way down to zero. This is the role of denormalized numbers.

Most floating point numbers are “normal” numbers. Normal numbers have about 7 digits of accuracy (for single precision) and 16 digits of accuracy (for double precision). In other words, we can approximate any desired value with about 7 (or 16) digits of accuracy.

Another way to look at denormalized numbers is this: For very small values, we cannot approximate the value with full accuracy. As we get closer and closer to zero, we can approximate the true value with fewer and fewer places of accuracy. For really tiny values, we may even be forced to use 0.0 to represent the value, essentially losing all accuracy.

We can make the following statements about denormalized numbers:

- All denormalized numbers are very close to zero.
- Denormalized numbers extend on both the positive and negative sides of zero.
- +0.0 and -0.0 are themselves represented as denormalized numbers.
- All denormalized numbers are regularly and evenly spaced. (Exception: +0.0 and -0.0 have an infinitesimal difference and are considered equal.)
- The largest denormalized number is just less than the smallest positive normal number.
- Likewise, the most negative denormalized number is just greater than the least negative normal number.
- It is generally safe to ignore the distinction between normalized and denormalized numbers when using floating point in your applications.

There are rules for determining the precision of the results of an arithmetic calculation involving scientific notation. But if very small values (i.e., denormalized numbers) arise during a computation, then your assumptions about precision will be violated and the final results will have reduced precision. In some cases, the final result will be a meaningless, incorrect value.

Warning: Always remember that numbers as represented in computers are NOT true mathematical numbers. Computer arithmetic is NOT mathematical arithmetic. Remember: "int"s are not integers and "floats" are not real or rational numbers.

Computer values and computation are mere approximations of mathematically pure ideals. A good programmer knows how important it is to understand and remember their differences in creating reliable software.

Named Values

Here are some important constants, which can be used in KPL code:

Constant	IEEE Name[57]	Value	Approx value

[57] In order to follow the Blitz naming conventions, Blitz does not use the official names specified by IEEE 754.

FP_EPS	eps	2^{-52}	2.2204×10^{-16}
FP_MIN_NORMAL	realmin	2^{-1022}	2.2251×10^{-308}
FP_MAX_NORMAL	realmax	$(2\text{-eps})^{1023}$	$1.7977 \times 10^{+308}$
FP_MAX_DENORMAL		footnote[58]	2.2251×10^{-308}
FP_MAX_SUBNORMAL		…same…	
FP_MIN_DENORMAL		2^{-1074}	4.9407×10^{-324}
FP_MIN_SUBNORMAL		…same…	
FP_EMIN	emin	-1022	
FP_EMAX	emax	1023	

The **machine epsilon** (eps) is the distance from 1.0 to the next larger floating point number.

The KPL keywords **nan** and **inf** can be used and it is common to see the following in KPL code:

nan
+inf *or just* **inf**
-inf
+0.0 *or just* **0.0**
-0.0

For example:

```
if d == -inf
  d = nan
endIf
```

Conversion to Decimal

IEEE 754 requires an ability to convert floating point values to and from decimal character strings.

[58] The value is 4,503,599,627,370,495 × 2^{-1074}, which equals 0x0.F_FFFF_FFFF_FFFF × 2^{-1022}.

Conversion to an external character sequence followed by conversion back will recover the original value, as long as 17 decimal digits are used and "round-to-nearest with ties to even" is used.

In other words, the following conversions will yield the same value:

double → decimal character string → double

as long as the character string contains at least 17 digits, and this is true in Blitz.

The following functions are the most useful:

```
doubleToString (d: double) returns String
stringToDouble (str: String) returns double
```

In addition, the following functions allow more control over the conversion:

```
doubleToStringWithOptions (...) returns String
stringToDoubleWithOptions (...) returns bool
parseFloat (...)
```

Furthermore, double values can be converted into decimal with **printf** and **sprintf**, which will actually invoke the above functions to do the work. The function **doubleToString** is more-or-less equivalent to

```
printf ("%g", ...)
```

These functions are documented elsewhere.

Printing - %e, %f, %g

Floating point numbers can be printed with **printf** using any one of three different formatting codes. (Printing with **printf** is documented fully elsewhere.)

The simplest and recommended method is to print with format code **%g**:

```
var d: double
...
printf ("value = %g\n", d)
```

If the value is reasonable and can be printed without too many digits, it will be printed without exponent. Otherwise, the value will be printed with exponent.

```
123.456
3.456e-100
```

With formatting code **%e**, the number will always be printed with exponent:

```
printf ("value = %e\n", d)
```

For example:

```
1000.0      as printed with %g
1.0e3       as printed with %e
```

Consider the following code:

```
d = 0.3
printf ("%g")
printf ("%e")
```

The default precision with **%e** is 17 digits. This decimal value 0.3 cannot be represented exactly with a floating point number. With 17 digits, it will be printed as:

```
2.9999999999999999e-1
```

The default precision with **%g** is 16 digits. This same value will be printed as:

```
0.3
```

> *If you want exactness, use **%e**. If you want readability, use **%g**.*

With formatting code **%f**, the value will be printed with a fixed, unchanging number of digits to the right of the decimal point.

```
printf ("value = %.4f\n", d)
```

Formatting code **%f** will produce output such as the following, which can be good for printing columns.

```
0.0007
1.0000
3.1416
```

With **%e**, **%f**, and **%g**, the following option may be included:

- field width
- left justification
- precision
- separators

If a **field width** is given, as in:

```
printf (">>>%10g<<<", d)
```

the value is printed with padding blanks, as in:

```
>>>      12.5<<<
```

If the **left justification flag** (a hyphen) is included, as in:

```
printf (">>>%-10g<<<", d)
```

the output is left-justified within the field:

```
>>>12.5      <<<
```

A **precision** may be included and is written following a period character (“.”), as in:

```
printf ("%.5g", d)
```

For **%e** and **%g**, the value will be rounded to that number of digits:

```
3.1416      This is pi, rounded to the nearest 5 digits.
```

For **%f**, the precision indicates the number of digits to the right of the decimal point. For example:

```
printf ("%.5f", d)
```

prints:

```
3.14159     This is pi, printed to 5 decimal places.
```

The **separator flag** is "#". If present, the value will have separators inserted for easier reading. For example:

```
printf ("%#g", d)
```

prints like this:

```
12,345,678.0          # adds commas to the left of the decimal
57.000_000_01         # adds underscores to the right
```

The following special values may be encountered. Regardless of which formatting code is used, they will print as:[59]

```
<pos infinity>
<neg infinity>
<not-a-number>
```

The following are **configurable options**:

- How are "+infinity" and "-infinity" to be printed?
- How is "not-a-number/ NaN" to be printed?
- Is a leading "+" sign to be printed?
- Is "+" to be printed for positive exponents?

Also, the characters used as separators and the decimal point can be changed. The defaults are:

```
.     Decimal point
,     Separator to the left of the decimal point
_     Separator to the right of the decimal point
```

You can change the defaults to accommodate European conventions, as in:

[59] The brackets <> are included. We elected to use these as defaults, rather than the traditional "inf", "-inf", and "nan" strings used in Unix-like systems, but this is configurable if you prefer the shorter forms. You can even use Unicode, as in "-∞".

```
10.000.000,0
0,5
7,000 000 001
```

These settings apply to any use of **%e**, **%f**, and **%g** and can be modified if needed. This is documented elsewhere.

Chapter 8: Stack Management

Stack Usage

KPL utilizes a runtime stack. Whenever a function or method[60] is invoked, a **stack frame** is allocated. Stack frames are sometimes called **activation records**.

Local variables, parameters and the return address can often be stored only in registers, but sometimes they will need to be stored in memory. When stored in memory, they are placed in the stack.

A stack frame is created when a function is invoked, and is destroyed (i.e., deallocated) when the function returns. Stack frames are allocated by **pushing** onto the stack and they are deallocated by **popping** off the stack.

Blitz-64 has been designed so that functions can sometimes avoid allocating a stack frame. A **leaf function**[61] that requires no temporary storage can avoid creating a stack frame.

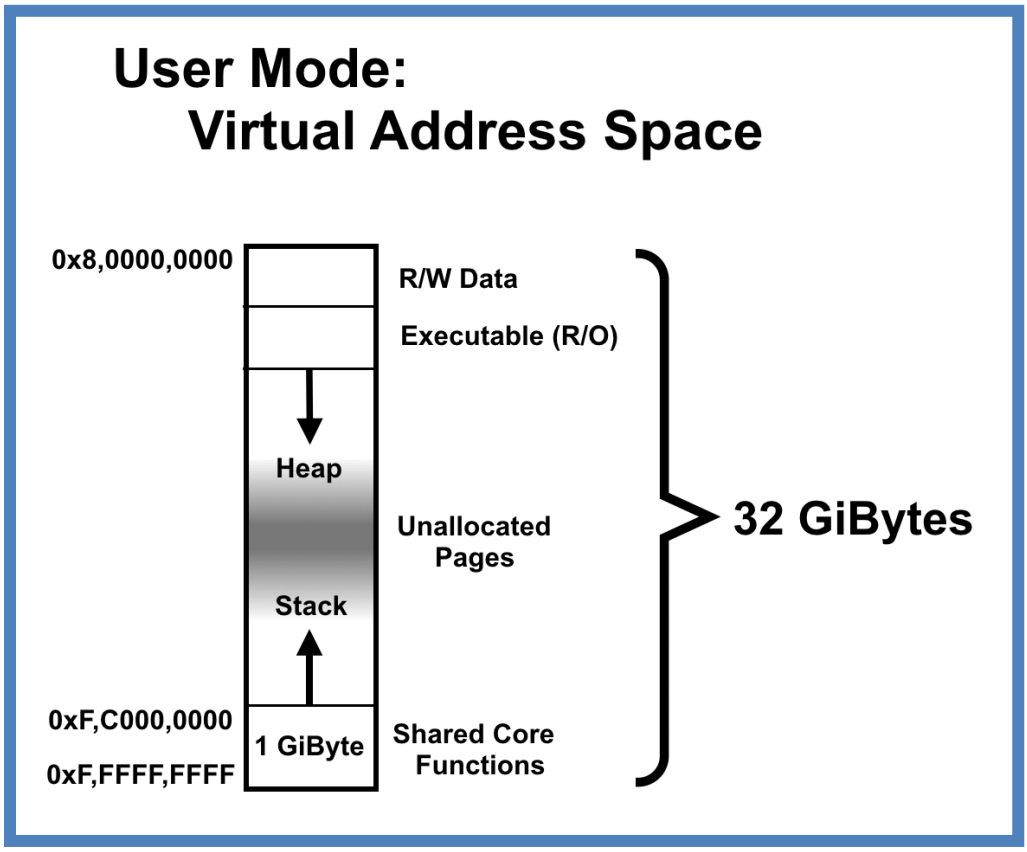
The stack frame is created by the initial code in the function, which is called the **function prologue**. This code will also zero out all the local variables, as required by KPL. The stack top is pointed to by the **stack top pointer**, which is kept in register r15 ("**sp**"). The stack grows downward from high memory so allocating a stack frame is achieved by subtracting the frame size from **sp**. Deallocating a frame is achieved by adding that same value back to **sp**.

Many programs also require **dynamically allocated memory on a heap**. In the simplest approach, the heap and the stack share a region of the virtual address space. The stack grows downward from the top of the space and the heap grows upward from the bottom of the space. This way, the available space may be used for

[60] As far as stacks are concerned, functions and methods work the same way.

[61] A leaf function is a function that does not invoke any other functions.

either stack or heap, as needed. When the stack and the heap meet, then the program has run out of memory.



Elsewhere, we discuss how the size of the stack can be bounded with the **Max_Stack_Usage** clause in KPL. The **Max_Stack_Usage** mechanism allows the programmer to determine and verify at compile-time that a program's stack will not exceed some limit.

Whether or not we know the maximum stack size before runtime, there are **several approaches** to dealing with the problem of stack memory requirements, which we discuss next.

The main issues:

- How much space to allocate for the stack?
- How to determine if the allocation is exceeded?
- What to do when the allocation is exceeded?

Stack Protocol #1

Imagine a large program with 1,000 functions and imagine that each requires 1,000 bytes, which seems quite generous. This program will never need more than 1,000,000 bytes for its stack, which is a relatively small amount of memory and a tiny fraction of the virtual address space.

The simplest approach is to preallocate a **fixed amount of memory** to the stack. The idea is that all programs will be given some default amount of space.[62]

A **default amount of 16 MiBytes** seems more than enough for non-recursive programs and probably enough for most recursive programs, as well. A 16 MiByte stack will occupy a mere 1/2,048 of the virtual address space[63] so there is plenty of room for such a stack.

Assuming that the executable code occupies 1 GiByte (a truly huge program) and that shared core functions occupy another 1 GiByte, this leaves 30 GiBytes for the heap and stack to share.

It is important to detect stack overflow, because it is important to detect all errors.

In the first approach we describe here, the virtual memory system is used to detect a stack that has grown beyond its preallocated memory region.

The idea is that the virtual address space will contain a small number of **sentinel pages**. We propose a default of 128 pages, which equates to 2 MiBytes. A sentinel page is dedicated to the purpose of **detecting stack overflow**. These pages are located just below the default allocation of the 16 MiByte stack region. The kernel will flag these pages as unmapped and any attempt to read or write to them will be caught by the kernel and will indicate that stack overflow has occurred.

In KPL, all stack frames are required to be initialized to zero values and this is done upon entry to a function. The function prologue code will write zeros into each newly allocated stack frame. This guarantees that the sentinel pages will be touched

---

[62] Our initial approach was to set the stack size to 1 MiByte, with no attempt to detect overflow. This size has proved adequate for all programs to-date.

[63] In Blitz-64, the page size is 16 KiBytes, so this is 1,024 pages out of the total of 2,097,152 pages available in the virtual address space. Also, there is negligible impact on the page table, since we assume each page can hold 2,048 page table entries.

upon function entry, at the time the stack exceeds its preallocated region, and before anything else happens.

For most programs, this protocol will be adequate. If the stack region is exceeded, it is considered a fatal error and **the thread is terminated**.

Stack Protocol #1a

The next embellishment is to allow for adjustment of the size of the stack region. Now, the programmer can choose the size of the stack region.

It may make sense to modify the format of the object and executable files to include a "Stack Allocation Size". Perhaps the compiler can automatically set this value. A reasonable approach might be to sum up the frame sizes for all functions. Using this value as a stack limit will suffice for any non-recursive program, and increasing it in some formulaic way would probably accommodate most programs with recursion.[64]

Nevertheless, with this approach, we need a way for the programmer to explicitly increase the stack space for those threads that exhibit deeper recursion than the default size accommodates.

Stack Protocol #2

The next most flexible approach is for the OS kernel to **adjust the stack region dynamically**. This removes any problems around recursive programs that have unpredictable stack needs.

With this technique, sentinel pages are used. When the current stack region is exceeded, the OS kernel will get a virtual memory exception. At that time, the kernel can increase the stack allocation and resume execution of the program. The program will not be aware of this interruption and will not be aware the stack region has suddenly grown larger.[65]

---

[64] We might go as far as to include a flag in the object and executable files to indicate whether the program can be determined to contain no recursion. For such program, no sentinel pages are necessary and we can avoid allocating them. But the sentinel pages do not impose much overhead, since the page are never populated, so it is probably not worth the trouble.

[65] The kernel will also need to set up a new sentinel region before returning to the interrupted program, so that if, in the future, the stack once again exceeds its preallocated size, the kernel can once again increase the size of the stack region.

With this approach, there must also be coordination with the memory allocated to the heap. The kernel must monitor both the size of the stack region and the size of the heap. As the heap grows, the heap manager will request additional pages from the kernel. The kernel will accommodate a growing heap, up until the heap reaches the stack sentinel pages. The kernel will allow the stack to grow, until it reaches the heap, leaving no room for the sentinel pages.

Multiple Threads

Many programs are expected to have multiple threads.

There are two ways to accommodate multiple threads and cooperating, concurrent processes.

In the first approach, at the time a new thread is created, the entire address space is copied. In this approach, there is **one thread per address space** and each thread lives in a completely separate, isolated address space. Any and all communication between the threads must involve the OS kernel, e.g., through interprocess "send" and "receive" operations.

In the second approach, **several threads share a single address space**. This approach is useful when the threads perform a lot of coordination and communication. This is the approach preferred for most Blitz-64 programs.

These two approaches make a distinction between **forking a thread** and **forking a process**. When a thread is forked, a new thread is added to the current address space. When a process is forked, a new address space is created and the new thread goes into it.

In Blitz-64, the term "**process**" means an address space and all of the threads within it. We assume that it is typical for there to be many threads within a single process.[66]

All threads within a single process will share an address space. All threads will share the read-only, executable code section and all threads will share the read-write, global data area. Furthermore, all threads in a process will share the heap space.

[66] A "process" in the sense we use it, is sometimes called a "task". Historically, the term "process" was conflated with "thread".

However, each thread must have its own stack region.

There are two approaches to accommodating programs with multiple threads, which we describe below.

- Put all stacks in the same virtual address space (protocol #3)
- Put the stacks in different address spaces (protocol #4)

Stack Protocol #3

In this approach, all stacks will be in the same virtual address space. Therefore, the stacks must occupy different address ranges. They cannot overlap.

As an example, imagine that there will be 256 threads and each stack will have a default size of 16 MiBytes. These means that 4 GiBytes of the address space must be allocated for stack storage.[67] Nevertheless, this is a modest amount of address space, leaving 28 GiBytes for the heap and other things.

This approach can accommodate a respectable number of threads (256) each with a respectable stack size (16 MiBytes). In fact, there is plenty of address space to increase these numbers, for those programs that need it.

This approach has the advantage that there is not too much interaction with the kernel when a new thread is created, and thread creation can be managed from within the process itself.[68]

Another advantage is that the threads can share data that is located on the stack. For example, one thread may create some local data on its stack and pass a pointer to that local stack data to another thread. Of course the first thread better not return before the second thread has completed using the data; it must wait for a signal from the second thread before returning![69]

---

[67] We'll also need some sentinel pages for each stack to make catch any stack overflow errors, as described previously.

[68] For example, different threads might be given stack regions of differing sizes, and these sizes can be determined by the process itself, based on information available only at runtime.

[69] The sharing of local stack data between threads is strongly discouraged, since bugs will be race-dependent and program correctness will be almost impossible to verify.

The actual parent process which creates the threads may have more information about how many threads will be created and may divide the available address space up as it sees fits, perhaps giving some threads larger stack regions than other threads.

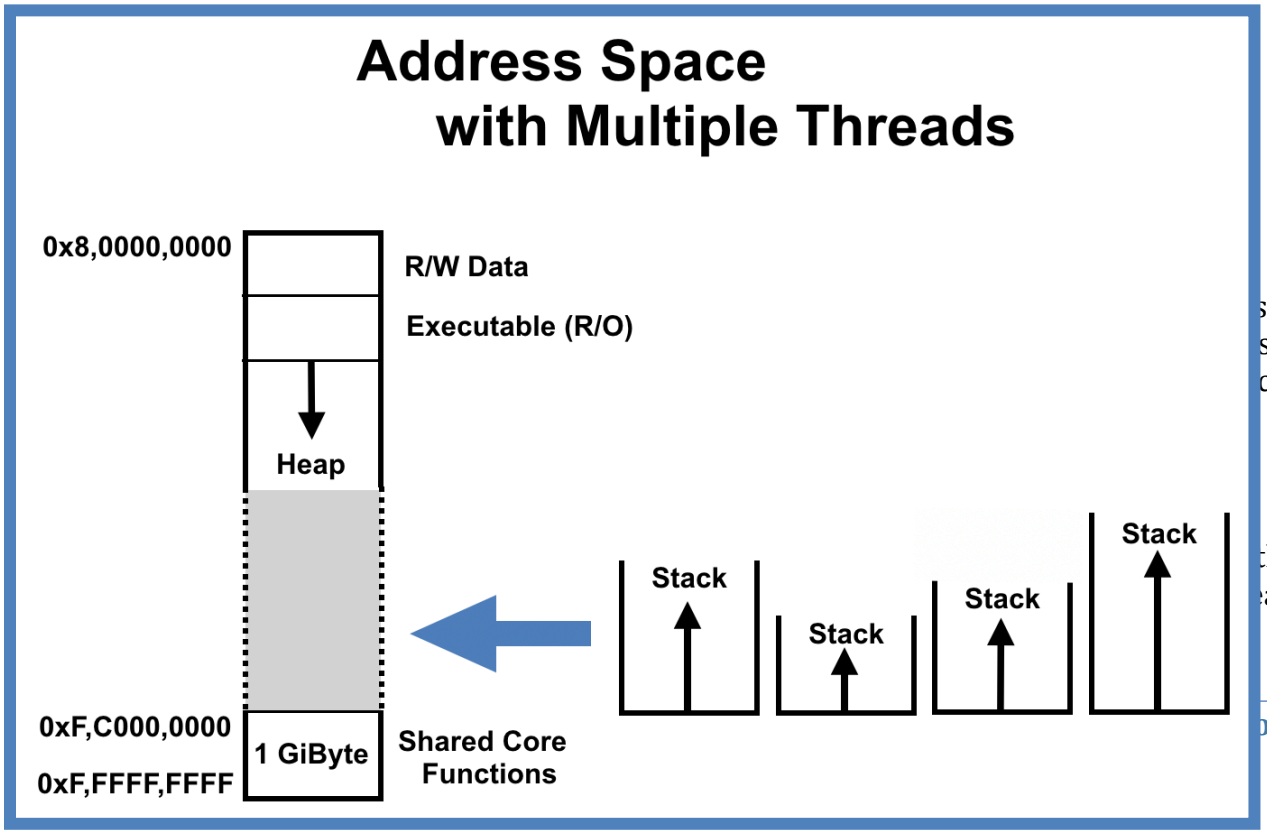
It is also reasonable for the parent process to allocate space within the heap for each thread's stack region. Then, whenever a thread completes and terminates, the parent can free that stack region, making it available for future threads.[70]

Stack Protocol #4

In this approach, all threads will share code, global data, and the heap, but the threads will not share stack space. Instead, the stacks for each thread will be located at the same address range.

In other words, the data that a given address refers to may be dependent on which thread is involved. The threads can communicate through shared global data in the R/W section and through data in the heap. But a pointer to a local stack variable is not usable in another thread.[71]

The benefit of this approach is that the parent process doesn't need to specify a maximum stack size when a thread is created. Each stack is allowed to grow until it reaches the top-most page occupied by the heap, giving the greatest room for growth.



s need
s
cated.

thread
eap

Stack Protocol #5

The previous approaches make sense for programs executed as user processes by an OS kernel that implements virtual memory. Not all programs operate in a such an environment.

KPL is also intended for standalone programs running on bare machines. In order to make the Blitz-64 system usable for embedded and single chip systems where memory and other resources are quite limited, there is another approach which we describe here.

In this approach, the programmer will determine exactly the maximum possible stack usage before runtime and the compiler will verify and check that the program meets the specification. Given this "maximum stack usage" specification, it becomes possible to preallocate a much smaller memory region for the stack with confidence that it will not overflow.

The previous approaches focussed on preallocating a large memory region for the stack, hoping that it was adequately large, and catching any stack overflow errors that occurred. If more space can be allocated to the stack region, then then thread can continue, but if not, then there is really nothing that can be done. The thread must be aborted. In some critical applications, it is unacceptable to allow the possibility — however remote it may seem — that a thread might unexpectedly run out of stack space and have to be perfunctorily terminated. Yet allocating overly large stack regions may be unacceptable.

One such program requiring this approach is the OS kernel itself.

We want to keep the memory occupied by the OS kernel as small as possible, in order to leave the maximal amount of physical memory available for use by user level address spaces. Another issue with an OS kernel is that, since it may not be memory mapped, the techniques of using sentinel pages may not work.

In the approach here, we will assume that the program exhibits no recursion. This is a reasonable assumption for OS kernels, small embedded systems, and programs demanding high reliability.

For each function or method, the programmer will determine how much stack space is required. In other words, the programmer will have to specify (in the KPL source

code) the number of bytes that will be pushed onto the stack whenever the function or method is called.

Using the Max_Stack_Usage Clause

KPL has a special syntactic construct to support this and the compiler will verify that the code meets the constraint. With the **Max_Stack_Usage** clause, the programmer can specify at compile-time how much stack space a method or function will use, and the compiler will check to verify that the method or function can never exceed this limit.

In order to use the **Max_Stack_Usage** mechanism, the programmer must first determine the size of the stack frame for each function and method in the program.

The KPL compiler has a command line option "**-stack**" which will cause the compiler to print information about the frame sizes of all functions and methods.[72]

```
% kpl MyPack … -stack
```

The maximum amount of stack space required by any function or method is the frame size plus the maximum stack usage for any function or method that might be called.

Let's see how it can be computed with an example.

Imagine our program has a function called "f" that does not call any other functions. Assume we know the frame size, from compiling with the "-stack" option:

```
f    frame size = 200 bytes
```

Then we can conclude:

```
f    max stack usage = 200 bytes
```

Next, suppose that we have a second function "g"and we know its frame size:

```
g    frame size = 100 bytes
```

[72] A language reference manual should describe the language and not a particular implementation, but this seems like the best way to motivate the KPL features involved.

Let's assume that "g" only calls "f".

If "g" is invoked, it will allocate 100 bytes and, if it happens to invoke "f", there will be an additional 200 bytes pushed onto the stack, at most. So we can conclude:

```
g    max stack usage = 300 bytes
```

Next, assume there is a third function called "h" and we know its frame size:

```
h    frame size = 1000 bytes
```

We examine the program and determine that "h" may invoke "f" and "g". Of course, "h" can only invoke one function at a time. If "h" happens to only invoke "f" on some execution, then that execution will only require 1200 bytes.

But for another execution, "h" might invoke "g" which requires more stack space. So, in the worst case, "h" can invoke "g" which will push an additional 300 bytes. Of the all the functions that "h" may invoke directly, we must look at the one that has the maximum stack usage. This is "g".

```
h    max stack usage = 1300 bytes
```

In this example, we have determined the maximum number of bytes that might be required when each of these functions is executed. Proceeding the same way, the programmer can determine the maximum stack usage for all functions and methods in the program.

The maximum stack usage for a thread's "main" function is then the greatest number of bytes required for that thread, which is the maximum stack usage we have computed. It is only an upper limit; it may be that there is no way the main function can actually consume that much stack space. However, we can be certain that it will never use more bytes.

To specify the maximum stack usage for a function, the KPL uses this syntax:

```
function ID (Args) [ returns Type ] [ '[' Max_Stack_Usage = Expr ']' ]
```

For example, functions from the example above might be declared like this:

```
functions
  f (x, y: int) returns int [ Max_Stack_Usage = 200 ]
  g (x, y: int) returns int [ Max_Stack_Usage = 300 ]
  h (x, y: int) returns int [ Max_Stack_Usage = 1000 ]
```

If the **Max_Stack_Usage** clause is used, it must appear on both the function prototype and the function itself and the values must be equal.

```
functions
  f (…) returns … [ Max_Stack_Usage = 200 ]

function f (…) returns … [ Max_Stack_Usage = 200 ]
  ...
endFunction
```

If the **Max_Stack_Usage** clause is used on a method, the clause must appear only on the prototype, not the method itself:

```
class ...
  methods
    myMeth (…) returns … [ Max_Stack_Usage = 500 ]
  ...
behavior ...
  method myMeth (…) returns …
    ...
  endMethod
  ...
```

The **Max_Stack_Usage** clause may also appear on function types, as in:

```
ptr to function (int, int) returns int [ Max_Stack_Usage = 100 ]
```

Any method or function with a **Max_Stack_Usage** clause may only invoke methods and functions that also have **Max_Stack_Usage** clauses. If a **Max_Stack_Usage** clause appears, the compiler will enforce it.

KPL's **Max_Stack_Usage** system works in practice.[73]

[73] Experience has shown that it is often time consuming to obtain tight limits. The best approach seems to be to complete all debugging and testing before adding any **Max_Stack_Usage** clauses.

The annoyance comes whenever a function is changed. Simply adding a "print" statement to some function can increase its stack usage. This has a cascading effect, causing violations of every function that invokes the modified function, and violations in every function that calls them, and so on. So adding **Max_Stack_Usage** clauses before debugging is complete causes extra work and, as far as I can tell, debugging never seems to be complete. One approach is to avoid specifying tight constraints. Instead, the idea is to add a few extra bytes to the **Max_Stack_Usage** clauses to accommodate small future changes to the code. Of course, this can translate into wasted memory in the preallocated stack region.

Chapter 9: Heap Management

Using the Heap System

In KPL, memory on the heap can be allocated with the **alloc** expression or by calling the function **MemoryAlloc**. After use, the memory should be freed with the **free** statement.

KPL supports several "heap management algorithms", which are numbered.

Heap Management Algorithm #0 is the default and will suffice for almost all programs. **Heap Management Algorithm #1** is more complex. At this writing, more complex algorithms, such as **Heap Management Algorithm #2**, which are described in this chapter, are not yet implemented.

Heap Management Algorithm #0 is the default. In it, memory is allocated sequentially and the **free** statement is a nop. A large region of memory is preallocated to contain the heap and this region does not grow. The size of the heap region is determined by the constant **HEAP_SIZE**, which is found in the **System** package. If a different size is required, this constant can be altered and the program recompiled. This approach is fastest and most efficient, works perfectly for most programs, and should be used unless there is a compelling reason otherwise.

Heap Management Algorithm #1 allocates memory sequentially and the **free** statement is a nop, just as in the default algorithm. However, a hidden "**byteCount**" field is stored with each allocation.[74] This allows the heap functions to perform some error checking which is not possible in the default algorithm. In particular, attempts

[74] The **byteCount** field is a doubleword preceding the chunk of memory returned to the application program. The pointer returned from a memory request will point to the address following this hidden field. The application program should ignore this hidden field, effectively remaining unaware of its existence.

to free memory more than once will be caught. Also, if a memory leak[75] is detected, it is possible with this algorithm to go through the heap sequentially and identify the objects that have not been freed. Getting information about these objects can assist in locating the leak.

Just as in the default approach, a large region of memory is preallocated to contain the heap and this region does not grow. The initial size of the heap region can be adjusted with the constant **HEAP_SIZE**.

Heap algorithms #0 and #1 are appropriate for use in both the kernel and user-level programs. Algorithm #2 is only for user-level programs.

With **Heap Management Algorithm #2**, the size of the heap region is not fixed. Instead, the heap will grow as necessary. The algorithm will request additional pages from the OS kernel, potentially consuming the vast stretch of virtual memory between the heap and the stack. Also, the **free** operation is no longer a nop. Requests by the application for memory will either be satisfied by reusing previously freed memory or be satisfied by requesting additional pages from the OS kernel.

If the programmer wishes to use anything other than the default algorithm, the function **initializeHeap** should be called before any allocations are attempted.[76] The argument to **initializeHeap** is an integer with the meaning

0: Default Heap Management Algorithm
1: Heap Management Algorithm #1
2: Heap Management Algorithm #2
…

[75] A "memory leak" occurs when a chunk of memory is allocated but never freed. With a memory leak bug, the heap will gradually will fill up with unused objects, ultimately causing problems.

[76] During KPL initialization, **initializeHeap (0)** will always be called, so the programmer does not need to invoke **initializeHeap** if the default algorithm is sufficient.

At any time, the programmer may invoke the following functions:

```
getHeapRemaining        () returns int
getHeapCurrentInUse     () returns int
getHeapTotalAllocation  () returns int
getHeapTotalFreed       () returns int
checkHeapConsistency    ()
runThruHeap             ()
```

With algorithms #0 and #1, in which the **free** operation is a nop, the **getHeapRemaining** function returns the number of unallocated bytes remaining in the heap. The **getHeapCurrentInUse** function returns the number of bytes that have been allocated by the application program but not yet freed. The **getHeapTotalAllocation** and **getHeapTotalFreed** functions give the number of bytes that have been allocated and freed over the lifetime of the program.

The **checkHeapConsistency** function would not normally be called, but may be of some use if there are bugs and the programmer suspects that the hidden heap structures have somehow become corrupted.[77]

In heap management algorithm #1 is in use and a memory leak has been detected, the programmer can invoke **runThruHeap**. This function will print some information about the chunks of memory that have been allocated but not freed, which can aid debugging.

If the heap fills up and there is no remaining space to satisfy an allocation request from the application code, the following error will be thrown:

ERROR_HeapFull

With algorithms #0 and #1, this error is real possibility and can be dealt with by modifying HEAP_SIZE and recompiling the program. With algorithm #2—the algorithm appropriate for user-level code which is only limited by the size of the virtual address space—this error will probably not occur before other errors related to address space page limits are detected.

[77] For example, if the program is writing to memory chunks that have already been freed, there is a possibility that the **checkHeapConsistency** function could catch this, if heap management algorithm #1 is in use.

The program can catch **ERROR_HeapFull** using KPL's **try-throw-catch** mechanism, or can accept the normal error handling process.

If the heap algorithms detect other errors, then the following error will be thrown:

ERROR_HeapViolation

There is an error message parameter giving more info. Example violations include (1) a negative byteCount in a request and (2) the detection of a heap inconsistency.

Allocating and Freeing

Recall that in KPL there are two ways to allocate memory on the heap.

First, the **alloc** clause can be executed, as in:

```
objPtr = alloc MyClass { ... }
arrPtr = alloc array of int { k of -1 }
```

The **alloc** construct allows for additional type-specific initialization and the allocated data is guaranteed to be initialized with zero values if no initialization is present.

Second, the program can explicitly call the function **MemoryAlloc**:

```
var p: ptr to byte
p = MemoryAlloc (n)   -- n = number of bytes to allocate
```

The **MemoryAlloc** simply returns a pointer to the new memory region[78].

Any memory allocated by **MemoryAlloc** is *not* guaranteed to be zeroed before use, although the **alloc** clause is guaranteed to initialize memory.

In either case, if the heap is full and no more memory remains, the error **ERROR_HeapFull** will be thrown.

[78] The request can be for any positive number of bytes; it need not be a multiple of 8 but the amount of memory allocated will be rounded up. The returned address will always be doubleword aligned.

To deallocate the space, the **free** statement can be used, or the function **MemoryFree** may be called:[79]

```
free objPtr
free arrPtr
...
MemoryFree (p)
```

NOTE: Currently, the stack usage of the **MemoryAlloc** and **MemoryFree** functions is not considered when the compiler verifies the **Max_Stack_Usage** clauses. Functions like this impose a small, fixed overhead on all functions. Changing the algorithm to something unusual could cause this fixed overhead to be exceeded.

Selecting a Heap Management Algorithm

In KPL, the function type makes it possible to reconfigure the heap management algorithm. The actual algorithm employed whenever **MemoryAlloc** or **MemoryFree** is invoked may be changed by the programmer.[80]

The following fields in the **ThreadData** object determine which functions are actually invoked whenever an **alloc** is executed or whenever **MemoryAlloc** or **MemoryFree** is called:

```
memAllocFun: ptr to function (int) returns ptr to byte
memFreeFun:  ptr to function (ptr to byte)
```

The function **KPLInitialize** contains this code:

```
memAllocFun = KPLDefaultMemoryAlloc
memFreeFun  = KPLDefaultMemoryFree
```

There are a number of algorithms to manage heap memory. The default is described next.

---

[79] The **free** statement simply invokes **MemoryFree**. Either technique can be used to return memory to the heap, regardless of whether it was allocated with **alloc** or **MemoryAlloc**.

[80] In fact, different threads within a single process may even use different algorithms, although that would be very unusual.

We'll refer to the code invoking **MemoryAlloc** and **MemoryFree** as the "application program" to distinguish it from the heap management algorithms.

Heap Algorithm #0

In this section we describe and discuss the simplest approach to heap management, which is implemented in the functions **KPLDefaultMemoryAlloc** and **KPLDefaultMemoryFree**.

The idea is that a large region of memory is preallocated to contain the heap. Newly allocated objects are placed one-after-the-other. The default **MemoryFree** operation is ignored and is a nop.

The heap will grow upward until space is exhausted, at which time the **ERROR_HeapFull** is thrown.

The default approach has a number of benefits.

KPL specifies that the user address space will be initialized to zeros. All pages must be zeroed by the OS kernel before being added to a virtual address space to prevent information from leaking from one process to another. This default approach is optimal, since there is no need for any additional zeroing of memory. Regions are only allocated once, and will have been zeroed by the kernel, so all allocations take a fixed quick time, independent of the number of bytes being allocated.

The allocation is extremely quick, since it requires nothing more than a couple of additions and a limit check.

All available memory in the heap region is devoted to user data; additional data structures and hidden header words are not needed.

The free operation is fast since it is a nop. Ultimately, the process will terminate and, at that time, all pages from the address space will be returned to the kernel's pool of free pages. Thus, the time spent on memory reclamation is truly minimal.

In the very simplest approach, the heap space can be predefined, in which case there is no interaction with the OS. The heap is essentially a fixed array, allocated when the address space is created. This situation is really the simplest and has worked surprisingly well for a number of complex programs to-date.

If the application program is running in a user-level virtual address space, we can grow the heap as needed. So we start with a small heap and, when it is all used up, we add pages. With this approach, the heap can grow and grow until either it hits the stack or it reaches some page limit imposed by the OS.

The best approach is to keep track of the current top of the heap, and have **MemoryAlloc** explicitly ask the OS for additional pages when they are needed.

An alternate approach is to simply use any and all memory in the virtual address space with explicitly coordinating with the OS. Initially, **MemoryAlloc** will only touch pages in the small region of memory where the heap starts, but as the application requires more memory, higher and higher addresses will be accessed. The OS would presumably allocate physical pages to populate the address space "on the fly" as they are needed and accessed. If the OS implements sentinel pages for the stack, then a heap overrun will be detected when the heap runs into the stack or—more precisely—when the heap runs into the stack sentinel pages.[81]

The Blitz-64 architecture limits the size of a virtual address spaces to 32 GiBytes. This size permits a lot of data, which should suffice for all reasonable program.[82] In situations where this limit becomes insufficient, it's time to break the program into multiple processes.

Many programs tend to have two phases. Roughly speaking, in the first phase, a data structure is created. During this phase, the program allocates memory from the heap. In the second phase, the program tends to process the data structure it created

---

[81] The problem here is that the application might request a huge amount of memory in such a way that the newly allocated chunk of memory includes any and all sentinel pages and runs into the stack region. The application might choose to begin using the higher addresses in this newly allocated chunk, which are actually bytes in use by the stack. This is a very unlikely scenario, but it's just the kind of thing that is incompatible with the Blitz philosophy. Therefore, the heap allocation algorithm must coordinate with the OS kernel to verify that the heap top remains below the stack sentinel pages.

[82] What is a "reasonable sized" program? Imagine a program with 10 million lines of code, with 100 bytes of machine instructions per line, which is truly huge. Yet the code only consumes 1 GiByte. As for data, imagine storing the novel "War and Peace" in memory, consuming 10 bytes per character, which is generous. You can store 10 copies of "War and Peace" in under 1 GiByte, so after code and data, there is still huge amount of space remaining. In fact, you've still got enough room for several hours of HD-quality video. A 32 GiByte virtual address space is really, really large. And keep in mind this limit is only *per process.*

earlier. All data accumulated is kept around and nothing is freed, since it is difficult to know which data items will be needed in the future.

For programs like this, there is really no good reason to free memory. It is more efficient to free all memory in bulk, when the program terminates. Furthermore, freeing memory will not make any difference. If some execution of the program was destined to run out of memory, freeing wouldn't have helped anyway, if all the allocations precede the first "free" operation.

However, a few programs do not have this behavior. For example, long-running programs can pose a problem because they are periodically allocating and then freeing memory, as demands to the program come and go. If the memory is not freed for such programs, then eventually the heap space will become exhausted.[83]

In the initial Blitz implementation, the heap size was fixed at 1,000,000 bytes by a constant in **System.c**. This size has sufficed for all programs to-date.[84]

The "allocate-and-never-free" approach seems very appropriate for an OS kernel, where we have these goals:

- Allocate as much memory to user space as possible
- Avoid any complex heap algorithm, which might introduce unpredictable delays

Management of kernel resources (e.g., page frames) inside the kernel will likely be handled differently than a simple heap, but the kernel may still need a traditional heap, particularly during start-up. The idea is that a small number of objects will be allocated during startup but after process scheduling begins, no new heap space will be allocated. So a heap that grows initially can be used to allocate the basic objects used by the kernel, but then the heap is "closed for business" and all remaining space is reallocated to the paging system.

For most user-level programs, the allocate-and-never-free approach is perfect. But for a few user-level programs, it's clearly inadequate.

---

[83] Usually, the program is going to fail because it requires so many pages that it exceeds some limit imposed by the OS and is aborted as a result. But it is also possible that the 32 GiByte virtual address space limit itself might cause the failure.

[84] A chess-playing program challenged this rather modest heap limit. But even with such a small heap size, the limiting factor for this program was processing speed, not memory usage.

Heap Algorithm #1

A **memory leak** occurs when a program fails to free memory that it no longer needs. It is important to verify that programs do not contain memory leaks.

Although we can often avoid the complexity of a heap algorithm that reclaims and re-uses free space, we still want to develop code that contains no memory leaks, under the assumption that our code will, in the future, be used in contexts where memory is reclaimed and re-used.

In the approach described in this section, we will use a simple algorithm that will allow us to detect memory leaks.

Our program will invoke **MemoryFree** to release all unused memory. As with the simplest heap approach, the **MemoryFree** operation will be nop, in the sense that we don't actually reclaim the memory space.

However we will add a way to verify that everything we allocate is ultimately freed.

Just as in **Heap Algorithm #0**, the **MemoryAlloc** function will allocate memory in sequential order, at increasing memory addresses, until memory is exhausted. After a region of memory is freed with the **MemoryFree** function, it will never be re-used.

However, we will keep track of how much memory has been freed, so that at any time, we can answer the following questions:

- How many bytes have been allocated?
- How many bytes have been freed?
- How many bytes remain in use?

If any bytes remain "in use" in an application that was supposed to clean up after itself, then we have a memory leak.

To implement this, we will add a "hidden" field to every allocation. The hidden field will be a doubleword and will contain the number of bytes in the memory region.

For example, if the call to **MemoryAlloc** asks for 40 bytes[85], a region of 48 bytes will be carved out of the heap. The first 8 bytes will become the hidden "**byteCount**" field and are filled in by **MemoryAlloc**. Then **MemoryAlloc** will return a pointer to the second doubleword, i.e., offset 8. The calling program will be unaware of the hidden **byteCount** field and it should remain unchanged until the chunk of memory is freed.[86]

At some later time, the program will issue a call to **MemoryFree**, providing a pointer to the memory, which will of course be the address of the second doubleword in the region of 48 bytes. By retrieving the hidden **byteCount**, **MemoryFree** can adjust the counters to track the number of bytes that have been freed.

With this technique, we can also detect any attempt to free memory that has previously been freed. The idea is that **MemoryFree** will also mark the **byteCount** field to indicate that this chunk has been freed. The simplest way to mark it is to negate the value and re-store it in the **byteCount** field.

Then, if the program ever calls **MemoryFree**, providing a pointer to a region with a negative **byteCount**, **MemoryFree** can immediately detect this and issue an error saying that this chunk of memory has already been freed.

With this technique, we might also be able to obtain some useful information after a memory leak has been detected. Here's how.

As the program runs and makes calls to **MemoryAlloc**, chunks of memory will be allocated sequentially and adjacently, from the beginning of the heap, on up. Each chunk will begin with a **byteCount**, thus allowing us to run through the heap, chuck-by-chunk, from the first allocation to the last. We can detect which chucks have been freed and which are still in use by checking whether the **byteCount** is negative or not. When we encounter a chuck that is still in use—that is, with a positive **byteCount**—we can print out the bytes. If it is an object, we might consider following its dispatch table pointer to print the class name. Otherwise, we might try interpreting the chuck as an array of bytes and printing it.

---

[85] As stated elsewhere, all allocations are in multiples of 64-bit doublewords.

[86] We make the assumption the application program is well-behaved and doesn't touch memory addresses that have not been legitimately allocated to it, but of course this sort of bug can happen in KPL and the application program might mangle the hidden **byteCount**, which will cause total havoc with the heap management algorithm.

Types of Heap Errors

In Blitz, memory is allocated on the heap through invocations of **MemoryAlloc** and released with calls to **MemoryFree**.

Some programs simply avoid ever calling **MemoryFree**. This is a perfectly reasonable approach in almost all situations and avoids bugs.

But programs that actively free memory can suffer from several types of bugs:

- **Premature Frees**: Memory is freed too soon
- **Memory Leaks**: Memory is freed too late
- **Multiple Frees**: Memory is freed more than once
- **Random Frees**: A bad address is provided

The first error is **premature freeing**. A region of memory is freed, although the program maintains a pointer to the region and subsequently reads or writes to the region.

Premature freeing is a nasty bug. Unfortunately, the program may often work correctly, in spite of this presence of this bug. Perhaps the prematurely freed region was not re-allocated before being reused so the program happens to produce the correct output. Or perhaps the region contains some new, different data. Perhaps a read from the region happens to return a reasonable but incorrect value; perhaps a write to the region overwrites some data with a reasonable but incorrect value. In short, the program with this bug can often continue to work correctly. Or the program can work incorrectly without being noticed, which is more insidious.

A premature free can easily go unnoticed, producing incorrect results. The bug can also be intermittent, causing errors and failures on some runs and not others, due to the vagaries of memory allocations.

Once the bug is noticed, it can be exceedingly difficult to track down and understand. The bug might be manifested because some data is mysteriously incorrect, as the result of a write to a prematurely free memory region. The actual bug is entirely unrelated to the data that was damaged and may have happened much earlier in execution.

Another thing that makes premature free bugs particularly difficult to track down is the fact that they can be very sensitive to memory layout. A small change to the

program can change the buggy behavior, making the bug intermittent. For example, adding a print statement to the program during debugging can subtly change where things are placed in memory, causing the bug to seem to disappear.

With “safe” and “unsafe” constructs, we have mostly assumed that a program containing only safe constructs can not crash. If there are errors, then either the program will output incorrect results or an appropriate error message will be printed, but it will not lose control and exhibit unpredictable behavior.

Unfortunately, a premature free bug violates this assumption. Consider a situation in which the program retains a pointer to a region of memory that is freed; then some new object is allocated in that same memory region; finally, the original pointer is used to store into memory, randomly overwriting fields in that new object. This can lead to a crash, even though the program only contains safe constructs[87].

The bug of premature freeing is vile, abominable, and very dangerous and is a very good reason to use languages with automatic garbage collection, where it cannot occur. Of course, if the **MemoryFree** operation is implemented as a nop, as in **Heap Algorithm #0** and **Heap Algorithm #1**, the safety guarantee is preserved.

The second error is a **memory leak**. In this case, the program fails to free a memory region that is no longer needed. As a result, nothing happens for a while. Instead, the heap gradually fills up with old, unneeded data, until the heap limit is exceeded and the program fails.

In some sense, a memory leak bug is less serious since the program output will be correct and nothing will happen until the program exhausts the available heap space. If the program produces output, it will at least be correct.

However, a memory leak bug can be very difficult to find. Typically, the heap is not exhausted immediately; the memory leak is slow and it may take quite some time before the program fails. This complicates debugging.

The first problem is to determine what sorts of things were left in memory, what was not freed that should have been. The second challenge is to determine why those objects were not freed. The programmer is essentially looking for the something that

[87] Use of the **safe** statement requires the **-unsafe** option when compiling, but calling **MemoryFree** does not require the **-unsafe** option.

is not there. Finding something that is missing is tricky, because it is hard to know where to look!

The third error might be called **multiple freeing**. The problem is that program attempts to free a region it has already freed. This is generally detectable, as with **Heap Algorithm #1**, discussed above.

A final error might be called **random freeing**. The problem is that the program calls **MemoryFree** providing a completely random address. This address was never returned from a call to **MemoryAlloc**. The address may or may not even be in the heap.

Generally speaking, this error is rare and checking for it may impose an unacceptable overhead. So it usually makes sense to avoid checking for it, letting the program fail without good error detection when it occurs.

Of course, calling **MemoryFree** with a null pointer may be common, but this will elicit an error with a reasonable error message.

Given all these factors, the **conclusion** is this: The simplest memory allocation algorithm (in which **nothing is ever freed**) is almost always **the best solution**. It's easy and efficient and doesn't suffer from the bugs of premature freeing, memory leak, and multiple freeing. This is the default KPL memory management algorithm and it should be used unless there is a compelling reason otherwise.

If the simple approach will not work, KPL allows the programmer to implement whichever the memory management algorithm makes the most sense for the application.

Heap Algorithm #2

Next, we describe the second approach Blitz will use to manage user-level heaps.

Each memory chunk—either allocated or free—will begin with a **byteCount** field in the first doubleword. (Upon return from **MemoryAlloc**, the caller will receive a pointer to the second doubleword so the **byteCount** will be a hidden field from the user program's perspective.)

When a chunk of memory is returned via **MemoryFree**, the **byteCount** will be negated, which will flag this chunk as "free".

Chunks will be allocated sequentially in the region dedicated to the heap. When the heap region fills up, we will have the option of growing the heap or reclaiming free space.

When the heap region fills up, our initial approach will be to grow the region. That is, the **MemoryAlloc** function will just ask the OS kernel for more pages. The heap region will then be enlarged and the allocation will proceed. In this initial "growth era", **MemoryAlloc** will ignore the free chunks. Any chunk returned via a call to **MemoryFree** will be marked with a negative **byteCount**, which is fast, but is otherwise ignored.

The benefit of this approach is that for the majority of programs (which never allocate huge amounts of memory), **MemoryAlloc** and **MemoryFree** will be fast.

At some point, we will reach a numerical page boundary which will signal an end to the initial "growth era". Instead of simply growing the heap region, we begin the "reclamation era".

Now, whenever **MemoryAlloc** is invoked, we will attempt to deliver a chunk from memory that has been freed previously. Only when this fails, will we ask the OS kernel for additional pages.

At all times (i.e., from the very first call to **MemoryAlloc**) we will maintain these variables:

HeapRegionStart	Address of the first byte of the heap region
HeapNextPtr	Address of the first byte beyond all free and allocated chunks
HeapRegionBeyond	Address of next byte just beyond the heap region
HeapBytesAllocated	Number of bytes in all allocated chunks
HeapBytesFree	Number of bytes in all free chunks
HeapTotalAllocation	Number of bytes allocated, including those later freed

At any time, these variables can be used to answer the questions[88]:

- How many bytes have been allocated?
- How many bytes have been freed?
- How many bytes remain in use?

During **MemoryAlloc**, we first determine if an allocation can be satisfied by just advancing **HeapNextPtr**. If it doesn't exceed **HeapRegionBeyond** then we simply carve out a new chunk, advance **HeapNextPtr**, and return.

If we are in the "growth era", we ask whether we are still below the limit and can remain in the "growth era". We will call this number

HEAP_GROWTH_LIMIT

If we are below this limit and don't have enough bytes at the top of the heap at **HeapNextPtr**, then we immediately call the OS kernel to obtain more pages. This allows us to satisfy the demand fairly quickly.

We will just keep doubling the size of the heap region until we reach the growth limit. So the heap region can start at (say) 1 MiByte and then, when this is found to be insufficient, we begin doubling the heap size. Of course, if the request to **MemoryAlloc** is larger than the heap region, we'll use that instead as the number of bytes to request from the OS kernel.

However, if the heap region size is has reached the **HEAP_GROWTH_LIMIT**, the "growth era" has come to an end and the "reclamation era" has begun. This is discussed next.

First, we look at whether the amount of space remaining between **HeapNextPtr** and **HeapRegionBeyond** plus **HeapBytesFree** is enough to satisfy the request. If so, then it is possible that we can find a chunk large enough to satisfy the request. We'll describe "searching the free space" in a moment.

---

[88] The answer to "How many bytes have been allocated?" is **HeapTotalAllocation** and includes the number of bytes allocated since the program began, including bytes subsequently freed. The answer to "How many bytes remain in use?" is **HeapBytesAllocated**. The difference between these two numbers provides the answer to "How many bytes have been freed?" We can also answer this question:"What percent of the current heap is free and available at this time, i.e., how efficiently is heap memory utilized?"

But if the amount of free space remaining is too small, no amount of searching will succeed, so we must ask the OS kernel for additional pages to add to the heap region. We assume that this will succeed until our address space is exhausted or we've reached some page limit imposed by the OS kernel. If the kernel cannot give us enough pages, then we will throw **ERROR_HeapFull** if the program is not terminated altogether.[89]

So here is how the search goes.

We start at the beginning of the heap region and go through it sequentially, chunk by chunk. Each chunk of memory in the heap will begin with a **byteCount**. If negative, then this chunk has been marked as free. As we proceed through the heap, we will watch for two free chunks in a row. Whenever we find two adjacent free chunks, we will coalesce/combine them into a single free chunk.

If we find a free chunk of adequate size to fulfill the **MemoryAlloc** request, we will stop and return the memory chunk.

If the free memory chunk is exactly the right size, we can return it as-is. Otherwise, we will split the free chunk into two pieces. We will set the size of the first piece to the desired size and the size of the second piece to the remainder. We mark the second chunk as free and we return the first chunk as the result.

(However, if the remainder from such a split would be smaller than the minimum chunk size of 16 bytes, we can't split it. We just ignore this chunk, and continue with the search.)

At any time in this search, we are positioned on the "current chunk". Once we find a chunk to satisfy the allocation request, we remember the address of the following chunk so, when **MemoryAlloc** is subsequently invoked, we can begin a new search from right after the previous allocation. Right before returning, we advance the "current chunk" pointer to the next chunk (which may be the second half of a free chunk that we just split) and save it so that the next search will begin there.

---

[89] It might be tempting to always perform the search for free space before the call to the OS kernel for more pages, even when it will definitely fail. The reason is that there may be some free chunks at the top of the heap region. By performing the search, we would coalesce these free chunks into a single free chunk as a side-effect. This would allow them to be combined with the free space obtained from the kernel call, possibly resulting in greater memory utilization. But we won't do this. It is likely that the recent calls to **MemoryAlloc** will have performed searches for free memory, which will have resulted in coalescing the top free chunks anyway.

When the search reaches the end of the heap, i.e., **HeapNextPtr**, we see if the last chunk was free. If so, we lower **HeapNextPoint** to include it, essentially eliminating any free chunk at the top of the heap region. In any case, we then reset the pointer to **HeapRegionStart** and continue the search.

We must remember where we started the search. If we reach the starting point without finding a chunk big enough to satisfy the request, then have wrapped around and searched the entire heap. At this point, we must invoke the OS kernel asking for more pages to add to the heap region.

By remembering where we left off on the last search and by wrapping around and searching until we reach our starting point, we are implementing a "next-fit" search.

Heap Consistency Checking

During debugging, the programmer may need to detect heap problems such as:

- **Premature Frees**: Memory is freed too soon
- **Memory Leaks**: Memory is freed too late
- **Multiple Frees**: Memory is freed more than once
- **Random Frees**: A bad address is provided

With the approach presented above, we have the ability to answer "How many bytes are currently allocated?" This helps in detecting **memory leaks**.

We can detect when **multiple frees** fairly easily. Each free chunk has a negative **byteCount**, while every allocated chunk has a positive **byteCount**. The **MemoryFree** operation will check the sign of the **byteCount** and signal an error if it is ever negative.

During **MemoryFree**, we can perform some checks that might catch **random frees**. For example, the address provided must be a doubleword aligned address within the heap region. When we fetch the **byteCount**, it is expected to be a positive value evenly divisible by 8, such that, when added to the chunk's address, it does not exceed **HeapNextPtr**. But random frees are less common and it is not worth it to perform these checks on every call to **MemoryFree**.

Premature frees are impossible to detect within **MemoryFree**, because nothing is really wrong until the application program tries read or write newly freed chunk of memory.

Due to bugs in the application program, it's possible that the heap will become corrupted. We can create a function (**HeapConsistencyCheck**) which will run through the heap and look for problems. Programmers might find this function useful when debugging programs with particularly nasty heap errors.

The **HeapConsistencyCheck** function runs through the entire heap region and checks all **byteCounts**. As mentioned, each **byteCount** should be a number divisible by 8 which, when added to the chunk's address, does not exceed **HeapNextPtr**. By noticing which **byteCounts** are positive and which are negative, we can recompute **HeapBytesAllocated** and **HeapBytesFree**. If these values are incorrect, then it indicates something went wrong.

It is conceivable that a **premature free** error will be caught by the **HeapConsistencyCheck** function, but not likely. Normally, the user program will not modify the **byteCounts** so, even if the chunk is freed and then accessed illegally, the **byteCount** is unlikely to be affected. Although unlikely, we might see a consistency failure if chunks are split and coalesced in such a ways that a **byteCount** field is at an address that was previously in the middle of a previously allocated chunk.

If we are serious about catching heap errors, particularly **premature free** errors, we need to modify the **MemoryFree** function to overwrite the entire chunk to be freed with a known garbage value. Such a value is sometimes called a "sentinel value"[90]. If the application program reads from the chunk after it was freed, it will retrieve this value which will hopefully cause an obvious program failure. We can also modify the **HeapConsistencyCheck** function to check to make sure every free chunk contains only this special sentinel value. This can catch writes to previously freed memory, as long as that memory has not already been reallocated by the application program.

To aid in debugging, we may also want to modify the **MemoryAlloc** function to initialize all newly allocated memory. Before each newly allocated chunk is returned

[90] The sentinel value should be an easily recognizable number, such as 0x7777777777777777. This is an odd number that, if used as an address is likely to cause an alignment exception. If used in an arithmetic computation, it may remain recognizable in the result, and will stand out in during debugging. If interpreted as ASCII, it might appear as "wwwwwwww".

to the application program, every byte will be filled with a sentinel value.[91] If this sentinel value ever appears within the program, we can assume that the program is picking up unwritten data. In other words, it is reading from bytes before initializing them. Recall that **MemoryAlloc** is NOT guaranteed to initialize memory to zeros. However, newly allocated memory will almost always contain zeros, because the heap memory is coming from fresh pages in the address space, which are guaranteed to be zeroed first. Thus, the use of uninitialized memory can escape detection until it happens that the memory chunk is being recycled by the heap manager and contains preexisting values.

[91] We can use the same sentinel value to keep things simple, or use a different value to help distinguish between a **premature free** error and the use of uninitialized memory.

About This Document

Document Revision History / Permission to Copy

Version numbers are not used to identify revisions to this document. Instead the date and the author's name is used. The document history is:

Date	Author
22 March 2021	Harry H. Porter III <Document created>
17 June 2021	Harry H. Porter III
10 November 2022	Harry H. Porter III <current version>

In the spirit of the open-source and free software movements, the author grants permission to freely copy and/or modify this document, with the following requirement:

> ***You must not alter this section, except to add to the revision history. You must append your date/name to the revision history.***

Any material lifted should be referenced.

Corrections and Errors

Please contact the author if you find…

- Inaccurate information that you can correct
- Incomplete information that you can fill in
- Confusing text that needs to be reworded

Thanks!

About the Author

Professor Harry H. Porter III teaches in the Department of Computer Science at Portland State University. He has produced several video courses, notably on the Theory of Computation. Recently he built a complete computer using the relay technology of the 1940s. The computer has eight general purpose 8 bit registers, a 16 bit program counter, and a complete instruction set, all housed in mahogany cabinets as shown. Porter also designed and constructed the Blitz System, a collection of software designed to support a university-level course on Operating Systems. Using the software, students implement a small, but complete, time-sliced, VM-based operating system kernel. Porter has habit of designing and implementing programming languages, the most recent being a language specifically targeted at kernel implementation.

Porter holds an Sc.B. from Brown University and a Ph.D. from the Oregon Graduate Center.

Porter lives in Portland, Oregon. When not trying to figure out how his computer works, he skis, hikes, travels, and spends time with his children building things.

Professor Porter's website: www.cecs.pdx.edu/~harry