IndexedText Template B/inxt

Purpose

Code to support the indexed text kind of value.

§**1. Head.** As ever: if there is no heap, there are no indexed texts.

```
#IFDEF MEMORY_HEAP_SIZE; ! Will exist if any use is made of indexed texts
```

§**2. Character Set.** On the Z-machine, we use the 8-bit ZSCII character set, stored in bytes; on Glulx, we use the opening 16-bit subset of Unicode (which though only a subset covers almost all letter forms used on Earth), stored in two-byte half-words.

The Z-machine does have very partial Unicode support, but not in a way that can help us here. It is capable of printing a wide range of Unicode characters, and on a good interpreter with a good font (such as Zoom for Mac OS X, using the Lucida Grande font) can produce many thousands of glyphs. But it is not capable of printing those characters into memory rather than the screen, an essential technique for indexed texts: it can only write each character to a single byte, and it does so in ZSCII. That forces our hand when it comes to choosing the indexed-text character set.

```
#IFDEF TARGET_ZCODE;
Constant IT_Storage_Flags = BLK_FLAG_MULTIPLE;
Constant ZSCII_Tables;
#IFNOT;
Constant IT_Storage_Flags = BLK_FLAG_MULTIPLE + BLK_FLAG_16_BIT;
Constant Large_Unicode_Tables;
#ENDIF;
{-segment:UnicodeData.i6t}
{-segment:Char.i6t}
```

§**3. KOV Support.** See the "BlockValues.i6t" segment for the specification of the following routines.

```
[ INDEXED_TEXT_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     return INDEXED_TEXT_TY_Create(arg1);
        CAST_KOVS:       return INDEXED_TEXT_TY_Cast(arg1, arg2, arg3);
        DESTROY_KOVS:    rfalse;
        PRECOPY_KOVS:    rfalse;
        COPY_KOVS:       rfalse;
        COMPARE_KOVS:    return INDEXED_TEXT_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  if (arg3 == -1) rtrue;
                         return INDEXED_TEXT_TY_ReadFile(arg1, arg2, arg3);
        WRITE_FILE_KOVS: return INDEXED_TEXT_TY_WriteFile(arg1);
        HASH_KOVS:       return INDEXED_TEXT_TY_Hash(arg1);
    }
];
```

§**4. Creation.** Indexed texts are are simply "C strings", that is, the array entries in the block are a sequence of character codes terminated by the character code 0, which is free for this purpose in both ZSCII and Unicode. Since none of the data in an indexed-text is a pointer back onto the heap, it can all freely be bitwise copied or forgotten, which is why we need do nothing special to copy or destroy an indexed text.

Note that a freshly allocated block contains 0s in its data section, so its array entries already form a null-terminated empty text.

```
[ INDEXED_TEXT_TY_Create opcast x;
    x =  BlkAllocate(32, INDEXED_TEXT_TY, IT_Storage_Flags);
    if (opcast) INDEXED_TEXT_TY_Cast(opcast, TEXT_TY, x);
    return x;
];
```

§**5. Casting.** In general computing, "casting" is the process of translating data in one type into semantically equivalent data in another: for instance, translating the integer 1 into the floating point number 1.0, which will have an entirely different binary representation but has roughly the same meaning.

Here we are given a snippet – a word-selection of the player's command – or an ordinary text, and must cast it into an indexed text. In each case, what we do is simply to print out the value we have, but with the output stream set to memory rather than the screen. That gives us the character by character version, neatly laid out in an array, and all we have to do is to copy it into the indexed text and add a null termination byte.

What complicates things is that the two virtual machines handle printing to memory quite differently, and that the original text has unpredictable length. We are going to try printing it into the array `IT_MemoryBuffer`, but what if the text is too big? Disastrously, the Z-machine simply writes on in memory, corrupting all subsequent arrays and almost certainly causing the story file to crash soon after. There is nothing we can do to predict or avoid this, or to repair the damage: this is why the Inform documentation warns users to be wary of using indexed text with large strings in the Z-machine, and advises the use of Glulx instead. Glulx does handle overruns safely, and indeed allows us to dynamically allocate memory as necessary so that we can always avoid overruns entirely.

In either case, though, it's useful to have `IT_MemoryBufferSize`, the size of the temporary buffer, large enough that it will never be overrun in ordinary use. This is controllable with the use option "maximum indexed text length".

The following routine is not as messy as it looks: it is complicated by the fact that the Z-machine and Glulx (a) use different formats when printing text to memory, and (b) handle overruns differently, as explained above.

```
#ifndef IT_MemoryBufferSize;
Constant IT_MemoryBufferSize = 512;
#endif;

Constant IT_Memory_NoBuffers = 2;

#ifndef IT_Memory_NoBuffers;
Constant IT_Memory_NoBuffers = 1;
#endif;

#ifdef TARGET_ZCODE;
Array IT_MemoryBuffer -> IT_MemoryBufferSize*IT_Memory_NoBuffers; ! Where characters are bytes
#ifnot;
Array IT_MemoryBuffer --> (IT_MemoryBufferSize+2)*IT_Memory_NoBuffers; ! Where characters are words
#endif;

Global RawBufferAddress = IT_MemoryBuffer;
Global RawBufferSize = IT_MemoryBufferSize;
```

```
Global IT_cast_nesting;

[ INDEXED_TEXT_TY_Cast tx fromkov indt
    len i str oldstr offs realloc news buff buffx freebuff results;
    #ifdef TARGET_ZCODE;
    buffx = IT_MemoryBufferSize;
    #ifnot;
    buffx = (IT_MemoryBufferSize + 2)*WORDSIZE;
    #endif;
    buff = RawBufferAddress + IT_cast_nesting*buffx;
    IT_cast_nesting++;
    if (IT_cast_nesting > IT_Memory_NoBuffers) {
        buff = VM_AllocateMemory(buffx); freebuff = buff;
        if (buff == 0) {
            BlkAllocationError("ran out with too many simultaneous indexed text conversions");
            return;
        }
    }
    .RetryWithLargerBuffer;
    if (tx == 0) {
        #ifdef TARGET_ZCODE;
        buff-->0 = 1;
        buff->2 = 0;
        #ifnot;
        buff-->0 = 0;
        #endif;
        len = 1;
    } else {
        #ifdef TARGET_ZCODE;
        @output_stream 3 buff;
        #ifnot;
        if (unicode_gestalt_ok == false) { RunTimeProblem(RTP_NOGLULXUNICODE); jump Failed; }
        oldstr = glk_stream_get_current();
        str = glk_stream_open_memory_uni(buff, RawBufferSize, filemode_Write, 0);
        glk_stream_set_current(str);
        #endif;
        @push say__p; @push say__pc;
        ClearParagraphing();
        if (fromkov == SNIPPET_TY) print (PrintSnippet) tx;
        else {
            if (tx ofclass String) print (string) tx;
            if (tx ofclass Routine) (tx)();
        }
        @pull say__pc; @pull say__p;
        #ifdef TARGET_ZCODE;
        @output_stream -3;
        len = buff-->0;
        if (len > RawBufferSize-1) len = RawBufferSize-1;
        offs = 2;
        buff->(len+2) = 0;
        #ifnot; ! i.e. GLULX
        results = buff + buffx - 2*WORDSIZE;
```

```
        glk_stream_close(str, results);
        if (oldstr) glk_stream_set_current(oldstr);
        len = results-->1;
        if (len > RawBufferSize-1) {
            ! Glulx had to truncate text output because the buffer ran out:
            ! len is the number of characters which it tried to print
            news = RawBufferSize;
            while (news < len) news=news*2;
            news = news*4; ! Bytes rather than words
            i = VM_AllocateMemory(news);
            if (i ~= 0) {
                if (freebuff) VM_FreeMemory(freebuff);
                freebuff = i;
                buff = i;
                RawBufferSize = news/4;
                jump RetryWithLargerBuffer;
            }
            ! Memory allocation refused: all we can do is to truncate the text
            len = RawBufferSize-1;
        }
        offs = 0;
        buff-->(len) = 0;
        #endif;
        len++;
    }
    IT_cast_nesting--;
    if (indt == 0) {
        indt = BlkAllocate(len+1, INDEXED_TEXT_TY, IT_Storage_Flags);
        if (indt == 0) jump Failed;
    } else {
        if (BlkValueSetExtent(indt, len+1, 1) == false) { indt = 0; jump Failed; }
    }
    #ifdef TARGET_ZCODE;
    for (i=0:i<=len:i++) BlkValueWrite(indt, i, buff->(i+offs));
    #ifnot;
    for (i=0:i<=len:i++) BlkValueWrite(indt, i, buff-->(i+offs));
    #endif;
    .Failed;
    if (freebuff) VM_FreeMemory(freebuff);
    return indt;
];
```

§**6. Comparison.** This is more or less `strcmp`, the traditional C library routine for comparing strings.

```
[ INDEXED_TEXT_TY_Compare indtleft indtright pos ch1 ch2 dsizeleft dsizeright;
    dsizeleft = BlkValueExtent(indtleft);
    dsizeright = BlkValueExtent(indtright);
    for (pos=0:(pos<dsizeleft) && (pos<dsizeright):pos++) {
        ch1 = BlkValueRead(indtleft, pos);
        ch2 = BlkValueRead(indtright, pos);
        if (ch1 ~= ch2) return ch1-ch2;
        if (ch1 == 0) return 0;
    }
    if (pos == dsizeleft) return -1;
    return 1;
];
[ INDEXED_TEXT_TY_Distinguish indtleft indtright;
    if (INDEXED_TEXT_TY_Compare(indtleft, indtright) == 0) rfalse;
    rtrue;
];
```

§**7. Hashing.** This calculates a hash value for the string, using Bernstein's algorithm.

```
[ INDEXED_TEXT_TY_Hash indt  rv len i;
    rv = 0;
    len = BlkValueExtent(indt);
    for (i=0: i<len: i++)
        rv = rv * 33 + BlkValueRead(indt, i);
    return rv;
];
```

§**8. Printing.** Unicode is not the native character set on Glulx: it came along as a late addition to Glulx's specification. The deal is that we have to explicitly tell the Glk interface layer to perform certain operations in a Unicode way; if we simply perform `print (char) ch;` then the character `ch` will be printed in ZSCII rather than Unicode.

```
[ INDEXED_TEXT_TY_Say indt  ch i dsize;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    dsize = BlkValueExtent(indt);
    for (i=0:i<dsize:i++) {
        ch = BlkValueRead(indt, i);
        if (ch == 0) break;
        #ifdef TARGET_ZCODE;
        print (char) ch;
        #ifnot; ! TARGET_ZCODE
        glk_put_char_uni(ch);
        #endif;
    }
];
```

§**9. Serialisation.** Here we print a serialised form of an indexed text which can later be used to reconstruct the original text. The printing is apparently to the screen, but in fact always takes place when the output stream is a file.

The format chosen is a letter "S" for string, then a comma-separated list of decimal character codes, ending with the null terminator, and followed by a semicolon: thus `S65,66,67,0;` is the serialised form of the text "ABC".

```
[ INDEXED_TEXT_TY_WriteFile txb len pos ch;
    len = BlkValueExtent(txb);
    print "S";
    for (pos=0: pos<=len: pos++) {
        if (pos == len) ch = 0; else ch = BlkValueRead(txb, pos);
        if (ch == 0) {
            print "0;"; break;
        } else {
            print ch, ",";
        }
    }
];
```

§**10. Unserialisation.** If that's the word: the reverse process, in which we read a stream of characters from a file and reconstruct the indexed text which gave rise to them.

```
[ INDEXED_TEXT_TY_ReadFile indt auxf ch i v dg pos tsize;
    tsize = BlkValueExtent(indt);
    while (ch ~= 32 or 9 or 10 or 13 or 0 or -1) {
        ch = FileIO_GetC(auxf);
        if (ch == ',' or ';') {
            if (pos+1 >= tsize) {
                if (BlkValueSetExtent(indt, 2*pos, 20) == false) break;
                tsize = BlkValueExtent(indt);
            }
            BlkValueWrite(indt, pos++, v);
            v = 0;
            if (ch == ';') break;
        } else {
            dg = ch - '0';
            v = v*10 + dg;
        }
    }
    BlkValueWrite(indt, pos, 0);
    return indt;
];
```

§**11. Recognition-only-GPR.** An I6 general parsing routine to look at words from the position marker `wn` in the player's command to see if they match the contents of the indexed text `indt`, returning either `GPR_PREPOSITION` or `GPR_FAIL` according to whether a match could be made. This is used when the an object's name is set to include one of its properties, and the property in question is an indexed text: "A flowerpot is a kind of thing. A flowerpot has an indexed text called pattern. Understand the pattern property as describing a flowerpot." When the player types EXAMINE STRIPED FLOWERPOT, and there is a flowerpot in scope, the following routine is called to test whether its pattern property – an indexed text – matches any words at the position STRIPED FLOWERPOT. Assuming a pot does indeed have the pattern "striped", the routine advances `wn` by 1 and returns `GPR_PREPOSITION` to indicate a match.

This kind of GPR is called a "recognition-only-GPR", because it only recognises an existing value: it doesn't parse a new one.

```
[ INDEXED_TEXT_TY_ROGPR indt
    pos len wa wl wpos bdm ch own;
    if (indt == 0) return GPR_FAIL;
    bdm = true; own = wn;
    len = BlkValueExtent(indt);
    for (pos=0: pos<=len: pos++) {
        if (pos == len) ch = 0; else ch = BlkValueRead(indt, pos);
        if (ch == 32 or 9 or 10 or 0) {
            if (bdm) continue;
            bdm = true;
            if (wpos ~= wl) return GPR_FAIL;
            if (ch == 0) break;
        } else {
            if (bdm) {
                bdm = false;
                if (NextWordStopped() == -1) return GPR_FAIL;
                wa = WordAddress(wn-1);
                wl = WordLength(wn-1);
                wpos = 0;
            }
            if (wa->wpos ~= ch or IT_RevCase(ch)) return GPR_FAIL;
            wpos++;
        }
    }
    if (wn == own) return GPR_FAIL; ! Progress must be made to avoid looping
    return GPR_PREPOSITION;
];
```

§**12. Blobs.** That completes the compulsory services required for this KOV to function: from here on, the remaining routines provide definitions of text-related phrases in the Standard Rules.

What are the basic operations of text-handling? Clearly we want to be able to search, and replace, but that is left for the segment "RegExp.i6t" to handle. More basically we would like to be able to read and write characters from the text. But texts in I7 tend to be of natural language, rather than containing arbitrary material – that's indeed why we call them texts rather than strings. This means they are likely to be punctuated sequences of words, divided up perhaps into sentences and even paragraphs.

So we provide facilities which regard a text as being an array of "blobs", where a "blob" is a unit of text. The user can choose whether to see it as an array of characters, or words (of three different sorts: see the Inform documentation for details), or paragraphs, or lines.

```
Constant CHR_BLOB = 1; ! Construe as an array of characters
Constant WORD_BLOB = 2; ! Of words
Constant PWORD_BLOB = 3; ! Of punctuated words
Constant UWORD_BLOB = 4; ! Of unpunctuated words
Constant PARA_BLOB = 5; ! Of paragraphs
Constant LINE_BLOB = 6; ! Of lines
Constant REGEXP_BLOB = 7; ! Not a blob type as such, but needed as a distinct value
```

§**13. Blob Access.** The following routine runs a small finite-state-machine to count the number of blobs in an indexed text, using any of the above blob types (except `REGEXP_BLOB`, which is used for other purposes). If the optional arguments `cindt` and `wanted` are supplied, it also copies the text of blob number `wanted` (counting upwards from 1 at the start of the text) into the indexed text `cindt`. If the further optional argument `rindt` is supplied, then `cindt` is instead written with the original text `indt` as it would read if the blob in question were replaced with the indexed text in `rindt`.

```
Constant WS_BRM = 1;
Constant SKIPPED_BRM = 2;
Constant ACCEPTED_BRM = 3;
Constant ACCEPTEDP_BRM = 4;
Constant ACCEPTEDN_BRM = 5;
Constant ACCEPTEDPN_BRM = 6;
[ IT_BlobAccess indt blobtype cindt wanted rindt
    brm oldbrm ch i dsize csize blobcount gp cl j;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return 0;
    if (blobtype == CHR_BLOB) return IT_CharacterLength(indt);
    dsize = BlkValueExtent(indt);
    if (cindt) csize = BlkValueExtent(cindt);
    else if (rindt) "*** rindt without cindt ***";
    brm = WS_BRM;
    for (i=0:i<dsize:i++) {
        ch = BlkValueRead(indt, i);
        if (ch == 0) break;
        oldbrm = brm;
        if (ch == 10 or 13 or 32 or 9) {
            if (oldbrm ~= WS_BRM) {
                gp = 0;
                for (j=i:j<dsize:j++) {
                    ch = BlkValueRead(indt, j);
                    if (ch == 0) { brm = WS_BRM; break; }
                    if (ch == 10 or 13) { gp++; continue; }
                    if (ch ~= 32 or 9) break;
```

```
        }
        ch = BlkValueRead(indt, i);
        if (j == dsize) brm = WS_BRM;
        switch (blobtype) {
            PARA_BLOB: if (gp >= 2) brm = WS_BRM;
            LINE_BLOB: if (gp >= 1) brm = WS_BRM;
            default: brm = WS_BRM;
        }
    }
} else {
    gp = false;
    if ((blobtype == WORD_BLOB or PWORD_BLOB or UWORD_BLOB) &&
        (ch == '.' or ',' or '!' or '?'
                or '-' or '/' or '"' or ':' or ';'
                or '(' or ')' or '[' or ']' or '{' or '}'))
        gp = true;
    switch (oldbrm) {
        WS_BRM:
            brm = ACCEPTED_BRM;
            if (blobtype == WORD_BLOB) {
                if (gp) brm = SKIPPED_BRM;
            }
            if (blobtype == PWORD_BLOB) {
                if (gp) brm = ACCEPTEDP_BRM;
            }
        SKIPPED_BRM:
            if (blobtype == WORD_BLOB) {
                if (gp == false) brm = ACCEPTED_BRM;
            }
        ACCEPTED_BRM:
            if (blobtype == WORD_BLOB) {
                if (gp) brm = SKIPPED_BRM;
            }
            if (blobtype == PWORD_BLOB) {
                if (gp) brm = ACCEPTEDP_BRM;
            }
        ACCEPTEDP_BRM:
            if (blobtype == PWORD_BLOB) {
                if (gp == false) brm = ACCEPTED_BRM;
                else {
                    if ((ch == BlkValueRead(indt, i-1)) &&
                        (ch == '-' or '.')) blobcount--;
                    blobcount++;
                }
            }
        ACCEPTEDN_BRM:
            if (blobtype == WORD_BLOB) {
                if (gp) brm = SKIPPED_BRM;
            }
            if (blobtype == PWORD_BLOB) {
                if (gp) brm = ACCEPTEDP_BRM;
            }
        ACCEPTEDPN_BRM:
```

```
                    if (blobtype == PWORD_BLOB) {
                        if (gp == false) brm = ACCEPTED_BRM;
                        else {
                            if ((ch == BlkValueRead(indt, i-1)) &&
                                (ch == '-' or '.')) blobcount--;
                            blobcount++;
                        }
                    }
                }
            }
            if (brm == ACCEPTED_BRM or ACCEPTEDP_BRM) {
                if (oldbrm ~= brm) blobcount++;
                if ((cindt) && (blobcount == wanted)) {
                    if (rindt) {
                        BlkValueWrite(cindt, cl, 0);
                        IT_Concatenate(cindt, rindt, CHR_BLOB);
                        csize = BlkValueExtent(cindt);
                        cl = IT_CharacterLength(cindt);
                        if (brm == ACCEPTED_BRM) brm = ACCEPTEDN_BRM;
                        if (brm == ACCEPTEDP_BRM) brm = ACCEPTEDPN_BRM;
                    } else {
                        if (cl+1 >= csize) {
                            if (BlkValueSetExtent(cindt, 2*cl, 2) == false) break;
                            csize = BlkValueExtent(cindt);
                        }
                        BlkValueWrite(cindt, cl++, ch);
                    }
                } else {
                    if (rindt) {
                        if (cl+1 >= csize) {
                            if (BlkValueSetExtent(cindt, 2*cl, 3) == false) break;
                            csize = BlkValueExtent(cindt);
                        }
                        BlkValueWrite(cindt, cl++, ch);
                    }
                }
            } else {
                if ((rindt) && (brm ~= ACCEPTEDN_BRM or ACCEPTEDPN_BRM)) {
                    if (cl+1 >= csize) {
                        if (BlkValueSetExtent(cindt, 2*cl, 4) == false) break;
                        csize = BlkValueExtent(cindt);
                    }
                    BlkValueWrite(cindt, cl++, ch);
                }
            }
        }
        if (cindt) BlkValueWrite(cindt, cl++, 0);
        return blobcount;
];
```

§**14. Get Blob.** The front end which uses the above routine to read a blob. (Note that, for efficiency's sake, we read characters more directly.)

```
[ IT_GetBlob cindt indt wanted blobtype;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    if (blobtype == CHR_BLOB) return IT_GetCharacter(cindt, indt, wanted);
    IT_BlobAccess(indt, blobtype, cindt, wanted);
    return cindt;
];
```

§**15. Replace Blob.** The front end which uses the above routine to replace a blob. (Once again, characters are handled directly to avoid incurring all that overhead.)

```
[ IT_ReplaceBlob blobtype indt wanted rindt cindt ilen rlen i;
    if (blobtype == CHR_BLOB) {
        ilen = IT_CharacterLength(indt);
        rlen = IT_CharacterLength(rindt);
        wanted--;
        if ((wanted >= 0) && (wanted<ilen)) {
            if (rlen == 1) {
                BlkValueWrite(indt, wanted, BlkValueRead(rindt, 0));
            } else {
                cindt = BlkValueCreate(INDEXED_TEXT_TY);
                if (BlkValueSetExtent(cindt, ilen+rlen+1, 5)) {
                    for (i=0:i<wanted:i++)
                        BlkValueWrite(cindt, i, BlkValueRead(indt, i));
                    for (i=0:i<rlen:i++)
                        BlkValueWrite(cindt, wanted+i, BlkValueRead(rindt, i));
                    for (i=wanted+1:i<ilen:i++)
                        BlkValueWrite(cindt, rlen+i-1, BlkValueRead(indt, i));
                    BlkValueWrite(cindt, rlen+ilen, 0);
                    BlkValueCopy(indt, cindt);
                }
                BlkFree(cindt);
            }
        }
    } else {
        cindt = BlkValueCreate(INDEXED_TEXT_TY);
        IT_BlobAccess(indt, blobtype, cindt, wanted, rindt);
        BlkValueCopy(indt, cindt);
        BlkFree(cindt);
    }
];
```

§**16. Replace Text.** This is the general routine which searches for any instance of `findt`, as a blob, in `indt`, and replaces it with the text `rindt`. It works on any of the above blob-types, but two cases are special: first, if the blob-type is `CHR_BLOB`, then it can do more than search and replace for any instance of a single character: it can search and replace any instance of a substring, so that `findt` is not required to be only a single character. Second, if the blob-type is the special value `REGEXP_BLOB` then `findt` is interpreted as a regular expression rather than something literal to find: see "RegExp.i6t" for what happens next.

```
[ IT_ReplaceText blobtype indt findt rindt
    cindt csize ilen flen i cl mpos ch chm whitespace punctuation;
    if (blobtype == REGEXP_BLOB or CHR_BLOB)
        return IT_Replace_RE(blobtype, indt, findt, rindt);
    ilen = IT_CharacterLength(indt);
    flen = IT_CharacterLength(findt);
    cindt = BlkValueCreate(INDEXED_TEXT_TY);
    csize = BlkValueExtent(cindt);
    mpos = 0;
    whitespace = true; punctuation = false;
    for (i=0:i<=ilen:i++) {
        ch = BlkValueRead(indt, i);
        .MoreMatching;
        chm = BlkValueRead(findt, mpos++);
        if (mpos == 1) {
            switch (blobtype) {
                WORD_BLOB:
                    if ((whitespace == false) && (punctuation == false)) chm = -1;
            }
        }
        whitespace = false;
        if (ch == 10 or 13 or 32 or 9) whitespace = true;
        punctuation = false;
        if (ch == '.' or ',' or '!' or '?'
            or '-' or '/' or '"' or ':' or ';'
            or '(' or ')' or '[' or ']' or '{' or '}') {
            if (blobtype == WORD_BLOB) chm = -1;
            punctuation = true;
        }
        if (ch == chm) {
            if (mpos == flen) {
                if (i == ilen) chm = 0;
                else chm = BlkValueRead(indt, i+1);
                if ((blobtype == CHR_BLOB) ||
                    (chm == 0 or 10 or 13 or 32 or 9) ||
                    (chm == '.' or ',' or '!' or '?'
                        or '-' or '/' or '"' or ':' or ';'
                        or '(' or ')' or '[' or ']' or '{' or '}')) {
                    mpos = 0;
                    cl = cl - (flen-1);
                    BlkValueWrite(cindt, cl, 0);
                    IT_Concatenate(cindt, rindt, CHR_BLOB);
                    csize = BlkValueExtent(cindt);
                    cl = IT_CharacterLength(cindt);
                    continue;
                }
```

```
            }
        } else {
            mpos = 0;
        }
        if (cl+1 >= csize) {
            if (BlkValueSetExtent(cindt, 2*cl, 9) == false) break;
            csize = BlkValueExtent(cindt);
        }
        BlkValueWrite(cindt, cl++, ch);
    }
    BlkValueCopy(indt, cindt);
    BlkFree(cindt);
];
```

§**17. Character Length.** When accessing at the character-by-character level, things are much easier and we needn't go through any finite state machine palaver.

```
[ IT_CharacterLength indt ch i dsize;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return 0;
    dsize = BlkValueExtent(indt);
    for (i=0:i<dsize:i++) {
        ch = BlkValueRead(indt, i);
        if (ch == 0) return i;
    }
    return dsize;
];
[ INDEXED_TEXT_TY_Empty indt;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) rfalse;
    if (IT_CharacterLength(indt) == 0) rtrue;
    rfalse;
];
```

§**18. Get Character.** Characters in a text are numbered upwards from 1 by the users of this routine: which is why we subtract 1 when reading the array in the block-value, which counts from 0.

```
[ IT_GetCharacter cindt indt i ch;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    if ((i<=0) || (i>IT_CharacterLength(indt))) ch = 0;
    else ch = BlkValueRead(indt, i-1);
    BlkValueWrite(cindt, 0, ch);
    BlkValueWrite(cindt, 1, 0);
    return cindt;
];
```

§**19. Casing.** In many programming languages, characters are a distinct data type from strings, but not in I7. To I7, a character is simply an indexed text which happens to have length 1 – this has its inefficiencies, but is conceptually easy for the user.

`IT_CharactersOfCase(indt, case)` determines whether all the characters in `indt` are letters of the given casing: 0 for lower case, 1 for upper case. In the case of ZSCII, this is done correctly handling all of the European accented letters; in the case of Unicode, it follows the Unicode standard.

Note that there is no requirement for `indt` to be only a single character long.

```
[ IT_CharactersOfCase indt case i ch len;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) rfalse;
    len = IT_CharacterLength(indt);
    for (i=0:i<len:i++) {
        ch = BlkValueRead(indt, i);
        if ((ch) && (CharIsOfCase(ch, case) == false)) rfalse;
    }
    rtrue;
];
```

§**20. Change Case.** We set `cindt` to the text in `indt`, except that all the letters are converted to the `case` given (0 for lower, 1 for upper). The definition of what is a "letter", what case it has and what the other-case form is are as specified in the ZSCII and Unicode standards.

```
[ IT_CharactersToCase cindt indt case i ch len bnd;
    if ((indt==0) || (BlkType(indt) ~= INDEXED_TEXT_TY)) return;
    len = IT_CharacterLength(indt);
    if (BlkValueSetExtent(cindt, len+1, 11) == false) return cindt;
    bnd = 1;
    for (i=0:i<len:i++) {
        ch = BlkValueRead(indt, i);
        if (case < 2) {
            BlkValueWrite(cindt, i, CharToCase(ch, case));
        } else {
            BlkValueWrite(cindt, i, CharToCase(ch, bnd));
            if (case == 2) {
                bnd = 0;
                if (ch == 0 or 10 or 13 or 32 or 9
                    or '.' or ',' or '!' or '?'
                    or '-' or '/' or '"' or ':' or ';'
                    or '(' or ')' or '[' or ']' or '{' or '}') bnd = 1;
            }
            if (case == 3) {
                if (ch ~= 0 or 10 or 13 or 32 or 9) {
                    if (bnd == 1) bnd = 0;
                    else {
                        if (ch == '.' or '!' or '?') bnd = 1;
                    }
                }
            }
        }
    }
    BlkValueWrite(cindt, len, 0);
    return cindt;
];
```

§**21. Concatenation.** To concatenate two indexed texts is to place one after the other: thus "green" concatenated with "horn" makes "greenhorn". In this routine, `indt_from` would be "horn", and is added at the end of `indt_to`, which is returned in its expanded state.

When the blob type is `REGEXP_BLOB`, the routine is used not for simple concatenation but to handle the concatenations occurring when a regular expression search-and-replace is going on: see "RegExp.i6t".

```
[ IT_Concatenate indt_to indt_from blobtype indt_ref
    pos len ch i tosize x y case;
    if ((indt_to==0) || (BlkType(indt_to) ~= INDEXED_TEXT_TY)) rfalse;
    if ((indt_from==0) || (BlkType(indt_from) ~= INDEXED_TEXT_TY)) return indt_to;
    switch(blobtype) {
        CHR_BLOB, 0:
            pos = IT_CharacterLength(indt_to);
            len = IT_CharacterLength(indt_from);
            if (BlkValueSetExtent(indt_to, pos+len+1, 10) == false) return indt_to;
            for (i=0:i<len:i++) {
                ch = BlkValueRead(indt_from, i);
                BlkValueWrite(indt_to, i+pos, ch);
            }
            BlkValueWrite(indt_to, len+pos, 0);
            return indt_to;
        REGEXP_BLOB:
            return IT_RE_Concatenate(indt_to, indt_from, blobtype, indt_ref);
        default:
            print "*** IT_Concatenate used on impossible blob type ***^";
            rfalse;
    }
];
```

§**22. Setting the Player's Command.** In effect, the text typed most recently by the player is a sort of indexed text already, though it isn't in indexed text format, and doesn't live on the heap. (We can't simply make it an indexed text, as tidy as that would seem, because then no I7 work could ever compile without a heap to use, and that would severely affect many works which have to fit in the Z-machine and can't afford the storage for a heap.)

```
[ SetPlayersCommand indt_from i len at;
    len = IT_CharacterLength(indt_from);
    if (len > 118) len = 118;
    #ifdef TARGET_ZCODE;
    buffer->1 = len; at = 2;
    #ifnot;
    buffer-->0 = len; at = 4;
    #endif;
    for (i=0:i<len:i++) buffer->(i+at) = CharToCase(BlkValueRead(indt_from, i), 0);
    for (:at+i<120:i++) buffer->(at+i) = ' ';
    VM_Tokenise(buffer, parse);
    players_command = 100 + WordCount(); ! The snippet variable ''player's command''
];
```

§**23. Stubs.** And the usual meaningless versions to ensure that function-names exist if there is no heap, and there are no indexed texts anyway.

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ INDEXED_TEXT_TY_Support t a b c; rfalse; ];
[ INDEXED_TEXT_TY_Say indt; ];
[ SetPlayersCommand indt_from; ];
[ INDEXED_TEXT_TY_Create; ];
[ INDEXED_TEXT_TY_Cast a b c; ];
[ INDEXED_TEXT_TY_Empty t; rfalse; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```