BlockValues Template B/blkvt

Purpose

Routines for copying, comparing, creating and destroying block values, and for reading and writing them as if they were arrays.

§**1. Head.** As with Flex.i6t, none of this material is worth compiling unless there is a heap on which the block values can be stored.

```
#IFDEF MEMORY_HEAP_SIZE;
```

§**2. Data Section.** In this segment, we provide a layer which comes between "Flex.i6t" and its eventual users. These are routines which handle the data section for the allocated blocks, and which isolate the end user from having to know how the blocks are divided. As far as the user is concerned, each block value has an "extent" E, and consists of an array of values indexed from 0 to $E-1$. These are bytes unless `BLK_FLAG_WORD` is set in the header, in which case they are words; unless `BLK_FLAG_16_BIT` is set, in which case they are two-byte quantities. (For instance, indexed text is stored using bytes on the Z-machine but 16-bit quantities on Glulx, since the ZSCII character set can fit into the range 0 to 255 but Unicode cannot.)

`BlkValueExtent(B)` returns the value E for the block `B`.

`BlkValueSetExtent(B, N)` resizes the block `B` so that value E is at least `N`. If this is a reduction, entries are lost from the end, i.e., from the highest-indexed entries. If it is an expansion, entries are added at the end and the existing entries are preserved.

`BlkValueRead(B, I)` returns the value of array entry `I` in `B`.

`BlkValueWrite(B, I, V)` puts the value `V` into array entry `I` in `B`.

```
[ BlkValueExtent block  tsize flags;
    if (block == 0) return 0;
    flags = block->BLK_HEADER_FLAGS;
    if (flags & BLK_FLAG_MULTIPLE == 0)
        tsize = BlkSize(block) - BLK_DATA_OFFSET;
    else
        for (:block~=NULL:block=block-->BLK_NEXT)
            tsize = tsize + BlkSize(block) - BLK_DATA_MULTI_OFFSET;
    if (flags & BLK_FLAG_16_BIT) return tsize/2;
    if (flags & BLK_FLAG_WORD) return tsize/WORDSIZE;
    return tsize;
];
[ BlkValueSetExtent block tsize flags wsize;
    if (block == 0) return 0;
    flags = block->BLK_HEADER_FLAGS; wsize = 1;
    if (flags & BLK_FLAG_WORD) wsize = WORDSIZE;
    if (flags & BLK_FLAG_16_BIT) wsize = 2;
    return BlkResize(block, (tsize)*wsize);
];
[ BlkValueRead block pos dsize hsize flags wsize ot op;
    if (block==0) rfalse;
    flags = block->BLK_HEADER_FLAGS; wsize = 1;
    if (flags & BLK_FLAG_WORD) wsize = WORDSIZE;
    if (flags & BLK_FLAG_16_BIT) wsize = 2;
    ot = block; op = pos;
    pos = pos*wsize;
    if (flags & BLK_FLAG_MULTIPLE) hsize = BLK_DATA_MULTI_OFFSET;
    else hsize = BLK_DATA_OFFSET;
    for (:block~=NULL:block=block-->BLK_NEXT) {
        dsize = BlkSize(block) - hsize;
        if ((pos >= 0) && (pos<dsize)) {
            block = block + hsize + pos;
            switch(wsize) {
                1: return block->0;
                2: #Iftrue (WORDSIZE == 2); return block-->0;
                    #ifnot; return (block->0)*256 + (block->1);
                    #endif;
                4: return block-->0;
            }
        }
        pos = pos - dsize;
    }
    "*** BlkValueRead: reading from index out of range: ", op, " in ", ot, " ***";
];
[ BlkValueWrite block pos val dsize hsize flags wsize ot op;
    if (block==0) rfalse;
    flags = block->BLK_HEADER_FLAGS; wsize = 1;
    if (flags & BLK_FLAG_WORD) wsize = WORDSIZE;
    if (flags & BLK_FLAG_16_BIT) wsize = 2;
    ot = block; op = pos;
    pos = pos*wsize;
    if (flags & BLK_FLAG_MULTIPLE) hsize = BLK_DATA_MULTI_OFFSET;
    else hsize = BLK_DATA_OFFSET;
    for (:block~=NULL:block=block-->BLK_NEXT) {
        dsize = BlkSize(block) - hsize;
        if ((pos >= 0) && (pos<dsize)) {
            block = block + hsize + pos;
            switch(wsize) {
                1: block->0 = val;
                2: #Iftrue (WORDSIZE == 2); block-->0 = val;
                    #ifnot; block->0 = (val/256)%256; block->1 = val%256;
                    #endif;
                4: block-->0 = val;
            }
            return;
        }
        pos = pos - dsize;
    }
    "*** BlkValueWrite: writing to index out of range: ", op, " in ", ot, " ***";
];
```

§3. KOV Support. To carry out the four fundamental operations on block values – creating, destroying, copying and comparing – we need to take account of what data they hold. Letting an indexed text go matters little: it contains only character codes. But letting a list of lists of numbers disappear is risky, because we might leave allocated blocks all over the heap for those individual lists of numbers which are now no longer in their parent list. Every KOV has different needs, because every KOV uses its data differently.

Each block KOV, then, provides a single "KOV support" function. These are named systematically by suffixing `_Support`: that is, the support function for `INDEXED_TEXT_TY` is called `INDEXED_TEXT_TY_Support` and so on. NI compiles a function called `KOVSupportFunction` which takes a KOV number as its argument and returns the relevant function.

The support function can be called with any of the following task constants as its first argument: it then has a further one to three arguments depending on the task in hand.

```
Constant CREATE_KOVS     = 1;
Constant CAST_KOVS       = 2;
Constant DESTROY_KOVS    = 3;
Constant PRECOPY_KOVS    = 4;
Constant COPY_KOVS       = 5;
Constant COMPARE_KOVS    = 6;
Constant READ_FILE_KOVS  = 7;
Constant WRITE_FILE_KOVS = 8;
Constant HASH_KOVS       = 9;
```

§4. Creation. To create a new value, we can't simply allocate a block and then fill it in some way appropriate to the kind of value needed: because we don't know what size of block would be a good idea, and even that much depends on the KOV. So the following routine delegates the whole process to the relevant KOV's support routines.

Depending on the KOV, we can also create it with a value cast from some existing value: for instance, an indexed text can be created with a value cast from the ordinary text "Marshmellow".

`K_Support(CREATE_KOVS, cast_from, skov)` – where `cast_from` and `skov` are optional arguments – is called to allocate a block on the heap and, if `cast_from` is non-zero, cast that data into the new block value. Of course `cast_from` itself is a value, and we don't specify its KOV: the support function is expected to know what it will get (if anything). `skov` stands for "strong kind of value" and is only applicable if `K` is a KOV constructor rather than a base KOV: for instance, if `K` is "list of...", then to create a "list of numbers" we would have `skov` equal to `NUMBER_TY`, whereas to create a "list of lists of texts" `skov` would be `LIST_OF_TY`.

```
Global block_value_tally;
[ BlkValueCreate kov cast_from skov  block sf;
    if (skov == 0 && (kov < 0 || kov >= BASE_KIND_HWM)) skov = kov;
    sf = KOVSupportFunction(kov);
    if (sf) block = sf(CREATE_KOVS, cast_from, skov);
    else { print "*** Impossible runtime creation ***^"; rfalse; }
#ifdef SHOW_ALLOCATIONS;
    print "[created ", kov, " at ", block, ": ", block_value_tally++, "]^";
#endif;
    return block;
];
```

§**5. Casting.** Just as we can create a value with a cast, we can also perform an assignment to an already-created block value in the form of a cast: well, for some kinds of value we can.

`K_Support(CAST_KOVS, fromval, fromkov, block)` casts from a value `fromval` with a specified KOV `fromkov` into the existing block value `block`.

```
[ BlkValueCast block tokov fromkov fromval  sf;
    sf = KOVSupportFunction(tokov);
    if (sf) return sf(CAST_KOVS, fromval, fromkov, block);
    else { print "*** Impossible runtime cast ***^"; rfalse; }
];
```

§**6. Destruction.** This must be called whenever the current contents of a block value is about to be lost, either because of overwriting or because the block value is in a variable which is going out of scope. What must be done depends on the kind of value: for instance, for indexed text nothing need be done, because the array entries only contains character values. But for a list of lists of numbers, for instance, array entries are pointers to lists of numbers. Simply overwriting or forgetting them means that the lists to which they point will be stuck on the heap forever, and will never be deallocated. So value destruction for a list involves destroying each individual entry and then deallocating it: and this may of course recurse.

`K_Support(DESTROY_KOVS, block)` takes whatever action is necessary to ensure that the values remaining in `block` can be discarded without leaving unreferenced data left on the heap.

```
[ BlkValueDestroy block  k rv sf;
    if (block == 0) return;
    k = block-->BLK_HEADER_KOV;
    sf = KOVSupportFunction(k);
    if (sf) return sf(DESTROY_KOVS, block);
    else { print "*** Impossible runtime deallocation ***^"; rfalse; }
];
```

§**7. Deep Copy.** A deep copy transfers the data contents – that is, the arrays described above – from one block to another; the old contents must be destroyed first. Note that we first perform a simplistic copy of the bits over, but that for some block values that won't be good enough. If the two blocks contain lists of lists, then each entry is itself a list, and a deep copy of each entry must also be performed.

`K_Support(COPY_KOVS, blockto, blockfrom)` takes whatever additional action might be necessary to ensure that pointers to heap data are not duplicated; it runs after the raw data has been copied bitwise and may only have to correct a few entries, or even none at all.

```
[ BlkValueCopy blockto blockfrom dsize i sf;
    if (blockto == 0) { print "*** Deep copy failed: destination empty ***^"; rfalse; }
    if (blockfrom == 0) { print "*** Deep copy failed: source empty ***^"; rfalse; }
    if (blockfrom->BLK_HEADER_N == 0) {
        ! A hack to handle precompiled array constants: N=0 blocks otherwise don't exist
        LIST_OF_TY_CopyRawArray(blockto, blockfrom, 1, 0);
        return blockto;
    }
    if (blockfrom-->BLK_HEADER_KOV ~= blockto-->BLK_HEADER_KOV) {
        print "*** Deep copy failed: types mismatch ***^"; rfalse;
    }
    BlkValueDestroy(blockto);
    dsize = BlkValueExtent(blockfrom);
    if (((blockfrom->BLK_HEADER_FLAGS) & BLK_FLAG_MULTIPLE) &&
        (BlkValueSetExtent(blockto, dsize, -1) == false)) {
        print "*** Deep copy failed: resizing failed ***^"; rfalse;
    }
    sf = KOVSupportFunction(blockfrom-->BLK_HEADER_KOV);
    if (sf) sf(PRECOPY_KOVS, blockto, blockfrom);
    for (i=0:i<dsize:i++) BlkValueWrite(blockto, i, BlkValueRead(blockfrom, i));
    if (sf) sf(COPY_KOVS, blockto, blockfrom);
    else { print "*** Impossible runtime copy ***^"; rfalse; }
    return blockto;
];
```

§**8. Comparison.** And it's a similar story with comparison.

`K_Support(COMPARE_KOVS, blockleft, blockright)` looks at the data in the two blocks and returns 0 if they are equal, a positive number if `blockright` is "greater than" `blockleft`, and a negative number if not. The interpretation of "greater than" depends on the KOV: it should be something which the user would find natural.

```
[ BlkValueCompare blockleft blockright  kov sf;
    if ((blockleft == 0) && (blockright == 0)) return 0;
    if (blockleft == 0) return 1;
    if (blockright == 0) return -1;
    if (blockleft-->BLK_HEADER_KOV ~= blockright-->BLK_HEADER_KOV)
        return blockleft-->BLK_HEADER_KOV - blockright-->BLK_HEADER_KOV;
    kov = blockleft-->BLK_HEADER_KOV;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(COMPARE_KOVS, blockleft, blockright);
    else { print "*** Impossible runtime comparison ***^"; rfalse; }
];
```

§**9. Creating Constants.** If the NI compiler reads, say, the string literal "Sea Devils" in the source text, and it is expecting to find indexed text – for instance if it's in a column of a table marked as containing indexed text – then how is this to find its way into memory as a block value? The heaps starts entirely empty, and NI cannot precompile parts of it: nor would we really want that, since it would hardwire assumptions about the heap's format into NI itself.

The answer is that at start-up time we perform creations of all of the constants we will need. Because they are constants, they will never be deallocated and never change in their contents – so they need not live on the heap at all, and in fact we store them elsewhere in memory. NI simply creates a blank array of the right 2^n size. Suppose that `X` is the address of this array. Then during the start-up rules, we at some point do this:

```
    BlkValueInitialCopy(X, BlkValueCreate(INDEXED_TEXT_TY, "Sea Devils"))
```

The effect is to create a suitable structure on the heap to hold this as indexed text, and then to copy it bitwise into `X`. A bitwise copy is ordinarily against the rules because it duplicates pointers, but of course the created block has been created for this purpose only: nothing points to it, and it is about to vanish – indeed the address of the block in question exists only briefly on the VM stack. (It is because of this that we don't perform a deep copy using the routine above.)

```
[ BlkValueInitialCopy blockto blockfrom dsize i;
    if (blockto == 0) { print "*** Initial copy failed: destination empty ***^"; rfalse; }
    if (blockfrom == 0) { print "*** Initial copy failed: source empty ***^"; rfalse; }
    dsize = 1; for (i=1: i<=blockfrom->BLK_HEADER_N: i++) dsize=dsize*2;
    for (i=0:i<dsize:i++) blockto->i = blockfrom->i;
    return blockto;
];
```

§10. Hashing. `K_Support(HASH_KOVS, block)` returns a hash value for the block. The algorithm used for hashing depends on the KOV, but it must have the property that two equivalent blocks (for which `COMPARE_KOVS` returns 0) produce the same hash value.

```
[ BlkValueHash block  kov sf;
    if (block == 0) return 0;
    kov = block-->BLK_HEADER_KOV;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(HASH_KOVS, block);
    else { print "*** Impossible runtime hashing ***^"; rfalse; }
];
[ KOVHashValue kov value;
    if (KOVIsBlockValue(kov)) return BlkValueHash(value);
    return value;
];
```

§11. Serialisation. Some block values can be written to external files (on Glulx): others cannot. The following routines abstract that.

If `ch` is -1, then `K_Support(READ_FILE_KOVS, block, auxf, ch)` returns `true` or `false` according to whether it is possible to read data from an auxiliary file `auxf` into the block value `block`. If `ch` is any other value, then the routine should do exactly that, taking `ch` to be the first character of the text read from the file which makes up the serialised form of the data.

`K_Support(WRITE_FILE_KOVS, block)` is simpler because, strictly speaking, it doesn't write to a file at all: it simply prints a serialised form of the data in `block` to the output stream. Since it is called only when that output stream has been redirected to an auxiliary file, and since the serialised form would often be illegible on screen, it seems reasonable to call it a file input-output function just the same.

```
[ BlkValueReadFromFile block auxf ch kov  sf;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(READ_FILE_KOVS, block, auxf, ch);
    rfalse;
];
[ BlkValueWriteToFile block kov  sf;
    sf = KOVSupportFunction(kov);
    if (sf) return sf(WRITE_FILE_KOVS, block);
    rfalse;
];
```

§**12. 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
[ BlkValueReadFromFile; rfalse; ];
[ BlkValueWriteToFile; rfalse; ];
[ BlkValueCreate x y z; ];
[ BlkValueDestroy x; ];
[ BlkValueCopy x y; ];
[ BlkValueCompare x y; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```