FileListMethods:

A package for building and manipulating linked-list structures in REALbasic binary files

Why do this?

Many applications can benefit from treating the data fork of a file as a binary stream. We can have random access to data within it and are not constrained by the necessity of sequential reading and writing of a large amount of information in order to read or write just a portion of it. For applications that require frequent changes to the stored data, it is possible to implement storage management so that space in the file that is no longer required can be reused for new data.

This package implements linked-list capabilities in REALbasic binary streams. Initially carried out as an experiment to find out if performance would be acceptable, it has been applied successfully in two applications -- the first an accounting program that uses fixed-size blocks to store a large number of individual transaction items and the second a text-editing program that needs to store variable-length text data as well as formatting information.

Data types

The basic element of storage used by this package is fixed-size blocks of any size you choose to define. These are linked by forward and backward pointers into lists of arbitrary length. A list can include blocks of any of the defined sizes.

Each block has the following fields in order:

a forward pointer (a 4-byte integer interpreted as a byte offset within the file)
a reverse pointer (as above)
a 4-byte integer giving the size of the data field in bytes
a data field of the defined size

A pointer to a block is the byte address (0...) of the block's data field i.e. the forward and back pointers and the size field begin 12 bytes earlier in the file.

Lists start from list heads which are stored at byte offsets of 8, 12, 16, ... from the start of the file. Within the lists the blocks are linked by both forward and backward pointers. The null pointer has the value 0.

You are free to use the data field of a block in any way e.g. starting from a particular byte position within it, read or write integers, doubles, ASCII text, pointers to other lists etc.

For each block size there is a list head to attach a list of free blocks so that they can be reused. When a request is made to allocate a new block of a particular size, the free list for that size is searched first, before the overall file size is increased. If a block cannot be obtained from the free list, it is allocated at the end of the file whose size is thereby increased.

Methods are included for writing and reading text strings of arbitrary length, using linked blocks as the basic storage technique. It is possible for these strings can be built up from a sequence of arbitrary-length tokens before recording in the file, and then separated into their constituents on reading.

Methods for general list administration

initDB(nLists as integer, nTypes as integer)

Declares the structure to be created with a potential for data lists 0...(nList - 1) based at byte offsets 8, 12, 16, ... in the binary file, and 'nTypes' different block sizes.

declareBlock(type as integer, size as integer)

Declares a block type 'type' (0...) having 'size' bytes for data in addition to forward and backward pointers and an integer field specifying the data field size. (At present the number of different sizes is limited to 16 by a fixed-size array 'bSizes' used to store the data size of the different block types. There is no good reason for doing it this way and it can easily be changed.)

newBlock(bs as binaryStream, type as integer) as integer

Allocates a new block in the stream 'bs' of type 'type' and returns a pointer to its data field.

addToList(bs as binaryStream, blk as integer, n as integer)

Links the block 'blk' at the start of data list 'n', in stream 'bs'. The list is doubly-linked i.e. has both forward and backward pointers.

deleteFromList(bs as binaryStream, blk as integer, n as integer)

Deletes the block referenced by 'blk' from data list 'n' in stream 'bs'.

firstBlock(bs as binaryStream, dataList as integer) as integer

Returns a pointer to the first block of the list 'dataList' (0...), or 0 if that list is empty, in stream 'bs'.

nextBlock(bs as binaryStream, blk as integer) as integer

Given 'blk' as the pointer to a block, returns a pointer to the next block in the list in stream 'bs', or 0 if it is the last one.

prevBlock(bs as binaryStream, blk as integer) as integer

Given 'blk' as the pointer to a block, returns a pointer to the previous block in the list in stream 'bs', or 0 if it is the first one.

insertPre(bs as binaryStream, blk as integer, p as integer)

Inserts the block pointed to by 'blk' in stream 'bs' before the block pointed to by 'p'.

insertPost(bs as binaryStream, blk as integer, p as integer)

Inserts the block pointed to by 'blk' in stream 'bs' after the block pointed to by 'p'.

freeList(bs as binaryStream, blk as integer)

Returns to free storage the list pointed to by 'blk' in binaryStream 'bs'. If 'blk' is 0, then nothing happens. The list may contain blocks of mixed types.

freeBlock(bs as binaryStream, blk as integer)

Determines which free list the block referenced by 'blk' in stream 'bs' belongs to, and adds it to that list. (N.B. there is a separate free list for each block size, and the list heads for these are immediately after the list heads for data lists (0...)

nFreeBlocks(bs as binaryStream, type as integer) as integer

Returns the number of free blocks of type 'type' (0... 16) in stream 'bs'.

openNewFile(f as folderItem, byref bs as binaryStream) as boolean

Creates a file as folderItem 'f'and opens it as a binaryStream 'bs'. If successful, returns true. (N.B. there is a place provided to call your choice of error alert if an exception occurs in this process.)

openFile(f as folderItem, bs as binaryStream) as boolean

Attempts to open folderItem 'f' as binaryStream 'bs', returning true if successful.

Recording arbitrary text strings in list form

declareTextList(blkType as integer)

Declares that text lists will be created and read using blocks of type 'blkType' (0...).

textToList(bs as binaryStream, txt as string) as integer

Stores 'txt' as a linked list of blocks in stream 'bs' and returns a pointer to that list. The text string is stored in the file as: <bcd byte-count> ESC <text string> (where ESC stands for the character chr(27)).

If the text string is a succession of sub-strings, combine them thus:

<token1> ESC <token2> ESC <token3> ESC ...

before saving them in the file as a linked list. Then separate them with 'getString()' (below).

listToText(bs as binaryStream, list as integer) as string

If 'list' points to the first block storing a text string in stream 'bs', then the value of that string is returned. If the list is empty, the function returns "".

getString(byref tokenStr as string) as string

Returns the first sub-string from string 's' up to, but not including, the ESC character and then discards that and the ESC character from 's'. If 'tokenStr' is a composite string: <token1> ESC <token2> ESC ..., and has been recorded in the file as a list pointed to by 'p', the separate tokens can be recovered thus:

```
s = listToText(bs, p)
t1 = getString(s)
t2 = getString(s)
...
```

Using the package

Drag the icon for the package into a project. Because it is a REALbasic module, the method names are known globally to the whole project, as are the two properties 'numLists' and 'numSizes' if they should be needed.

Initializing a data file

At the beginning of a program should be something like this:

```
initDB(5, 5)          // 5 data lists, 5 block sizes
declareBlock(1, 152)  // block type for ...
declareBlock(2, 40)   // block type for ...
// Declare block type to be used for text stored in list form
declareTextList(1)
```

At present the values of the 'initDB( )' parameters 'nLists' and 'nTypes' are recorded in the file so that, in principle, the program could read them and thereby determine how many data lists there are and how many different types of block. The programs so written so far use only one file structure and simply assume that any file that is read will have been created with the same values of these constsnts, which are stored in the globals 'numLists' aznd 'numSizes' as a result of the call to 'initDB()' when the program starts up.

Storage and retrieval with blocks

Suppose that 'bs' is a binary stream and that we wish to store an integer, a double, and a short text string in a data block of type 2 and link it at the beginning of data list 1. Proceed as follows:

```
dim p as integer
p = newBlock(bs, 2)
addToList(bs, p, 1)
bs.position = p   // start of block's data area
bs.writeLong(myInteger)
bs.writeDouble(myDouble)
```

```
myString = left("xyz" + "          ", 10) // pad to a fixed length with spaces
bs.write(myString)
```

Note that the data format within the block is quite arbitrary and is simply determined by what the program writes in it.

To read from a block in this format, we first navigate to the block and then read its data:

```
p = firstBlock(bs, 1)  // start of the list
// now get to the block by whatever means
// is appropriate
p = nextBlock(bs, p)
...
// ok, we're there. now read
bs.position = p
myInteger = bs.readLong
myDouble = bs.readDouble
myString = bs.read(10)   // assuming 10-byte strings
```

Michael Maclean e:mail: m.maclean@paradise.net.nz
Christchurch
New Zealand