Flex Template B/flext

Purpose

To allocate flexible-sized blocks of memory as needed to hold arbitrary-length strings of text, stored actions or other block values.

B/flext.§1 Overview; §2 Blocks; §3 Multiple Blocks; §4 Head; §5 Block Routines; §6 Debugging Routines; §7 The Heap; §8 Initialisation; §9 Net Free Space; §10 Make Space; §11 Block Allocation; §12 Merging; §13 Recutting; §14 Deallocation; §15 Resizing; §16 Stubs

---

§**1. Overview.** Each I7 value is represented at run-time by an I6 word: on the Z-machine, a 16-bit number, and on Glulx, a 32-bit number. The correspondence between these numbers and the original values depends on the kind of value: "number" comes out as a signed twos-complement number, but "time" as an integer number of minutes since midnight, "rulebook" as the index of the rulebook in order of creation, and so on.

Even if a 32-bit number is available, this is not enough to represent the full range of values we might want: consider all the possible hundred-word essays of text, for instance. In some cases, then, the value is (either directly or indirectly) a pointer, telling the run-time code that the data is not in the value itself but can be found at a given location in memory. For instance, a "text" value is a (packed) address of either some encoded text or a routine to print text. When NI compiles references to a text value, it also creates the data to which this address belongs.

This works well if the data need not change in size, and if we never need to create or throw away values. But if we want a variable which can hold an arbitrary string of text which we will compose for ourselves, say, then we need at run-time to find space somewhere in memory to hold that data, and we need to be able to cope if that space runs out, and lastly we need a way to reclaim the memory again when the text's usefulness has finished. For instance, if a rule uses a temporary "let" variable which will hold indexed text then the block of indexed text data must be allocated; the variable must then be set to a pointer to this data; the rule must then run, and lastly the block of data deallocated again.

Values of this kind are called "block values" since they are pointers to blocks of memory. We have to be careful when making assignments to variables having these kinds of value. In I7, text like "change the motto text to the player's command" must not be compiled to I6 code such as `MT = PC;`, because that simply sets `MT` to have the same address as `PC`. There are now two independent pointers to the same piece of text, so that changing either one would change both, while the previous contents of `MT` are un-pointed to. The latter problem is almost as bad as the former, because it means that memory has been wasted forever – it would never be reclaimed. Repeat the process often enough and all the memory will be lost in this way: this is called a "leak".

I7 treats block values exactly like other values, as far as the writer is concerned. There is no concept of "a pointer to..." in I7 source text. Instead:

(1) For every allocated block of data, there is exactly one I7 value – stored in an I6 local or global variable, in a property, in a table entry, or even as part of another allocated block of data – which points to it.
(2) When assigning `Y` to `X`, we always perform a "deep copy" of the contents of the block pointed to by `Y` into that pointed to by `X`: we never copy pointers, which would be a "shallow copy".
(3) If `X` is the I6 representation of a block kind of value, then `X` is either 0 – meaning "not allocated yet" – or a valid pointer to a block of data. This is purely for efficiency's sake, making table columns of indexed text (for instance) a sparse representation when, as often happens, not many are filled in. When assigning to 0, we must first allocate a block, then change the pointer from 0 to point to it, and assign to the newly-allocated block. The user is oblivious to all of this.
(4) Any value which will be lost – for instance, a value in a variable which goes out of scope – must have its block deallocated first.
(5) Any value which is passed as an argument to a function must be copied first, as otherwise there are two pointers to the same block of data, one in the calling stack frame and the other in the one called;

in other words, we call by value, never by reference. (As we shall see, I7 phrases can indeed by defined which work by reference rather than by value, but those are defined using inline I6, not as function calls.)

As simple as this scheme looks – there is no need for garbage collection or reference counting – it is not entirely easy to get right. There are many complications, but the basic slogan is one pointer, one block.

§**2. Blocks.** A "block" is a continuous range of 2^n bytes of memory, where $n \geq 3$ for a 16-bit VM (i.e., for the Z-machine) and $n \geq 4$ for a 32-bit VM (i.e., on Glulx). Internally, a block is divided into a header followed by a data section.

The header consists of 4, 8 or 16 bytes, depending on the word size and the kind of block (see below). It always begins with a byte specifying n, the binary logarithm of its size: thus the largest block representable is 2^{255} bytes long, but somehow I think we can live with that. The second byte contains a bitmap of (at present) four flags, whose meanings will be explained below. The second *word* of the block, which might be at byte offset 2 or 4 from the start of the block depending on the word-size of the VM, is a number specifying the kind of value (KOV) which the block contains data of.

It might be objected that KOVs are not reducible to simple numbers. For instance, for any KOV K there is another KOV "list of K", so there is an infinite range of possibilities. "List of..." is what, in other languages, would be called a type constructor; whereas a KOV like "indexed text" is what would be called a base type, since it is not the result of any type constructor. In I7, there is a finite range of base types and type constructors, and these have distinct ID numbers: that is what is stored in the `BLK_HEADER_KOV` field. A block which has KOV "list of indexed texts" will have the same value here as a block which has KOV "list of numbers": it will store the I6 constant `LIST_OF_TY`. (To find out whether such a list does indeed contain numbers or texts – and it is essential to be able to do this – one must look at the data section of the block. See "Lists.i6t".)

The data section of a block begins at the byte offset `BLK_DATA_OFFSET` from the address of the block: but see below for how multiple-blocks behave differently.

These definitions must not be altered without making matching changes to the compiler.

```
Constant BLK_HEADER_N = 0;
Constant BLK_HEADER_FLAGS = 1;
Constant BLK_FLAG_MULTIPLE = $$00000001;
Constant BLK_FLAG_16_BIT   = $$00000010;
Constant BLK_FLAG_WORD     = $$00000100;
Constant BLK_FLAG_RESIDENT = $$00001000;
Constant BLK_HEADER_KOV = 1;
Constant BLK_DATA_OFFSET = 2*WORDSIZE;
```

§**3. Multiple Blocks.** There are two kinds of block values: those which can always be stored in a single block (for instance, a floating-point number stored in exactly 8 bytes of data would be suitable for this), and those which can change unpredictably in size and might at any point overflow their current storage, so that they may need to occupy multiple blocks (for instance, an indexed text). In such a multiple-block KOV, the data is stored in a doubly linked list of blocks, and the I6 value for the result is the pointer to the block which heads the linked list. For instance, the indexed text

```
"But now I worship a celestiall Sunne"
```

might be represented by an I6 value `BN` which points to a list of blocks like so:

```
NULL <-- BN: "But now I wor" <--> BN2: "ship a celestiall Sunne" --> NULL
```

Note that the unique pointer to `BN2` is the one in the header of the `BN` block. When we need to grow such a text, we add additional blocks; if the text should shrink, blocks at the end can at our discretion be deallocated. If the entire text should be deallocated, then all of the blocks used for it are deallocated, starting at the back and working towards the front.

A multiple-block is one whose flags byte contains the `BLK_FLAG_MULTIPLE`. This information is redundant since it could in principle be deduced from the kind of value stored in the block, which is recorded in the `-->BLK_HEADER_KOV` word, but that would be too slow. `BLK_FLAG_MULTIPLE` can never change for a currently allocated block, just as it can never change its KOV.

A multiple-block header is longer than that of an ordinary block, because it contains two extra words: `-->BLK_NEXT` is the next block in the doubly-linked list of blocks representing the current value, or `NULL` if this is the end; `-->BLK_PREV` is the previous block, or `NULL` if this is the beginning. The need to fit these two extra words in means that the data section is deferred, and so for a multiple-block data begins at the byte offset `BLK_DATA_MULTI_OFFSET` rather than `BLK_DATA_OFFSET`.

```
Constant BLK_DATA_MULTI_OFFSET = 4*WORDSIZE;
Constant BLK_NEXT 2;
Constant BLK_PREV 3;
```

§**4. Head.** On the Z-machine, though not always on Glulx, the entire heap has to be allocated at compile-time, and we tend to want to make it reasonably large to cover most eventualities (though the heap size is controllable with use options, so the user does have control over this).

Many small works of IF never have need of the heap at all, and at the same time can't spare the memory for it. NI only creates the constant `MEMORY_HEAP_SIZE`, the number of bytes initially given over to the heap, if need arises. So the code and arrays in this segment will never be compiled unless needed (but see also the "Stubs" paragraph below).

```
#IFDEF MEMORY_HEAP_SIZE;
! Constant SHOW_ALLOCATIONS = 1; ! Uncomment this for debugging purposes
```

§5. Block Routines.

```
[ BlkType txb;
    return txb-->BLK_HEADER_KOV;
];
[ BlkSize txb bsize n; ! Size of an individual block, including header
    if (txb == 0) return 0;
    for (bsize=1: n<txb->BLK_HEADER_N: bsize=bsize*2) n++;
    return bsize;
];
[ BlkTotalSize txb tsize; ! Combined size of multiple-blocks for a value
    if (txb == 0) return 0;
    if ((txb->BLK_HEADER_FLAGS) & BLK_FLAG_MULTIPLE == 0)
        return BlkSize(txb);
    for (:txb~=NULL:txb=txb-->BLK_NEXT) {
        tsize = tsize + BlkSize(txb);
    }
    return tsize;
];
```

§6. Debugging Routines.

These two routines are purely for testing the code.

```
[ BlkDebug txb n k i bsize tot dtot kov;
    if (txb == 0) "Block never created.";
    kov = txb-->BLK_HEADER_KOV;
    print "Block ", txb, " (kov ", kov, "): ";
    for (:txb~=NULL:txb = txb-->BLK_NEXT) {
        if (k++ == 100) " ... and so on.";
        if (txb-->BLK_HEADER_KOV ~= kov)
            print "*Wrong kov=", txb-->BLK_HEADER_KOV, "* ";
        n = txb->BLK_HEADER_N;
        for (bsize=1:n>0:n--) bsize=bsize*2;
        i = bsize - BLK_DATA_OFFSET;
        dtot = dtot+i;
        tot = tot+bsize;
        print txb, "(", bsize, ") > ";
    }
    print dtot, " data in ", tot, " bytes^";
];
[ BlkDebugDecomposition from to txb pf;
    if (to==0) to = NULL;
    for (txb=from:(txb~=to) && (txb~=NULL):txb=txb-->BLK_NEXT) {
        if (pf) print "+";
        print BlkSize(txb);
        pf = true;
    }
    print "^";
];
```

§**7. The Heap.** Properly speaking, a "heap" is a specific kind of structure often used for managing uneven-sized or unpredictably changing data. We use "heap" here in the looser sense of being an amorphous-sized collection of blocks of memory, some free, others allocated; our actual representation of free space on the heap is not a heap structure in computer science terms. (Though this segment could easily be rewritten to make it so, or to adopt any other scheme which might be faster, without modifying the rest of the template or NI itself.) The heap begins as a contiguous region of memory, but it need not remain so: on Glulx we use dynamic memory allocation to extend it.

For I7 purposes we don't need a way to represent allocated memory, only the free memory. A block is free if and only if it has `-->BLK_HEADER_KOV` equal to 0, which is never a valid kind of value, and also has the multiple flag set. We do that because we construct the whole collection of free blocks, at any given time, as a single, multiple-block "value": a doubly linked list joined by the `-->BLK_NEXT` and `<--BLK_PREV`.

A single block, at the bottom of memory and never moving, never allocated to anyone, is preserved in order to be the head of this linked list of free blocks. This is a 16-byte (i.e., $n = 4$) block, which we format when the heap is initialised in `HeapInitialise()`. Thus the heap is full if and only if the `-->BLK_NEXT` of the head-free-block is `NULL`.

So far we have described a somewhat lax regime. After many allocations and deallocations one could imagine the list of free blocks becoming a very long list of individually small blocks, which would both make it difficult to allocate large blocks, and also slow to look through the list. To ameliorate matters, we maintain the following invariants:

(a) In the free blocks list, `B-->BLK_NEXT` is always an address after `B`;
(b) For any contiguous run of free space blocks in memory (excluding the head-free-block), taking up a total of T bytes, the last block in the run has size 2^n where n is the largest integer such that $2^n \leq T$.

For instance, there can never be two consecutive free blocks of size 128: they would form a "run" in the sense of rule (b) of size $T = 256$, and when T is a power of two the run must contain a single block. In general, it's easy to prove that the number of blocks in the run is exactly the number of 1s when T is written out as a binary number, and that the blocks are ordered in memory from small to large (the reverse of the direction of reading, i.e., rightmost 1 digit first). Maintaining (b) is a matter of being careful to fragment blocks only from the front when smaller blocks are needed, and to rejoin from the back when blocks are freed and added to the free space object.

```
Array Blk_Heap -> MEMORY_HEAP_SIZE + 16; ! Plus 16 to allow room for head-free-block
```

§**8. Initialisation.** To recap: the constant `MEMORY_HEAP_SIZE` has been predefined by the NI compiler, and is always itself a power of 2, say 2^n. We therefore have $2^n + 2^4$ bytes available to us, and we format these as a free space list of two blocks: the 2^4-sized "head-free-block" described above followed by a 2^n-sized block exactly containing the whole of the rest of the heap.

```
[ HeapInitialise n bsize blk2;
    blk2 = Blk_Heap + 16;
    Blk_Heap->BLK_HEADER_N = 4;
    Blk_Heap-->BLK_HEADER_KOV = 0;
    Blk_Heap->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
    Blk_Heap-->BLK_NEXT = blk2;
    Blk_Heap-->BLK_PREV = NULL;
    for (bsize=1: bsize < MEMORY_HEAP_SIZE: bsize=bsize*2) n++;
    blk2->BLK_HEADER_N = n;
    blk2-->BLK_HEADER_KOV = 0;
    blk2->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
    blk2-->BLK_NEXT = NULL;
    blk2-->BLK_PREV = Blk_Heap;
];
```

§**9. Net Free Space.** "Net" in the sense of "after deductions for the headers": this is the actual number of free bytes left on the heap which could be used for data. Note that it is used to predict whether it is possible to fit something further in: so there are two answers, depending on whether the something is multiple-block data (with a larger header and therefore less room for data) or single-block data (smaller header, more room).

```
[ HeapNetFreeSpace multiple txb asize;
    for (txb=Blk_Heap-->BLK_NEXT: txb~=NULL: txb=txb-->BLK_NEXT) {
        asize = asize + BlkSize(txb);
        if (multiple) asize = asize - BLK_DATA_MULTI_OFFSET;
        else asize = asize - BLK_DATA_OFFSET;
    }
    return asize;
];
```

§**10. Make Space.** The following routine determines if there is enough free space to accommodate another `size` bytes of data, given that it has to be multiple-block data if the `multiple` flag is set. If the answer turns out to be "no", we see if the host virtual machine is able to allocate more for us: if it is, then we ask for 2^m further bytes, where 2^m is at least `size` plus the worst-case header storage requirement (16 bytes), and in addition is large enough to make it worth while allocating. We don't want to bother the VM by asking for trivial amounts of memory.

This looks to be more memory than is needed, since after all we've asked for enough that the new data can fit entirely into the new block allocated, and we might have been able to squeeze some of it into the existing free space. But it ensures that heap invariant (b) above is preserved, and besides, running out of memory tends to be something you don't do only once.

(The code below is a refinement on the original, suggested by Jesse McGrew, which handles non-multiple blocks better.)

```
Constant SMALLEST_BLK_WORTH_ALLOCATING = 12; ! i.e. 2^12 = 4096 bytes

[ HeapMakeSpace size multiple  newblocksize newblock B n;
    for (::) {
        if (multiple) {
            if (HeapNetFreeSpace(multiple) >= size) rtrue;
        } else {
            if (HeapLargestFreeBlock(0) >= size) rtrue;
        }
        newblocksize = 1;
        for (n=0: (n<SMALLEST_BLK_WORTH_ALLOCATING) || (newblocksize<size): n++)
            newblocksize = newblocksize*2;
        while (newblocksize < size+16) newblocksize = newblocksize*2;
        newblock = VM_AllocateMemory(newblocksize);
        if (newblock == 0) rfalse;
        newblock->BLK_HEADER_N = n;
        newblock-->BLK_HEADER_KOV = 0;
        newblock->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
        newblock-->BLK_NEXT = NULL;
        newblock-->BLK_PREV = NULL;
        for (B = Blk_Heap-->BLK_NEXT:B ~= NULL:B = B-->BLK_NEXT)
            if (B-->BLK_NEXT == NULL) {
                B-->BLK_NEXT = newblock;
                newblock-->BLK_PREV = B;
                jump Linked;
```

```
            }
        Blk_Heap-->BLK_NEXT = newblock;
        newblock-->BLK_PREV = Blk_Heap;
        .Linked; ;
        #ifdef SHOW_ALLOCATIONS;
        print "Increasing heap to free space map: "; BlkDebugDecomposition(Blk_Heap, 0);
        #endif;
    }
    rtrue;
];
[ HeapLargestFreeBlock multiple txb asize best;
    best = 0;
    for (txb=Blk_Heap-->BLK_NEXT: txb~=NULL: txb=txb-->BLK_NEXT) {
        asize = BlkSize(txb);
        if (multiple) asize = asize - BLK_DATA_MULTI_OFFSET;
        else asize = asize - BLK_DATA_OFFSET;
        if (asize > best) best = asize;
    }
    return best;
];
```

§**11. Block Allocation.** The routine `BlkAllocate(N, K, F)` allocates a block with room for `size` net bytes of data, which will have kind of value `K` and with flags `F`. If the flags include `BLK_FLAG_MULTIPLE`, this may be either a list of blocks or a single block. It returns either the address of the block or else throws run-time problem message and returns 0.

In allocation, we try to find a block which is as close as possible to the right size, and we may have to subdivide blocks: see case II below. For instance, if a block of size 2^n is available and we only need a block of size 2^k where $k < n$ then we break it up in memory as a sequence of blocks of size $2^k, 2^k, 2^{k+1}, 2^{k+2}, ..., 2^{n-1}$: note that the sum of these sizes is the 2^n we started with. We then use the first block of size 2^k. To continue the comparison with binary arithmetic, this is like a subtraction with repeated carries:

$$10000000_2 - 00001000_2 = 01111000_2$$

```
[ BlkAllocate size kov flags
    dsize n m free_block min_m max_m smallest_oversized_block secondhalf i hsize head tail;
    if (HeapMakeSpace(size, flags & BLK_FLAG_MULTIPLE) == false)
        return BlkAllocationError("ran out");
    ! Calculate the header size for a block of this KOV
    if (flags & BLK_FLAG_MULTIPLE) hsize = BLK_DATA_MULTI_OFFSET;
    else hsize = BLK_DATA_OFFSET;
    ! Calculate the data size
    n=0; for (dsize=1: dsize < hsize+size: dsize=dsize*2) n++;
    ! Seek a free block closest to the correct size, but starting from the
    ! block after the fixed head-free-block, which we can't touch
    min_m = 10000; max_m = 0;
    for (free_block = Blk_Heap-->BLK_NEXT:
        free_block ~= NULL:
        free_block = free_block-->BLK_NEXT) {
        m = free_block->BLK_HEADER_N;
        ! Current block the ideal size
        if (m == n) jump CorrectSizeFound;
```

```
    ! Current block too large: find the smallest which is larger than needed
    if (m > n) {
        if (min_m > m) {
            min_m = m;
            smallest_oversized_block = free_block;
        }
    }
    ! Current block too small: find the largest which is smaller than needed
    if (m < n) {
        if (max_m < m) {
            max_m = m;
        }
    }
}
if (min_m == 10000) {
    ! Case I: No block is large enough to hold the entire size
    if (flags & BLK_FLAG_MULTIPLE == 0) return BlkAllocationError("too fragmented");
    ! Set dsize to the size in bytes if the largest block available
    for (dsize=1: max_m > 0: dsize=dsize*2) max_m--;
    ! Split as a head (dsize-hsize), which we can be sure fits into one block,
    ! plus a tail (size-(dsize-hsize), which might be a list of blocks
    head = BlkAllocate(dsize-hsize, kov, flags);
    if (head == 0) return BlkAllocationError("head block not available");
    tail = BlkAllocate(size-(dsize-hsize), kov, flags);
    if (tail == 0) return BlkAllocationError("tail block not available");
    head-->BLK_NEXT = tail;
    tail-->BLK_PREV = head;
    return head;
}
! Case II: No block is the right size, but some exist which are too big
! Set dsize to the size in bytes of the smallest oversized block
for (dsize=1,m=1: m<=min_m: dsize=dsize*2) m++;
free_block = smallest_oversized_block;
while (min_m > n) {
    ! Repeatedly halve free_block at the front until the two smallest
    ! fragments left are the correct size: then take the frontmost
    dsize = dsize/2;
    secondhalf = free_block + dsize;
    secondhalf-->BLK_NEXT = free_block-->BLK_NEXT;
    if (secondhalf-->BLK_NEXT ~= NULL)
        (secondhalf-->BLK_NEXT)-->BLK_PREV = secondhalf;
    secondhalf-->BLK_PREV = free_block;
    free_block-->BLK_NEXT = secondhalf;
    free_block->BLK_HEADER_N = (free_block->BLK_HEADER_N) - 1;
    secondhalf->BLK_HEADER_N = free_block->BLK_HEADER_N;
    secondhalf-->BLK_HEADER_KOV = free_block-->BLK_HEADER_KOV;
    secondhalf->BLK_HEADER_FLAGS = free_block->BLK_HEADER_FLAGS;
    min_m--;
}
! Once that is done, free_block points to a block which is exactly the
! right size, so we can fall into...
! Case III: There is a free block which has the correct size.
```

```
    .CorrectSizeFound;
    ! Delete the free block from the double linked list of free blocks: note
    ! that it cannot be the head of this list, which is fixed
    if (free_block-->BLK_NEXT == NULL) {
        ! We remove final block, so previous is now final
        (free_block-->BLK_PREV)-->BLK_NEXT = NULL;
    } else {
        ! We remove a middle block, so join previous to next
        (free_block-->BLK_PREV)-->BLK_NEXT = free_block-->BLK_NEXT;
        (free_block-->BLK_NEXT)-->BLK_PREV = free_block-->BLK_PREV;
    }
    free_block-->BLK_HEADER_KOV = KindAtomic(kov);
    free_block->BLK_HEADER_FLAGS = flags;
    if (flags & BLK_FLAG_MULTIPLE) {
        free_block-->BLK_NEXT = NULL;
        free_block-->BLK_PREV = NULL;
    }
    ! Zero out the data bytes in the memory allocated
    for (i=hsize:i<dsize:i++) free_block->i=0;
    return free_block;
];
[ BlkAllocationError reason;
    print "*** Memory ", (string) reason, " ***^";
    RunTimeProblem(RTP_HEAPERROR);
    rfalse;
];
```

§**12. Merging.** Given a free block `block`, find the maximal contiguous run of free blocks which contains it, and then call `BlkRecut` to recut it to conform to invariant (b) above.

```
[ BlkMerge block first last pv nx;
    first = block; last = block;
    while (last-->BLK_NEXT == last+BlkSize(last))
        last = last-->BLK_NEXT;
    while ((first-->BLK_PREV + BlkSize(first-->BLK_PREV) == first) &&
        (first-->BLK_PREV ~= Blk_Heap))
        first = first-->BLK_PREV;
    pv = first-->BLK_PREV;
    nx = last-->BLK_NEXT;
    #ifdef SHOW_ALLOCATIONS;
    print "Merging: "; BlkDebugDecomposition(pv-->BLK_NEXT, nx); print "^";
    #endif;
    if (BlkRecut(first, last)) {
        #ifdef SHOW_ALLOCATIONS;
        print " --> "; BlkDebugDecomposition(pv-->BLK_NEXT, nx); print "^";
        #endif;
    }
];
```

§**13. Recutting.** Given a segment of the free block list, containing blocks known to be contiguous in memory, we recut into a sequence of blocks satisfying invariant (b): we repeatedly cut the largest 2^m-sized chunk off the back end until it is all used up.

```
[ BlkRecut first last tsize backsize mfrom mto bnext backend n dsize fine_so_far;
    if (first == last) rfalse;
    mfrom = first; mto = last + BlkSize(last);
    bnext = last-->BLK_NEXT;
    fine_so_far = true;
    for (:mto>mfrom: mto = mto - backsize) {
        for (n=0, backsize=1: backsize*2 <= mto-mfrom: n++) backsize=backsize*2;
        if ((fine_so_far) && (backsize == BlkSize(last))) {
            bnext = last; last = last-->BLK_PREV;
            bnext-->BLK_PREV = last;
            last-->BLK_NEXT = bnext;
            continue;
        }
        fine_so_far = false; ! From this point, "last" is meaningless
        backend = mto - backsize;
        backend->BLK_HEADER_N = n;
        backend-->BLK_HEADER_KOV = 0;
        backend->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
        backend-->BLK_NEXT = bnext;
        if (bnext ~= NULL) {
            bnext-->BLK_PREV = backend;
            bnext = backend;
        }
    }
    if (fine_so_far) rfalse;
    rtrue;
];
```

§**14. Deallocation.** There are two complications: first, when we free a multiple block we need to free all of the blocks in the list, starting from the back end and working forwards to the front – this is the job of `BlkFree`. Second, when any given block is freed it has to be put into the free block list at the correct position to preserve invariant (a): it might either come after all of the currently free blocks in memory, and have to be added to the end of the list, or in between two, and have to be inserted mid-list, but it can't be before all of them because the head-free-block is kept lowest in memory of all possible blocks. (Note that Glulx can't allocate memory dynamically which undercuts the ordinary array space created by I6: I6 arrays fill up memory from the bottom.)

Certain blocks *outside* the heap are marked as "resident" in memory, that is, are indestructible. This enables Inform to compile constant values.

```
[ BlkFree block fromtxb ptxb;
    if (block == 0) return;
    if ((block->BLK_HEADER_FLAGS) & BLK_FLAG_RESIDENT) return;
    BlkValueDestroy(block);
    if ((block->BLK_HEADER_FLAGS) & BLK_FLAG_MULTIPLE) {
        if (block-->BLK_PREV ~= NULL) (block-->BLK_PREV)-->BLK_NEXT = NULL;
        fromtxb = block;
        for (:(block-->BLK_NEXT)~=NULL:block = block-->BLK_NEXT) ;
        while (block ~= fromtxb) {
```

```
            ptxb = block-->BLK_PREV; BlkFreeSingleBlock(block); block = ptxb;
        }
    }
    BlkFreeSingleBlock(block);
];
[ BlkFreeSingleBlock block free nx;
    block-->BLK_HEADER_KOV = 0;
    block->BLK_HEADER_FLAGS = BLK_FLAG_MULTIPLE;
    for (free = Blk_Heap:free ~= NULL:free = free-->BLK_NEXT) {
        nx = free-->BLK_NEXT;
        if (nx == NULL) {
            free-->BLK_NEXT = block;
            block-->BLK_PREV = free;
            block-->BLK_NEXT = NULL;
            BlkMerge(block);
            return;
        }
        if (UnsignedCompare(nx, block) == 1) {
            free-->BLK_NEXT = block;
            block-->BLK_PREV = free;
            block-->BLK_NEXT = nx;
            nx-->BLK_PREV = block;
            BlkMerge(block);
            return;
        }
    }
];
```

§**15. Resizing.** When the content of a value stretches or shrinks, we will sometimes need to change the size of the block(s) containing the data – though not always: we might sometimes need to resize a 1052-byte text to a 1204-byte text and find that we are sitting in a 2048-byte block in any case. We either shed blocks from the end of the chain, or add new blocks at the end, that being the simplest thing to do. Sometimes it might mean preserving a not very efficient block division, but it minimises the churn of blocks being allocated and freed, which is probably good.

```
[ BlkResize block req newsize dsize newblk kov n i otxb flags;
    if (block == 0) "*** Cannot resize null block ***";
    kov = block-->BLK_HEADER_KOV;
    flags = block->BLK_HEADER_FLAGS;
    if (flags & BLK_FLAG_MULTIPLE == 0) "*** Cannot resize inextensible block ***";
    otxb = block;
    newsize = req;
    for (:: block = block-->BLK_NEXT) {
        n = block->BLK_HEADER_N;
        for (dsize=1: n>0: n--) dsize = dsize*2;
        i = dsize - BLK_DATA_MULTI_OFFSET;
        newsize = newsize - i;
        if (newsize > 0) {
            if (block-->BLK_NEXT ~= NULL) continue;
            newblk = BlkAllocate(newsize, kov, flags);
            if (newblk == 0) rfalse;
            block-->BLK_NEXT = newblk;
```

```
            newblk-->BLK_PREV = block;
            rtrue;
        }
        if (block-->BLK_NEXT ~= NULL) {
            BlkFree(block-->BLK_NEXT);
            block-->BLK_NEXT = NULL;
        }
        rtrue;
    }
];
[ DebugHeap;
    print "Managing a heap of initially ", MEMORY_HEAP_SIZE+16, " bytes.^";
    print HeapNetFreeSpace(false), " bytes currently free.^";
    print "Free space decomposition: "; BlkDebugDecomposition(Blk_Heap);
    print "Free space map: "; BlkDebug(Blk_Heap);
];
```

§**16. Stubs.** To ensure that the template code will still compile if `MEMORY_HEAP_SIZE` is undefined and there's no heap: none of these routines do anything in such a situation, but nor are they ever called – it's just that I6 source code may refer to them anyway, so they need to exist as routine names.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ HeapInitialise; ];
[ BlkFree; ];
[ DebugHeap;
    "This story file does not use a heap of managed memory.";
];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```