Technical Report

UCAM-CL-TR-30
ISSN 1476-2986

Number 30

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A portable BCPL library

John Wilkes

October 1982

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge
Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Contents

1. Introduction

2. Standard Facilities
   2.1 Header files and conditional compilation tags
   2.2 Standard manifests
   2.3 Initialization and termination
   2.4 The stack
   2.5 The heap
   2.6 Character and string manipulation
   2.7 Globals
   2.8 Time and date
   2.9 The exception system
   2.10 Coroutines
   2.11 Miscellaneous

3. Input/Output
   3.1 Library-provided streams
   3.2 Controlling streams
   3.3 Character mode
   3.4 Binary mode
   3.5 Binary mode bulk transfers
   3.6 Character mode input functions
   3.7 Character mode output functions
   3.8 A counting filter

4. Error Handling
   4.1 The default handlers
   4.2 Manifests for system errors
   4.3 Hardware detected errors

5. Internal Details
   5.1 Program.information
   5.2 Debugging
   5.3 Stream Descriptor Block format
   5.4 Opening and closing streams
   5.5 Reading from streams
   5.6 Writing to streams
   5.7 Miscellaneous

6. Extensions
   6.1 Floating point
   6.2 Record I/O
   6.3 Interval timer
   6.4 Random access

7. Implementation notes
   7.1 Existing implementations

A portable BCPL library

Abstract

Too often, programs written in BCPL are difficult to port from one system to another, not because of the language, but because of differences between 'standard' libraries. Almost without exception, the definitions of these libraries are loose, woolly and inaccurate--the proposed BCPL standards document being a prime example. The author has developed and implemented a new BCPL library which is explicitly designed to aid the portability of programs between systems. In addition to being largely portable itself, its has two other features of interest: it uses an exception handling system instead of return codes, and it makes no distinction between system and user defined stream handlers. This paper defines the interface to the package.

1. Introduction

This document describes a portable BCPL library which is being made available on several of the machines in the Cambridge University Computer Laboratory. The term 'portable' here refers to the programs that use the library, rather than the library itself, although it is hoped that much of the library code will itself be system independent.

This specification is not intended to be completely free standing: it is expected that readers will have some knowledge of one or more BCPL implementations, and so it aims merely to describe and clarify those areas where the proposed library differs from the "standard" ones available.

The aims of this specification are the following:

1. One single definition of a library that can be used on several machines.

2. A fairly rigorous definition to avoid needless differences in implementations.

3. A reasonably rich set of facilities.

4. The use of exceptions for error handling, but in such a fashion that the default state can be understood without knowledge of this.

5. The result should be implementable on all but the smallest machines in the Laboratory, but it is not necessarily intended to be transportable to all possible BCPL systems on all possible machines.

In an absolute sense, some portability has been sacrificed in the belief that extreme forms of it correspond to bizarre or limiting features which are unlikely to be relevant in our environment. For example, no attempt is made to cope with machines which use ones-complement arithmetic, and conditional compilation facilities are assumed in the compilers. Since the library is rich in facilities, the code to implement it is unlikely to be small, and byte addressed 16-bit machines may experience some problems in supporting it.

The current document describes a first implementation - feedback from potential users and implementors for other machines would be most welcome if it is forthcoming quickly.

December 1981 - Original version.
April 1982 - Revised for release of the first Tripos implementation.
June 1982 - Minor revisions for BCPL Users' Group Conference.
October 1982 - Small editorial changes, AppendOutput, catchall exceptions.

2. Standard Facilities

2.1 Header files and conditional compilation tags

The two and three letter prefixes "LB" and "Lib" have been reserved for the library. As normal, the main library header file is called "LIBHDR"; in addition, there is a set of error manifests in the file "LBERRHDR". Specific implementations may have other subsidiary header files.

Some conditional compilation tags are set by the header and the compiler to characterize the hardware and system software for which a program is being compiled. In our local environment, one (and only one) of the following machine tags will be set:

```
$$PDP11
$$IBM370
$$VAX11
$$LSI4
$$68000
$$CAP1    $$CAP3
```

Also, one (and only one) of the following system tags will be set:

```
$$TRIPOS
$$RSX11M
$$UNIX
$$MVT   $$MVS   $$CMS
```

In addition, there are some derived tags, which are set from the values of the previous two:

```
$$WordAddr      TRUE if McAddrInc = 1 (see below)
$$WordSize16    TRUE if 16-bit machine
$$WordSize32    TRUE if 32-bit machine
$$EBCDIC        Only on IBM machines!
$$ASCII         Most others (but not necessarily all)
$$FloatPKG      TRUE if the floating point package is available
$$RisingStack   TRUE if stack grows upwards, FALSE if downwards
$$BigEndian     TRUE if bits and bytes within a word are numbered
                      from the most significant end towards the least
$$LittleEndian TRUE if bits and bytes within a word are numbered
                      from the least significant end towards the most
```

In addition, LIBHDR always sets the tag $$PortableBlib to TRUE, to characterize the use of this library. Declarations private to the library (such as the more esoteric offsets in stream descriptor blocks) may be accessed by setting the tag $$LibPriv to TRUE.

2.2 Standard manifests

The following manifests are available in LIBHDR:

FirstUserGlobal	- first user global available (also as FUG)
BytesPerWord	- number of bytes in a BCPL word
BitsPerWord	- the number of bits in a BCPL word
BitsPerByte	- the number of bits in a BCPL byte (note that BitsPerByte * BytesPerWord may not equal BitsPerWord)
McAddrInc	- the number of m/c address units to a BCPL word
MaxInt	- the largest representable positive number
MinInt	- the most negative number representable
MaxChar	- the largest value that can be held in a byte
EndStreamCh	- the value returned at end of an input stream
NothingReadCh	- the value returned after UnRdCh is called on a new stream
NIL	- a value which is not (if possible) a valid BCPL or machine address

Both EndStreamCh and NothingReadCh will have values outside the character set (i.e. they will not be in the range 0 - MaxChar). It is expected that NIL will be used as an 'end-of-chain' marker in linked list applications.

There are several other sets of manifests - including those for I/O functions and standard error codes - which are detailed later in this document.

2.3 Initialization and termination

The user's code will be called in the usual way, by invocation of the routine Start. Before this is done, a standard environment will be set up.

A program is finally terminated by a call of the library routine Stop which tidies up and then exits (somehow). If the host understands return codes from programs, the argument to Stop is used as the value to return. FINISH is defined to be a call of Stop (via the global vector) with the value StopNoComment as argument. If Start ever returns, a FINISH is obeyed.

Common return codes have manifests (even on systems where they are not all distinct, they will all be available):

StopNoComment	- nothing about success or failure to report
StopSuccess	- operation succeeded
StopWarning	- operation succeeded, with warnings
StopError	- operation did not succeed
StopFatalError	- catastrophic termination of operation

If the operating system has no concept of 'no comment', then its value for 'success' should be used as a synonym.

2.4 The stack

Each process runs with a BCPL stack for its local variables and procedure invocation details. Normally, there is just one such stack (that belonging to Start), but coroutine (see below) or coordinator systems may have more than one. There is always precisely one active or current stack, which is that belonging to the current process.

A routine may assume that the stack (and hence its arguments) are in contiguous cells, but the direction in which the stack grows may or may not be towards increasing addresses. Conditional compilation on the value of $$RisingStack may be helpful.

Two functions manipulate the BCPL stack:

```
lvl := Level()
       LongJump( lvl, label )
```

The result from Level is only useful with LongJump and the exception system. In particular, it is not a valid way of estimating the remaining stack space. LongJump interacts with the exception system (see below) and the handlers for the UnwindException. It is an error to try to jump out of the current stack.

Note that the function Aptovec is not provided by this library (Getvec is the preferred alternative).

2.5 The heap

The BCPL heap is a dynamic storage area managed separately from the stack. The storage allocated remains available for use until explicitly freed, or the program terminates. The program does NOT have to tidy up after itself (although some may see this as good practice). There is no garbage collection. It is defined that coalition of storage areas returned to the heap occurs, although when and how often it happens is at the discretion of the implementation (it is only actually needed when Maxvec is called or if a Getvec would otherwise fail). The heap may itself not be a single contiguous area. Some simple consistency checks are made in an attempt to trap some common errors such as running off the end of a vector.

```
 v := Getvec( upperBound )
      Freevec( v )
      Shrinkvec( v, newUpperBound )
ub := Sizevec( v )
ub := Maxvec()
 v := GetMaxVec( minUPB, maxUPB, increment )
```

The argument to Getvec must be positive or zero, even on word addressed machines. The contents of the store acquired is undefined. Passing a non-valid area address to Freevec is an error (which may not be trapped by all implementations); this includes calls with argument zero.

Shrinkvec will reduce the size of a previously allocated vector by relinquishing its upper part. It is an error to attempt to increase the size of a vector by this means. Note that there may be a minimum size of shrink needed to cause any releasing of space back to the heap. The

function Sizevec will return the current upper bound for an allocated vector. As with Freevec, it is an error to quote an invalid vector address; this will not always be trapped by all implementations.

Maxvec indicates the upper bound of the largest vector which could be allocated from the program's heap area, except that the calculation will not allow for any expansion of the heap area if this facility is supported by the host. (As a result, it is not as useful as might be supposed - the value returned is not at all the same thing as the largest which could be asked for.) It will return -1 if there is no heap space remaining, which will cause an immediate error if it is passed straight to Getvec.

A 'conditional Getvec' facility is provided by GetMaxVec, which takes as arguments two upper bounds. The first is interpreted as the minimum size acceptable, the second as the maximum being asked for; they must obey the relation

```
0 <= minUPB <= maxUPB
```

The result returned will be the address of the largest vector which could be acquired whose size is between these two limits and a multiple of increment. The upper bound of the vector actually allocated is put into its zeroth word. If the request cannot be satisfied from the heap currently allocated to the program, the heap may be extended; if there is insufficient heap to satisfy even the smaller upper bound, an error occurs. Note that specifying the two upper bounds equal will effectively result in an unconditional request for that amount. Some virtual memory machines may permit very very large values for maxUPB, so care should be exercised!

The library will always use Getvec whenever it needs to acquire storage for internal use, unless it is prevented from doing so by the host operating system. The user may replace Getvec with a private storage management function, if desired, but note that the library assumes that a call to it either succeeds or does not return.

2.6 Character and string manipulation

The string manipulation functions access individual bytes by means of the % operator (and thus use this as their standard of byte ordering). They all expect the length of the string to be recorded in the zeroth byte.

```
nch := CapitalCh( ch )
yes := EqualString( string1, string2 )
dif := CompareString( string1, string2, toUppercase )
out := CopyString( tovec, fromstr )
out := AppendString( ec, addthis )
pos := IndexIn( string, ch )
out := CopyBytes( tovec,tooffset, fromvec,fromoffset, bytes )
out := CopyWords( tovec, fromvec, words )
```

The first will convert the lowercase letters of the alphabet into their uppercase equivalents, passing all other characters through unchanged. (On an EBCDIC machine, the 'others' include the character codes in the gaps in the letter sequences.)

EqualString does not uppercase the strings for comparison, although CompareString will do so if requested. The result from the latter is zero for equality, -ve or +ve according to whether string1 or string2 is the

'lesser' of the two. Comparisons are done in accordance with the collating sequence of the host machine; the shorter string may be considered to be extended to the length of the longer with characters lower than any in the machine's collating sequence.

The effect of CopyString is defined to be that obtained by copying the bytes (in % order) from 0 to the length of the input string. If tovec is zero a suitably sized vector will be acquired by use of Getvec, otherwise it is assumed that the vector pointed to is of sufficient length. The result returned is always the address of the vector used as the destination. (If Getvec is used and fails, an exception will occur.)

AppendString adds the second string to the end of the first, and returns the latter's address as its result.

IndexIn returns the byte offset of the first occurrence of ch in string, or zero if it cannot be found. The length byte is ignored during the search.

The routines CopyBytes and CopyWords will move a sequence of bytes or words from one location to the next. For the first, the source and destination start points are given in terms of a BCPL cell address and a byte offset from it. It is expected that they will be implemented efficiently. Both will do a Getvec if their first argument is zero; in any case, the destination vector is returned as the result.

Note that PackString, UnpackString, GetByte and PutByte are not provided by this library.

2.7 Globals

To ascertain whether a global is undefined or not, or set a global to some 'undefined' value,

```
yes := IsUndefined( globalAddr )
       Undefine   ( globalAddr )
```

may be used. Quoting an address which is not in the global vector is an error. Note that the contents of an undefined global are left entirely up to the implementation.

Programs may assume that the user part of the global vector is contiguous. The manifest FirstUserGlobal (and its shorthand form FUG) defines the global number of the first free slot in the global vector that is available to the user. Note that its value may vary between implementations.

2.8 Time and date

Two functions are available for determining the time of day and the calendar date:

```
v1 := TimeStamp( v1 )
v2 := TimeString( v2, v1 )
```

The former does an atomic acquisition of both the date and time and puts the year, month, day, hour, minute, second, millisecond, and day of the

week into the first 8 words of the result vector. (The manifests TI.year, TI.month, TI.day, TI.hour, TI.minute, TI.sec, TI.msec and TI.wday define the offsets.) The latter takes a vector produced by TimeStamp and converts it into BCPL string form in vector v2.

```
v2+TI.date      - the date, in format "dd-Mmm-yy"
v2+TI.time      - the time, in format "hh:mm:ss"
v2+TI.monthName - the month of the year, in format "Mmmm..."
v2+TI.dayName   - the day of the week, in format "Wwww..."
```

A leading zero will be converted to a space on the day of the month but not on the hour in the day. Monday is the first day of the week, Sunday the seventh.

In both cases, if the output vector argument supplied has the value zero, a suitably sized vector will be acquired with Getvec. If the v1 vector is zero for TimeString, an internal call of TimeStamp will be made. In any case, the result returned is the vector used to store the values or strings. The upper bounds of the vectors to be supplied to the routines are given by the manifests TimeStampUPB and TimeStringUPB.

2.9 The exception system

An exception represents a class of errors that can arise (e.g. input/output errors on a particular stream). Exceptions are represented as addresses of BCPL words that are initialized to zero.

An exception handler is a user function to be called when a particular exception occurs. It may be passed sufficient state to exit to the place at which the handler was set up, or (sometimes) to fix up the error and carry on. Multiple handlers may be declared for an exception. Each handler has an associated level, the first handler used being the one with the 'highest' level (the level moves in the same direction as the BCPL stack, with 'higher' corresponding to more deeply nested).

There is a completely separate set of exception handlers for each stack, so that if an exception is signalled, only the handlers associated with the current stack will be scanned. The coroutine mechanism provides a scheme whereby exceptions can be signalled from one stack to another.

```
id := OnException( exceptionID, level, func, on1,on2 )
```

creates a new handler for an exception, and returns a unique identifier for it. (It is conventional to use the address of a global variable as an exceptionID, but this is not mandatory: the only requirement is that the value of the exceptionID is unique. There is no need for it to be a BCPL cell address, and so the earlier requirement that the cell be initialized to zero is hereby lifted. Nevertheless, the library exceptionIDs are all generated by taking the address of the global used to identify them.) The exception handler (which is a routine with address func) is inserted into the list at a point corresponding to the given level, which should be the result of a call to the function Level. If two handlers for the same exception have the same associated level, the later one added will be the 'higher' of the two. Often, on1 and on2 will be level and label values that can be used as arguments to LongJump to return from the handler to the

scope of the routine which set it up. It is an error to add a handler for an exception at a 'higher' level than that of the caller of OnException. If the value of the exceptionID is NIL, the handler acts as a catchall and will be invoked if the search for a handler on any exception reaches the associated level.

```
OffException( exceptionID, id )
RestoreExceptionLevel( level )
```

may be used to remove exception handlers. The first removes the indicated exception handler for the given exception, or the 'top' one if id=NIL - note that this may not be the most recently added one. It is an error if the handler cannot be removed for some reason (e.g. if the exception has no handlers, or id is invalid). The second removes all handlers with the given or 'greater' level. Neither may be used to remove an active exception handler.

```
res := Signal( exceptionID, how, errcode, a,b,c,d,e,f,g )
```

is the call which is used to signal (or raise) an exception. In its turn it calls the top handler of the given exception:

```
res := func( exceptionID, how, on1,on2, errcode, a,b,c,d,e,f,g )
```

Note that the third and fourth parameters are from the OnException call which set up the handler. The parameter how should be one of

EX.Fatal	- it is quite inconceivable that the user program continue after this error. If a handler tries to LongJump out, Stop will be called.
EX.DontReturn	- the call to Signal should not return (but may exit via a LongJump).
EX.Notify	- the exception is only raised to notify an event. If there is no handler for a notification then the call of Signal simply returns - there is no recourse to the default handler (see below). The usual handler exits are available; the normal course is to return. Note that the issuer of the Signal cannot not expect a result.
EX.NonFatal	- otherwise - a result may be expected.

The exception handler may exit in a number of different ways:

1. It may return a result which will be passed back to the caller unless how was EX.DontReturn or EX.Fatal, in which case Stop(StopError) or Stop(StopFatalError) is obeyed.

2. It may issue a LongJump to another point on its stack, except that if how was EX.Fatal, the call will be converted into one of Stop(StopFatalError).

3. It may call Stop directly.

4. It may issue a call to the routine ResignalException(), which indicates that this handler is unable to cope and the exception should be re-signalled as if this handler was not present (the call will never return). This has the advantage of consuming less stack space than simply re-signalling the same exception since the current handler's stack frame is removed before the next handler is called.

If anybody (including the handler or the exception system) explicitly or implicitly calls Stop or LongJump, then all those handlers that are between the current level and the target level will be removed. For LongJump the target level is the value of the first argument; for Stop it is a value 'below' the level of Start. Handlers at the destination level remain. If there are any are handlers for the UnwindException (see below) they will be invoked before they are removed, with how set to EX.Notify, errcode set to ERR.UnwindStop or ERR.UnwindLongJump as appropriate, and the argument a to the target level. Explicit signalling of the UnwindException is allowed, but the how parameter is always treated as EX.Notify.

A handler or a routine it calls may signal the same exception again, in which case the handler to be invoked will be the next one down the list. This can be prevented by having the handler reinstate itself by a call to

```
ok := ReinstateExceptionHandler()
```

Indefinite recursion should be avoided! This call is ignored in a call from an active handler for the UnwindException.

Two exceptions are predeclared by the exception system (as opposed to the rest of the library); they are global variables whose address space should be used as an ExceptionID:

- The UnwindException is used when the stack is being lifted by Stop or LongJump as described above. It is initialized to zero.
- The DefaultException is used when a handler is needed for some other exception and there is none. If there is no DefaultException handler, then Stop(StopFatalError) or Stop(StopError) is called. This exception is initialized with the standard library error handler.

2.10 Coroutines

A coroutine is a process with a distinct BCPL stack which shares the code, global vector and heap space for the program as a whole. Transfer of control from one coroutine to another is always explicit, rather than dependent upon external factors such as a time-slicing coordinator. A coroutine may be ACTIVE, in which case the program is executing code in its stack; or INACTIVE, when it is waiting to be re-activated. Each ACTIVE coroutine has a parent; some INACTIVE coroutines may also have parents - this will be the case if they have invoked another coroutine as an offspring, and it has not yet returned.

A new coroutine can only be created or destroyed by calls to the functions

```
coptr := CoCreate( routine, stackSize )
         CoDelete( coptr )
```

The first makes a new stack for the coroutine and returns a coroutine pointer, which is a non-zero number. (Such coroutine pointers are only useful in the context of the functions discussed here.) Any of the standard errors (such as insufficient heap) may occur, and will be reported in the usual way. The new coroutine immediately executes a CoWait with its coroutine pointer as argument, thus returning this as the result from the CoCreate call. When control is next passed to this routine, by a CoCall or CoResume, execution will proceed by calling the routine with the value passed in this call as its argument.

The second releases the stack space for a coroutine. It is an error to quote the coroutine pointer of a coroutine which has a parent.

To invoke an already-created coroutine, the function

```
res := CoCall( coptr, arg )
```

should be used. It passes control to the routine previously defined in a CoCreate call, giving it arg as its single argument or as the result from the call it made to CoWait or CoResume to suspend itself. The parent of the target coroutine is marked as being the one that issued the CoCall. It is an error for a coroutine to try to pass control to one which already has a parent. Normally, the called coroutine should suspend itself at some stage by a call to

```
newarg := CoWait( res )
```

which will pass res back as the result of the CoCall. The coroutine will next be activated by a further call of CoCall, and the arg value to that call will appear as the result of the CoWait.

To pass control to a waiting coroutine and mark it as having a parent which is the parent of the current one, the function

```
newarg := CoResume( coptr, arg )
```

may be used. It appears like a combination of calls to CoWait and CoCall, and the same restrictions about not re-activating a coroutine with a parent apply.

If a coroutine body returns from its initial call, its result will be used as the argument to CoWait, and the result that returns used to re-invoke the coroutine body.

The coroutine system interacts with the exception system: there is a separate list of exception handlers associated with each coroutine. To signal an exception across a coroutine boundary, the function

```
res := CoSignal( coptr, exceptionID, how, errcode, a,...g )
```

should be used. It behaves exactly like the normal Signal call, except that the call takes place on the stack of the given coroutine. If the handler ever returns, control is passed back to the calling coroutine; otherwise it behaves like a call to CoCall, except that the signalled coroutine may have a parent.

It is an error to LongJump out of the current coroutine.

2.11 Miscellaneous

MulDiv(a, b, c) is defined to calculate the result of multiplying a by b and dividing the result by c. It will always do so in such a fashion as to preserve the intermediate double-length product to full accuracy. The remainder from the division (which will be of the same sign as a*b) is left in the global LibResult2. [Note that there is no global variable named Result2 provided by this library.]

The two functions Max(a,b) and Min(a,b) calculate the (signed) maximum and minimum of two values.

3. Input/Output

The basis of the I/O system is the function which defines streams. Streams may be duplex (mode = IO.InOut), or simplex (Mode = IO.In, IO.Out or IO.Append). Several built-in stream definitions (such as for external files and the like) are provided. The lowest level stream opening function is:

```
stream := FindStream( mode, func, workSize, a,b,c,d,e,f,g )
```

It returns a stream pointer for an open stream of the given mode or causes an error if it cannot be opened. A valid stream identifier is a non-zero number - it is the address of a Stream Descriptor Block (SDB), which is the internal data structure describing the state of a stream.

The argument func is the address of a 'stream function', to be invoked by the library to perform operations such as opening, closing and carrying out transfers on the stream; workSize is the number of words in a work vector to be made available (NOT its upper bound); and a to f are arguments to be copied into the first 7 (or less if workSize is too small) words of it.

When invoked, func will be passed an SDB address as its first argument and a manifest of the form IO.<function> (e.g. IO.Rewind) as the second indicating the action to be taken; subsequent extra arguments may also be provided. All library-defined control manifests will be strictly positive, so that negative values are available to the user to implement extended control operations.

If the stream function wishes, the library will manipulate an internal buffer, issuing calls to the stream function only when the request cannot be satisfied with the buffer. The initial state will cause the stream function to be called on every transfer. By having the library handle buffers in this fashion, it is hoped that much code can be shared between

stream implementations and some increase in runtime efficiency obtained, particularly in the commonest cases (transferring data to and from external buffered media).

The stream function should signal the exception associated with the stream when the stream is exhausted or a transfer error (such as device full or parity error) occurs. Obeying FINISH with open streams will cause them to be closed automatically, in the reverse order to that in which they were opened. More information about the precise interface to stream functions and the contents of SDBs is provided later in this document.

3.1 Library-provided streams

The following high level functions are available; they may be considered to map onto calls of FindStream with suitable arguments:

```
stream1 := FindInput ( fileDefinition1 )
stream2 := FindOutput( fileDefinition2 )
stream3 := FindAppend( fileDefinition3 )

stream4 := FindString( mode, string, maxlen )
```

It is defined that fileDefinitions are strings. An attempt to open a given file more than once for (simultaneous) input is allowed, and will cause completely distinct streams to be established; opening the same file for output whilst it is already open will either succeed to produce distinct output files, or cause an error. FindAppend is equivalent to FindOutput, except that the file must exist beforehand (if the host allows this to be checked), and the next write to it will be appended to the data already there. If possible, an error will occur otherwise.

FindString takes as its first argument one of the manifests IO.In, IO.Out or IO.Append, and returns a simplex stream to do the relevant thing to the vector/string provided. If mode is IO.Out or IO.Append, the maxlen parameter must be supplied which specifies the highest byte offset that can be written to. An error occurs if an attempt is made to make the string longer than this. On output, the length byte will not necessarily be maintained as a true reflection of the current string length while the stream is open.

Three streams are defined on entry to Start: SysIn and SysOut are set to the values of the currently selected streams, or zero - typically, they will refer to the standard streams specific to the host system; SysErr is set to a stream, on which every effort is made to see that characters sent to it are made visible. It is the preferred stream for reporting errors on; note that it may have the same value as SysOut on some systems.

3.2 Controlling streams

Streams may be selected and the current stream pointers enquired after:

```
            SelectOutput( stream )
            SelectInput ( stream )
stream :=   Input()
stream :=   Output()
   yes :=   IsValidStream( stream )
```

The effect of passing junk to SelectXXput is to cause an error. Selecting zero sets the state of unselection, in which it is an error to attempt to transfer characters or obey control functions. Input and Output will return zero if there is no selected stream. The last function returns TRUE if its argument looks like the address of an SDB, or FALSE otherwise. No error occurs in the latter case.

Once a stream has been finished with, it may (should) be closed, a process which normally releases back to the program any buffer and control structure space acquired from the heap or operating system. Some streams may be rewound (repositioned to their beginning), or closed for output and re-opened immediately for input. In either case, the previous stream pointer is returned as the result of the appropriate call.

```
            EndRead()
            EndWrite()
            EndStream( stream )
stream :=   EndToInput()
stream :=   Rewind()
```

If the stream to which these functions are applied is selected, the state of unselection will be set for input or output (or both) as appropriate. EndToInput will cause unselection of the current output stream, but neither it nor Rewind will affect the selection of the current input stream. If a stream cannot be closed or re-opened an error occurs. EndStream applied to a duplex stream will close both halves; EndRead or EndWrite will normally only affect the indicated half. EndStream may be applied to a simplex stream.

Calling a control function will lead to an invocation of

```
res := IOcontrol( stream, IO.<function>, a,b,c,d,e,f,g  )
```

which will itself map onto a call of the appropriate stream function. Res is only defined if the appropriate function is expected to return a result. Users may invoke IOcontrol directly if they wish, but should note that this mechanism bypasses any internal buffering that the library may be doing.

Two functions to indicate the 'real-time' response of streams are provided:

```
chars :=  TestInCount()
chars :=  TestOutCount()
```

The first returns a lower bound to the number of characters which the current input stream has buffered up, ready to be read, or zero if there are none; the second indicates a lower bound to the number of characters that could be immediately swallowed by the current output stream without hanging.

3.3 Character mode

Streams can be read from or written to in one of two modes: character mode or binary mode. Character mode is indicated by use of RdCh and WrCh, binary mode by BinRdCh and BinWrCh.

Whilst writing in character mode, certain bytes will be interpreted by the library and translated into special actions; they may also be generated by a set of convenience routines which map onto calls of WrCh:

routine	char	action
SameLine	*C	return to beginning of a line
NewLine	*N	start a new line
NewPage	*P	start a new page

Note that the effect of calling the functions is defined in terms of writing the control characters and not the other way around. The external representation of these calls is host system dependent. To send the bytes used to represent *C, *N and *P, BinWrCh should be used.

The only other control characters which are allowed as arguments to WrCh are *B (backspace), *E (escape), *L (linefeed, but only if distinct from *N) and *T (tab), which are passed through as data characters. It is an error to pass it any other control character value, or a value which does not fit into a single byte; a user-supplied error handler may choose to return a value to be written out - it will not be checked further.

Any internally buffered characters may be flushed out by means of the function

```
ForceOut()
```

which will not result in any data characters being sent down the stream. Some streams may have and use mechanisms for passing control information, in which case a ForceIn exception (see Chapter 5) will be raised when this point in the stream is encountered whilst it is being read. Calling ForceOut several times in succession may be equivalent to a single call on some systems.

On input, calling RdCh will yield successive characters until the stream is closed, exhausted or some error occurs. Note that it may be possible to encounter an end of stream that is not immediately preceded by a line terminator. It is an error to read a character in the range 0 - MaxChar with RdCh which could not have been written with WrCh (this applies particularly to control characters); a user supplied handler for this error may choose to return a character to be passed back in place of the offending one.

```
UnRdCh()
```

will backspace the current input stream by one character. It is an error to call it before a character has been read from a stream (which will cause RdCh/BinRdCh to return the value NothingReadCh), or more than once between reading two characters. A 'last character' value is preserved with each stream, and will be remembered over stream selection.

A program can enquire whether a particular stream is (may be) connected to an 'interactive' device such as a terminal by the call

```
yes := IsInteractive( stream )
```

which maps (via IOcontrol) onto a call of the relevant stream function. If this information is not available, the result should always be TRUE. Writing *C, *N or *P in character mode to a stream labelled as interactive will also cause any internal buffers to be written out. An interactive input stream is not required to support any other line terminator than *N, although some implementations may do so.

```
count := MaxCharsOnLine()
```

returns the number of characters which may be written in character mode to the current output stream before a wrapoutput error occurs. The value returned will be MaxInt if the error will never occur; a NoInfo error will be caused if the information is not available.

3.4 Binary mode

In binary mode no translations whatsoever are done by the library to the characters being transferred. Operating system dependent line or record breaks may be caused implicitly by buffer overflows or the like on output; such breaks are completely ignored on input. BinWrCh will discard all but the bottom byte of any value passed to it.

The effect of mixing BinRdCh and RdCh on the same stream is well defined provided that no mixing of modes by use of UnRdCh is made - i.e. the effect of

```
RdCh() ... UnRdCh() ... BinRdCh()
```

is undefined. Indeed, the reader of a stream should use the same sequence of RdCh/BinRdCh calls as the writer made of WrCh/BinWrCh when the stream was output - the effect of not doing so is also undefined.

3.5 Binary mode bulk transfers

In binary mode there is a bulk transfer mode known as vector mode which emulates the result of repeated calls to BinRdCh (except that it may be more efficient). The transfer functions are:

```
len := BinReadVector ( vector, maxlen )
       BinWriteVector( vector, len )
```

The first will read maxlen characters into vector as if by repeated use of BinRdCh. Len will equal maxlen unless an error (e.g. end of stream) took place during the operation. The second will output len characters, again as if using repeated calls to BinWrCh.

A second type of binary bulk transfer, (record mode) makes direct use of underlying operating system primitives, and is thus not portable: it is described below in the section "Extensions".

3.6 Character mode input functions

The character mode input functions are constructed from calls to RdCh. All leave the terminating character to be read by the next call (i.e. an implicit UnRdCh will have taken place).

The basic function for reading numbers is:

```
num := ReadDigits( base, width, sign )
```

It takes as arguments the radix of the number to be read (which must be between 2 and 37 inclusive), the maximum number of characters to consume whilst reading the number (which must be strictly positive), and one of +1 (positive), 0 (unsigned) or -1 (negative) to indicate the sign of the number to be read.

Characters are mapped onto digits in the order 0-9, A-Z; lower case being treated as equivalent to upper case. Numbers are defined as contiguous sequences of digits (no leading signs or layout characters (*S, *T, *C, *N, *P) are allowed by this routine).

If no digits are encountered an error occurs; likewise when trying to read a number that would overflow the wordsize of the machine. All values which can be represented in a single word can be read (in particular, MinInt may be read successfully).

For compatibility with existing libraries there is a read-integer function which skips all layout characters (*S, *T, *C, *N, *P) whilst looking for a number:

```
num := ReadN()
```

Since this uses ReadDigits, the same conventions about errors and formats of numbers hold good, except that it understands immediately preceding signs ('+' or '-' characters). Numbers without a leading sign are treated as positive. Another variant skips leading spaces and tabs (*S, *T) and then reads an unsigned number in a given base:

```
num := ReadNumber( base )
```

Two functions are used by the library for converting characters into numbers and vice versa:

```
num  := CharToNum( char, base )
char := NumToChar( num,  base )
```

In both, base must be between 2 and 37 inclusive. The character argument may be in either case - upper and lower case are treated as equivalent; the number to character mapping is the same as for ReadDigits. An error occurs if the character or number is not representable.

A function is available to read in a string terminated by any of a set of supplied characters:

```
len := ReadChars( vector, maxlen, terminators )
```

Up to maxlen (which must be <= MaxChar) characters are read into a string constructed in vector, stopping when one of the characters in the string terminators is read, end-of-stream encountered, or an error occurs (such as maxlen+1 characters having been read but no terminator encountered). The actual number of characters in the string is returned as the result and placed into vector%0.

For skipping over characters to be ignored (e.g. leading layout characters) or for skipping forward until one of a set of characters is met, two functions are provided:

```
count := SkipWhile( discardChars )
count := SkipUntil( terminatorChars )
```

The first swallows and discards all characters until one which is not in the string discardChars is met; the second stops only when one of the characters in the string terminatorChars is read. Both also stop when a value outside the character set (e.g. EndStreamCh) is encountered or an error occurs on the stream. They return the number of characters read (not including the one on which they stopped).

3.7 Character mode output functions

By analogy with the input functions there is a standard set of character mode output functions, which all use WrCh as their basic primitive. The two lowest-level functions for writing out numbers are:

```
count := WriteDigits   ( num, width, base, signed )
count := WriteLowDigits( num, width, base )
```

The former writes out num in the given (or greater, if need be) width to the given base. If signed is FALSE, a leading '-' sign will be provided if num is negative; if TRUE, a leading '+' or '-' sign will always be supplied. The latter outputs the bottom width digits of the number, extending it to the left with leading zeroes if needed, in an unsigned form in the given base (which must be a power of two).

The most general output function is WriteF, which takes a format string and a number of arguments and interpretes the latter in accordance with escape sequences in the former. The escape sequences are used to cause invocation of other output functions:

```
count := WriteF( formatString, a1, ... a12 )
                                  // WriteF escape
count := WriteI( num, width )     // "%In" - decimal in fixed width
count := WriteN( num )            // "%N"  - decimal in min width
count := WriteO( num, width )     // "%On" - octal (only low digits)
count := WriteP( num, width )     // "%Pn" - octal in fixed width
```

```
count := WriteR( num, radix )    // "%Rn" - based number, min width
count := WriteS( str )           // "%S"  - string in minimum width
count := WriteT( str, width )    // "%Tn" - string in fixed width
count := WriteU( num, width )    // "%Un" - unsigned decimal
count := WriteX( num, width )    // "%Xn" - hex (only low digits)
count := WriteY( num, width )    // "%Yn" - hex in fixed width
```

All the functions return a count of the number of characters they output. WriteF takes up to 12 arguments in addition to the format string. Note that the escape character used and the last character of the function name are always the same; as a result, there is no WriteD function provided.

If a number or string cannot be held in the indicated field width, it will be output correctly in the minimum width necessary, with the exception of WriteO and WriteX, for which only the bottom digits of the arguments are output, complete with leading zeroes. Hexadecimal and octal numbers are written in unsigned form.

The following two definitions hold good:

```
LET WriteN( arg ) = WriteI( arg, 0 )
AND WriteS( arg ) = WriteT( arg, 0 )
```

WriteI, WriteP, WriteT, WriteU and WriteY will justify their output within the given field width - to the left if the width is negative, to the right otherwise.

WriteF escapes may be in either case; numeric values following the escapes are a single digit interpreted in base 37, which may be immediately preceded by a sign ('+' or '-'). Additional escapes are:

%B	Uses BinWrCh, by analogy with %C
%C	Uses WrCh
%Fn	Treats the next n+1 arguments as a recursive WriteF call
%Zn	Treats the next argument as a routine call with the following n arguments as parameters
%$	Skips the next argument
%%	Outputs %

Unrecognized escapes cause an error. The function invoked by %Z should return the number of characters it caused to be written out.

3.8 A counting filter

For applications which wish to monitor the number of characters written to a line, or the number of lines written to a page, a 'filter' stream function is provided:

```
newstream := FindCounts( mode, oldstream, expandtabs, noPages )
```

This counts the number of characters read from or written to the stream since the last *N, *C or *P, together with the number of *Ns read or written since the last *P (or the beginning of the file). Mode should be one of IO.In or IO.Out; oldstream should be an already-open stream of the same type; if the expandtabs argument is TRUE, a *T will cause the chars-on-line counter to be incremented to the next tab stop (tab stops are fixed at every eight columns, starting with column 1), otherwise, tabs are

treated like ordinary characters. If noPages is TRUE, *P characters do not reset the lines-on-page count (this is useful for counting lines-in-file). These counts can be interrogated by means of the call

```
vector := ReadCounts( newstream, vector )
```

where vector is a vector with upper bound ReadCountsUPB, which will have

```
the associated stream pointer in   vector!FC.Stream,
the characters-on-line count in    vector!FC.CharOnLine,
the lines-on-page count in         vector!FC.LineOnPage and
the pages-in-file count in         vector!FC.PageInFile
```

placed into it at the given offsets; its address will be returned as the result. This filter only works in character mode: it is an error to use BinRdCh/BinWrCh on it. Unset items will have the value zero (e.g. just after opening the stream or doing a rewind).

4. Error Handling

The library makes use of the exception system described above for handling errors. All errors will cause the signalling of an exception and subsequent invocation of an error handler. There are two library error classes: stream-specific and general. The former are signalled on an exception associated with each stream (stored in the SDB); the latter on the general library exception (LibException). Two standard handlers are provided; both classes of exceptions are initialized (the former at FindStream time, the latter before invocation of Start) to have one of the library handlers.

4.1 The default handlers

The stream specific handler interprets and understands errors which arise as the result of operations (such as reading or closing) which are specific to one particular stream. Certain errors are trapped and particular action is taken:

- The end-of-stream exception from calls of RdCh or BinRdCh results in a negative value (EndStreamCh) being returned.
- The wraparound error is ignored.
- The exception (not strictly an error) signalled as when a ForceOut is read causes the result of a RdCh or BinRdCh call to be returned as the result of the original call (i.e. the ForceOut is effectively ignored).

Any other errors are passed back to the general error handler by explicitly signalling the error on the general exception.

The general error handler copes with those errors that are not related to particular streams and those errors which the stream handler chooses to pass on to it. In general, its behaviour is to issue an error message and then call Abort (or Stop, if Abort is not defined) to exit from the program, except that an invalid floating point length is ignored. If Abort should return, Stop will be called.

The effect of the standard error handlers is to trap those operations that most programs will be unable to cope with (e.g. I/O errors, coding errors) or traditionally never bother to find out about (e.g. illegal input syntax). The result is that the exception system underlying the library is completely transparent to the simple user - if a call returns, it can be deemed to have succeeded - but the library permits more subtle error handling by those who understand its workings. It also means that errors get reported as soon as possible. For example, if a FindInput fails the user will be notified immediately and not have to wait until an attempt is made to write something to the resulting invalid stream.

Any of the standard handlers may be removed if desired, in which case the standard exception package processing will take place. More normally, a program may add handlers onto the exception to be invoked before the library ones are reached. In this case, the action for error codes which the new handlers do not recognize should be to call ResignalException.

When a library exception handler is invoked, the arguments specific to the call (i.e. not the first three) will be a manifest indicating the reason for the call, and (possibly) some extra information.
There is a function to convert library error codes into WriteF style strings:

```
WriteF( LibErrorString(code), code, arg1, arg2, ... )
```

where the argN parameters are the call-specific parameters - usually the parameters passed to the routine which detected the error.

4.2 Manifests for library errors

The following manifests are used to represent errors issued by the library. They are in two groups: those which are initially raised on the exception associated with each stream, and those which are invoked on the shared library exception. All have names commencing with ERR. First, the stream signals:

ERR.Success	Not an error!
ERR.Maxlen	Trying to put too many chars into a string.
ERR.EndToInput	Unable to re-open or close.
ERR.Rewind	Unable to rewind.
ERR.EndRead	Unable to close.
ERR.EndWrite	Unable to close.
ERR.EndStream	Unable to close.
ERR.IOfunction	Unknown IO.<function> in a stream function.

ERR.EndOfStream	End of input file read (not strictly an error)
ERR.ReadError	Read error on i/p from a stream function.
ERR.WriteError	Write error on o/p from stream function.
ERR.WrCh	Invalid character to be written.
ERR.RdCh	Invalid character read.
ERR.UnRdCh	Too many calls.
ERR.ReadDigitsSize	Number too large to represent.
ERR.NoDigitsRead	Number not read before non-digit encountered.
ERR.ReadChars	Max string length > MaxChar.
ERR.WrapOutput	A wraparound is about to occur on an operating system record in character mode.
ERR.ForceIn	Not an error - a ForceOut has been read.
ERR.NoInfo	Information requested not available.
ERR.Point	Invalid notevec.

Next, the manifests used for exceptions and errors signalled on the library exception. The ones marked with an asterisk are in the class of hardware detected errors, about which a little more is said below.

ERR.Success	Not an error!
ERR.NoInfo	Information requested not available.
ERR.NoHeap	Insufficient heap space.
ERR.CorruptHeap	Somebody has trampled on the heap.
ERR.Getvec	Invalid size to Getvec.
ERR.GetMaxvec	Invalid arguments to GetMaxVec.
ERR.Freevec	Invalid pointer to Freevec.
ERR.Shrinkvec	Trying to increase size of a vector.
ERR.NotGlobal	Address not in global vector to IsUndefined or Undefine.
ERR.UndefinedGlobal	Calling a global which has not been defined.
ERR.CoPtrInvalid	Junk coroutine pointer.
ERR.CoHasParent	Trying to activate a coroutine which already has a parent.
ERR.LongJump	Trying to jump out of the current stack.
ERR.OnException	Invalid arguments to OnException.
ERR.OffException	No handler to remove.
ERR.ReinstateExceptionHandler	Calling from outside a handler.
ERR.ResignalException	Calling from outside a handler.
ERR.UnwindStop	Not an error - call to an UnwindException handler.
ERR.UnwindLongJump	Not an error - call to an UnwindException handler.
ERR.FindStream	Invalid parameters to FindStream.
ERR.FindInput	Failure to open a host system file for input.
ERR.FindOutput	Failure to open a host system file for output.
ERR.FindAppend	Failure to open a host system file for appending.
ERR.FindInOut	Failure to open a duplex stream to a host system file.
ERR.Stream	Invalid stream pointer.
ERR.NoInStream	The current input stream is unselected.
ERR.NoOutStream	The current output stream is unselected.
ERR.NumToChar	Invalid number to be converted.
ERR.ReadDigits	Invalid argument(s).
ERR.WriteF	Invalid escape combination in format string.
ERR.TimeString	Invalid timestamp presented.
ERR.FindCounts	Invalid arguments to FindCounts.
ERR.ReadCounts	Calling ReadCounts on a non-FindCounts stream.

```
ERR.WriteLowDigits Invalid base value.
ERR.ZeroDivide*    Fixed point division by zero.
ERR.AddressError*  An address violation has been detected.
ERR.FloatSqrt      Square root of a negative argument.
ERR.FloatLength    Invalid length.
ERR.FloatZeroDivide* Division by zero.
ERR.FloatOverflow*   Overflow (N.B. underflow will be ignored).
```

In addition, all the exceptions defined for the stream error handler may be passed back to the default error handler. All library error codes are non-negative.

4.3 Hardware detected errors

Most BCPL implementations are extremely tolerant about the handling of hardware detected errors such as arithmetic overflow. Some errors, however, such as division by zero, should usually be (and indeed normally are) seen as programming errors, and cause some form of standard system action of greater or lesser degrees of unpleasantness. It is intended that implementations of this library will go to considerable lengths, if need be, to allow such hardware detected errors to be passed back through the standard exception system. The initially defined set of such errors consists of the ones marked with an asterisk in the list above.

5. Internal details

This chapter contains information that should only be of interest to those wishing to make use of the library to write their own streams, or who are interested in a more precise definition of its behaviour. It also contains information on some functions which are not normally considered to lie within the 'user domain', but which nevertheless may occasionally be found useful (although their exploitation is to be discouraged, in general).

5.1 Program information

Some functions are defined which return word addresses giving information about the extent of the BCPL stack and the global vector. If the information is not available (e.g. the global vector is fragmented), an error occurs.

```
StackBase()     StackTop()
GlobalBase()    GlobalTop()
MaxGlobal()
```

The value returned by StackTop may vary from one call to the next in some systems (e.g. if the heap and stack compete for space, or if a new coroutine or process has been entered). It is defined that the address of a local variable may be used with the result from StackTop (if there is one) to calculate the amount of stack space remaining, provided note is taken of which way the stack grows. Note that the following relation may NOT hold good:

```
GlobalTop()  =  GlobalBase() + MaxGlobal()
```

To alleviate overflow problems on word addressed machines, a function is available to compare the values of two unsigned numbers, typically addresses:

```
cmp := UCompare( addr1, addr2 )
```

The result is -ve, 0 or +ve, depending upon whether addr1 is less than, the same as, or greater than addr2. In addition, LibResult2 will be set to the (unsigned) difference between the two values.

5.2 Debugging

The following routines are available to help with debugging (although their actual inclusion in a working program may be optional). It is expected that systems with interactive debugging facilities will in general put more emphasis on those than the routines provided here.

```
Abort( code )
MapCode()
MapGlobals()
MapStack()
```

Abort is the final recourse of the error handler to the system-dependent part of the library; it is called after any error messages have been output. If it is undefined, then Stop(StopFatalError) is invoked instead. The other routines all output information to the currently selected stream, the last causing a backtrace to the 'bottom' of the active stack from the current level.

5.3 Stream Descriptor Block format

The communication between the library and a user-supplied I/O function is largely by means of a Stream Descriptor Block (SDB). This contains the following information (there may be more fields private to the library itself), defined in terms of manifests which refer to offsets within it:

SDB.Mode	type of stream (IO.In, IO.Out, IO.InOut)
SDB.Func	control function for the stream
SDB.Exception	exception for stream-specific errors
SDB.LastCh	the last value returned by RdCh/BinRdCh
SDB.UnRdChFlag	true if UnRdCh call since RdCh/BinRdCh
SDB.ForceFlag	true if there is a pending ForceIn
SDB.InBuffer	buffer for the input half
SDB.InBuffUpb	extent of the input buffer
SDB.InBuffPos	last position in input buffer read from
SDB.InBuffHWM	last filled offset in the input buffer
SDB.OutBuffer	buffer for the output half
SDB.OutBuffUpb	extent of the output buffer
SDB.OutBuffPos	last offset written to in the output buffer
SDB.IsInteractive	TRUE if so
SDB.LineMode	TRUE if *N/*C/*P to go direct to stream function
SDB.UserDataSize	words (upb+1) of user data available
SDB.UserData	first word of user data area

The stream function is invoked by IOcontrol in the form:

```
res := func( sdb, IO.<function>, arg1, arg2, ... )
```

The set of functions defined by the library is given below.

IO.In	/ one of these four
IO.InOut	¦ will be called
IO.Out	¦ to initialize
IO.Append	\ the stream
IO.EndRead	/ one of these
IO.EndWrite	¦ three will be called
IO.EndStream	\ on termination
IO.Rewind	reset an input stream
IO.EndToInput	convert output to input
IO.RdCh	buffer empty (character mode) & to read
IO.WrCh	character mode write; interpret char given
IO.BinRdCh	buffer empty (binary mode) & to read
IO.BinWrCh	binary mode write
IO.ForceOut	flush buffers & propagate if possible
IO.TestInCount	return number that can be written NOW
IO.TestOutCount	return number in hand NOW
IO.SameLine	writing *C in character mode
IO.NewLine	writing *N in character mode
IO.NewPage	writing *P in character mode
IO.MaxCharsOnLine	return chars-to-WrapError, or MaxInt
IO.Note	remember a position
IO.Point	and restore one

5.4 Opening and closing streams

The user-supplied arguments (a-g) to FindStream will not appear as arguments on the initial call, but will be placed in the user data area of the SDB.

Stream functions are responsible for allocating and releasing any buffer space; to make use of the library buffer handling, the input and output pointers should be set correctly to reflect the amount of space remaining or number of characters left. The library will take care of allocating and releasing the SDB itself.

The words at offset SDB.IsInteractive and SDB.LineMode are initialized to FALSE before the stream function is called for the first time. They should be changed if desired.

Simplex streams will always be called with IO.EndRead or IO.EndWrite, even if EndStream is called to close them.

If the stream is opened with mode = IO.Append, this will be passed to the stream function at open time, but the value stored in the SDB will be IO.Out.

5.5 Reading from streams

When a character is read from the buffer and the buffer is found to be empty, the library will invoke the stream function with either IO.RdCh or IO.BinRdCh as the function, depending upon the way in which it was itself called. The stream function should return the next value to be passed back to the user (there is no restriction that it must fit into a byte). It may also put some characters into the buffer pointed to by the SDB (or change the buffer pointer suitably), in which case any relevant other fields should be updated (e.g. the last-valid position). The number of characters transferred should be the minimum of the numbers required to:

- fill the buffer,
- encounter the external representation of a ForceOut,
- reach the end of an operating system record (for a record-based system), or
- reach the end of a line written in character mode (for a stream-based system).

If a ForceOut is to terminate the buffer, the flag in SDB.ForceFlag should be set TRUE. When the library tries to read the character after the buffer has been emptied, it will look to see if the forceout flag is set and, if so, signal the ERR.ForceIn exception with either RdCh or BinRdCh as an additional argument. The exception handler should normally return the character to be passed back to the user as the result of the call which detected the condition. (The standard I/O handler simply invokes the function it is given after Reinstating itself.) Note that the exception system will normally bypass the currently active handler when an exception is re-signalled (directly or indirectly) whilst it is executing. If an exception handler for the ERR.ForceIn signal wishes to do a RdCh on the same stream - which may in turn cause the exception to be signalled again -

it should make a call to the exception system to re-instate itself. Take care to avoid infinite recursion!

The library buffer handling software will assume that the buffer address in the SDB is valid if the position and high-water-mark pointers indicate that there is room in it for the current character, otherwise it will not be looked at. The values at offsets IO.InBuffPos, IO.InBuffHWM and are initialized to -1 when the stream is opened, rewound or has EndToInput called on it; this is done before the stream function is invoked.

5.6 Writing to streams

On output, the library will invoke the stream function (with one of IO.WrCh or IO.BinWrCh) if the buffer is full, passing the character 'in its hand' as an extra argument. The stream function should empty the buffer, reset the pointers and accept the given character (one way would be to put it into the buffer).

If (and only if) the word at offset SDB.LineMode is TRUE, the stream function may also be invoked with one of the modes IO.SameLine, IO.NewLine or IO.NewPage; this may occur even when the buffer is not full. These represent the corresponding calls to WrCh with the associated characters (the character is passed as the next argument). This can be used to cater for those systems on which special action is needed to generate their external representation (an end-of-record, for example).

With all other stream operations (including ForceOuts, EndStream, etc), the buffer will NOT be flushed by the library beforehand: this is the responsibility of the stream function.

For implementations which make use of operating system records, the buffer used should normally be of the same length as one of those. If a character mode write is attempted and the record is full, the stream function should signal the ERR.WrapOutput exception on the stream, unless implicit buffer handling takes place to relieve the situation (e.g. on an IO.NewLine).

Some systems (especially record-oriented ones) may not have any suitable external representation of a ForceOut, and hence can never generate an ERR.ForceIn exception on input. They should nevertheless cause any buffers to be flushed on output if the stream is (may be) connected to an interactive device such as a terminal.

5.7 Miscellaneous

The IO.TestInCount function should return an indication of how many characters the stream function has in its hand, while the IO.TestOutCount function should return the number of characters that the stream function can immediately consume. They will only be called when the required information cannot be deduced from the current state of the buffer (i.e. it is full when writing or empty when reading).

Buffers and their sizes may be altered at any time by the stream function, provided that any other affected values in the SDB are updated.

The stream function may in its turn cause other calls of the I/O system, although recursion on itself should be treated with care.

6. Extensions

There are a number of areas which are outside the scope of the basic library, but which are nevertheless worth attempting to specify here to encourage consistency, should they be implemented.

6.1 Floating point

If a floating point package is provided (e.g. if the host compiler or machine does not support the language extensions for floating point), it should take the form indicated below. It should be noted that, for portability reasons, it is highly desirable for all BCPL implementations to provide this package, whether or not they have the floating point language extensions available. It is not necessary for it to be included by default in programs. Its availability on the system will be indicated by the setting of the conditional compilation tag $$FloatPKG.

Since some machines permit several different lengths of floating point numbers, the function

```
Flength( length )
```

is defined to allow the 'current length' to be altered. It takes as argument one of the manifests for the upper bounds of vectors used to hold floating point numbers, which have the following names:

```
FloatUPB
FloatSingleUPB
FloatDoubleUPB
FloatQuadUPB
```

The first is the default; it will have the same value as (at least) one of the others. Specifying an invalid or unsupported length causes an error, which the standard error handler will cause to be ignored.

The functions provided are as follows:

```
v := Ffloat( v, n )
```

```
 n := Ffix( v )
 v := Fabs( v )
v1 := Fcopy( v1, v2 )
v1 := Fadd ( v1, v2 )
v1 := Fsub ( v1, v2 )
v1 := Fmul ( v1, v2 )
v1 := Fdiv ( v1, v2 )
cmp:= Fcompare( v1, v2 )
v1 := Fsqrt   ( v1, v2 )
v1 := Fjitter ( v1, v2 )
```

where the vs are vector addresses and the ns are BCPL words holding integers.

The result from Fcompare will be -ve, 0 or +ve depending on whether the value in v1 is less than, equal to, or greater than that in v2.

The 'fuzz' used for equality comparisons may be set by a call to Fjitter, which sets the new value to that in v2, and returns the old one (which is initially zero) in v1. It should not be negative.

Fsqrt computes the square root of v2 and places the result in v1. A negative number causes an error.

6.2 Record I/O

Binary record I/O is a mode provided to make direct use of any underlying operating system primitives. Programs which use it should not expect to be portable, since such primitives may or may not exist, and may even be presented with a slightly different interface.

```
len := BinReadRecord ( record, maxlen )
 ok := BinWriteRecord( record, len    )
```

The result from BinReadRecord will be the number of characters read (which may be less than the given buffer length). Some additional information is put into the system error code global (LibErrorCode): if it has the value ERR.Success the transfer was of a complete record; if the value is ERR.IncompleteRecord there is more of this record waiting to be read. If an error occurs, the appropriate exception will be signalled. It is only possible to get an end-of-stream exception when no data characters were read (i.e. len will be zero). There is no obligation for an implementation to support mixed use of single character (character or binary modes) and record transfer on a stream.

6.3 Interval timer

If an interval timer is available, it may be invoked by the call of the form

```
ticks := IntervalTime()
```

The result will be given in terms of the manifest TicksPerSecond; what it represents is host dependent.

A program may arrange to be suspended for a given period of real time by calling Delay(ticks), which will only return when at least ticks time intervals (as defined by the TicksPerSecond manifest) have elapsed.

6.4 Random Access

Random Access facilities allow a user to re-position a sequential stream - they are to be distinguished from block-mode Direct Access, which is considerably less portable. The operations provided are:

```
stream  := FindInOut( fileDescriptor )
notevec := Note( notevec )
           Point( notevec )
```

The first opens an already existing file for updating (mode = IO.InOut). The fileDescriptor is a string such as might be passed to FindInput. Not all external files may be opened for update. The second remembers sufficient information about the position of a stream so as to be able to restore it later, by means of the third operation.

A notevec is a vector with upper bound NoteVecUPB; its contents are entirely private to the stream implementation, and only of use in a subsequent call of Point. Not all streams will support Note and Point; some may do so even though they were not opened in mode IO.InOut. It is an error to tamper with a notevec, and it may or may not be legal to reuse one on a stream which has been closed and re-opened in the meantime, depending upon the stream function.

7. Implementations

An implementation of this library on a particular host machine and operating system takes place by providing a relatively small set of interfaces to a host-independent 'kernel', which implements the bulk of the operations. If efficiency is important, it is of course possible to implement some of the code provided in the 'kernel' in the local assembly language, and omit the relevant sections from the BCPL parts of the library with suitable conditional compilation directives. For portability, however, BCPL algorithms are provided for everything possible.

The interface to the host-specific section is in two parts: that which is user-accessible (e.g. MapStack) and that which is private to the library (e.g. a 'real' LongJump). At present, the user-accessible part contains:

```
Level
TimeStamp
Coroutines
IsUndefined, Undefine
MapCode, MapStack, MapGlobals
StackBase, StackTop, GlobalBase, GlobalTop, MaxGlobal
The floating point package
BinReadRecord, BinWriteRecord
IntervalTime, Delay
```

and the private interface (defined in the file "PRIVHDR") has:

Begin
: sets up the standard environment & calls the user-supplied Start().

LevelToWords(level)
: Returns a BCPL (word) address in the stack, given the result of a call to Level().

PrivLongJump(coptr, level, label)
: Does a 'real' LongJump, to a different coroutine stack if coptr is not zero.

ExtendHeap(minUPB, maxUPB)
: Like GetMaxVec, but from the system heap.

FindFile(mode, name)
: Opens a named file in the given mode (one of IO.In, IO.Out, IO.InOut or IO.Append).

HPsecondaryUPB
: A variable to give the minimum number words to allocate at a time from the system heap.

Almost all BCPL code which has machine and operating system specificities is concentrated in the one module LBHOST; additional assembler modules may be needed to fully implement the library on a particular system.

If implementation-specific functions and routines are to be made available (e.g. to cope with parameter passing from the operating system), some convention which includes the name of the host system in the name of the routine should be used. (For example, the Tripos implementation uses names of the form TriposSendPkt.) This allows those parts of client programs which are host system specific to be easily identified, and clearly places the burden of choosing to use such facilities on the individual programmer.

7.1 Existing Implementations

An implementation of this library exists for the 68000 version of Tripos in the Computer Laboratory, and one for the CAP research computer is under way. Plans (and some initial investigations) have been made for a version for Berkeley UNIX running on VAX-11s.