Tables Template B/tabt

Purpose

To read, write, search and allocate rows in the Table data structure.

§**1. Format.** The I7 Table structure is not to be confused with the I6 `table` form of array: it is essentially a two-dimensional array which has some metadata at the top of each column.

The run-time representation for a Table is the address `T` of an I6 `table` array: that is, `T-->0` holds the number of columns (which is at most 99) and `T-->i` is the address of column number `i`. Columns are therefore numbered from 1 to `T-->0`, but they are also identified by an ID number of 100 or more, with each different column name having its own ID number. (This is so that multiple tables can share columns with the same name, and correlate them: it also means that NI's type-checking machinery can know the kind of value of a table entry from the name of the column alone.)

Each column `C` is also a `table` array, with `C-->1` holding the unique ID number for the column's name, `C-->2` holding the blank entry flags offset and `C-->3` up to `C-->(C-->0)` holding the entries.

`C-->1` also contains four upper bit flags. These are also defined in "Tables.w" in the NI source, and the values must agree.

```
Constant TB_COLUMN_SIGNED      $4000;
Constant TB_COLUMN_TOPIC       $2000;
Constant TB_COLUMN_DONTSORTME  $1000;
Constant TB_COLUMN_NOBLANKBITS $0800;
Constant TB_COLUMN_CANEXCHANGE $0400;
Constant TB_COLUMN_ALLOCATED   $0200;
Constant TB_COLUMN_NUMBER      $01ff; ! Mask to remove upper bit flags
Constant COL_HSIZE 2; ! Column header size: two words (ID/flags, blank bits)
```

§**2. Find Column.** Columns can be referenced either by their physical column numbers – from 1 to, potentially, 99 – or else by unique ID numbers associated with column names. For instance, if a table has a column called "liquid capacity", then all references to its "liquid capacity entry" are via the ID number associated with this column name, which will be $\geq$ 100 and on the other hand $\leq$ `TB_COLUMN_NUMBER`. At present, this is only 511, so there can be at most 411 different column names across all the tables present in the source text. (It is just about possible to imagine this being a problem on a very large work, so we will probably one day revise the above to make use of the larger word-size in Glulx and so raise this limit. But so far nobody has got even close to it.)

```
[ TableFindCol tab col f i no_cols n;
    no_cols = tab-->0;
    for (i=1: i<=no_cols: i++)
        if (col == ((tab-->i)-->1) & TB_COLUMN_NUMBER) return i;
    if (f) { RunTimeProblem(RTP_TABLE_NOCOL, tab); return 0; }
    return 0;
];
```

§**3. Number of Rows.** The columns in a table can be assumed all to have the same height (i.e., number of rows): thus the number of rows in `T` can be calculated by looking at column 1, thus...

```
[ TableRows tab first_col;
    first_col = tab-->1; if (first_col == 0) return 0;
    return (first_col-->0) - COL_HSIZE;
];
```

§**4. Blanks.** Each table entry is stored in a single word in memory: indeed, column `C` row `R` is at address `(T-->C)-->(R+COL_HSIZE)`.

But this is not sufficient storage in all cases, because each entry can be either a value or can be designated "blank". Since, for some columns at least, the possible values include every number, we find that we have to store $2^{16} + 1$ possibilities given only a 16-bit memory word. (Well, or $2^{32} + 1$ with a 32-bit word, depending on the virtual machine.) This cannot be done.

We therefore need, at least in some cases, an additional bit of storage for each table entry which indicates whether or not it is blank. If we provided such a bit for every table entry, that would be a fairly simple system to implement, but it would also be wasteful of memory, with an overhead of about 5% in practice: and memory in the virtual machine is in very short supply. The reason such a system would be wasteful is that many columns are known to hold values which are in a narrow range; for instance, a time of day cannot exceed 1440, and there will never be more than 10,000 rulebooks or scenes, and so on. For such columns it would be more efficient and indeed faster to indicate blankness by using an exceptional value in the memory cell which is such that it cannot be valid for the kind of value stored in the column. We therefore provide a "blanks bitmap" for only some columns.

This leads us to define the following dummy value, chosen so that it is both impossible for most kinds of value – which is easy to arrange – and also unlikely for even those kinds of value where it is legal. For instance, -1 would be impossible for enumerative kinds of value such as rulebooks and scenes, but it would be a poor choice for the dummy value because it occurs pretty often as an integer. Instead we use the constant `IMPROBABLE_VALUE`, whose value depends on the word size of the virtual machine, and which is declared in "Definitions.i6t".

An entry is therefore blank if and only if either

(a) its column has no blanks bitmap and the stored entry is `TABLE_NOVALUE`, or
(b) its column does have a blanks bitmap, the blanks bit for this entry is set, and the stored entry is also `TABLE_NOVALUE`.

To look up the blanks bitmap is a little slower than to access the stored entry directly. Most of the time, entries accessed will be non-blank: so it is efficient to have a system where we can quickly determine this. If we look at the entry and find that it is not `TABLE_NOVALUE`, then we know it is not a blank. If we find that it is `TABLE_NOVALUE`, on the other hand, then quite often the column has no blanks bitmap and again we have a quick answer: it's blank. Only if the column also has a blanks bitmap do we need to check that we haven't got a false negative. (The more improbable `TABLE_NOVALUE` is as a stored value, the rarer it is that we have to check the blanks bitmap for a non-blank entry.)

```
Constant TABLE_NOVALUE = IMPROBABLE_VALUE;
```

§**5. Masks.** The blanks bitmaps are stored as bytes; we therefore need a quick way to test or set whether bit number i of a byte is zero, where $0 \leq i \leq 7$. I6 provides no very useful operators here, whereas memory lookup is cheap, so we use two arrays of bitmaps:

```
Array CheckTableEntryIsBlank_LU
    -> $$00000001
        $$00000010
        $$00000100
        $$00001000
        $$00010000
        $$00100000
        $$01000000
        $$10000000;
Array CheckTableEntryIsNonBlank_LU
    -> $$11111110
        $$11111101
        $$11111011
        $$11110111
        $$11101111
        $$11011111
        $$10111111
        $$01111111;
```

§**6. Testing Blankness.** The following routine is the one which checks that there is no false negative: it should be used when we know that the table entry is `TABLE_NOVALUE` and we need to check the blank bit, if there is one, to make sure the entry is indeed blank.

The second word in the column table header, `C-->2`, holds the address of the blanks bitmap: this in turn contains one bit for each row, starting with the least significant bit of the first byte. If the table contains a number of rows which isn't a multiple of 8, the spare bits at the end of the last byte in the blanks bitmap are wasted, but this is an acceptable overhead in practice.

```
[ CheckTableEntryIsBlank tab col row i at;
    if (col >= 100) col = TableFindCol(tab, col);
    if (col == 0) rtrue;
    if ((tab-->col)-->(row+COL_HSIZE) ~= TABLE_NOVALUE) {
        print "*** CTEIB on nonblank value ", tab, " ", col, " ", row, " ***^";
    }
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) rtrue;
    row--;
    at = ((tab-->col)-->2) + (row/8);
    if ((TB_Blanks->at) & (CheckTableEntryIsBlank_LU->(row%8))) rtrue;
    rfalse;
];
```

§**7. Force Entry Blank.** We blank a table cell by storing `TABLE_NOVALUE` in its entry word and also setting the relevant bit in the blanks bitmap, if there is one.

We need to be careful if the column holds a kind of value where values are pointers to blocks of allocated memory, because if so then overwriting such a value might lead to a memory leak. So in such cases we call `BlkFree` to free the memory block. (Note that each memory block is pointed to by one and only one I7 value at any given time: we are using them as values, not pointers to values. So if this reference is deleted, it's by definition the only one.) `TABLE_NOVALUE` is chosen such that it cannot be an address of a memory block, which is convenient here. (The value 0 means "no memory block allocated yet".)

```
[ ForceTableEntryBlank tab col row i at oldv flags;
    if (col >= 100) col = TableFindCol(tab, col);
    if (col == 0) rtrue;
    flags = (tab-->col)-->1;
    oldv = (tab-->col)-->(row+COL_HSIZE);
    if ((flags & TB_COLUMN_ALLOCATED) && (oldv ~= 0 or TABLE_NOVALUE))
        BlkFree(oldv);
    (tab-->col)-->(row+COL_HSIZE) = TABLE_NOVALUE;
    if (flags & TB_COLUMN_NOBLANKBITS) return;
    row--;
    at = ((tab-->col)-->2) + (row/8);
    (TB_Blanks->at) = (TB_Blanks->at) | (CheckTableEntryIsBlank_LU->(row%8));
];
```

§**8. Force Entry Non-Blank.** To unblank a cell, we need to clear the relevant bit in the bitmap. We then go on to write a new value in to the entry – thus overwriting the `TABLE_NOVALUE` value – but that isn't done here; the expectation is that whoever calls this routine is just about to write a new entry anyway.

The exception is again for columns holding a kind of value pointing to a memory block, where we create a suitable initialised but uninteresting memory block for the KOV in question, and set the entry to that.

```
[ ForceTableEntryNonBlank tab col row i at oldv flags tc kov;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rtrue;
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) return;
    flags = (tab-->col)-->1;
    oldv = (tab-->col)-->(row+COL_HSIZE);
    if ((flags & TB_COLUMN_ALLOCATED) &&
        (oldv == 0 or TABLE_NOVALUE)) {
        kov = UNKNOWN_TY;
        tc = ((tab-->col)-->1) & TB_COLUMN_NUMBER;
        kov = TC_KOV(tc);
        if (kov ~= UNKNOWN_TY) {
            i = kov;
            if (KindBaseArity(i) > 0) i = KindAtomic(i); else i = 0;
            (tab-->col)-->(row+COL_HSIZE) = BlkValueCreate(kov, 0, i);
        }
    }
    row--;
    at = ((tab-->col)-->2) + (row/8);
    (TB_Blanks->at) = (TB_Blanks->at) & (CheckTableEntryIsNonBlank_LU->(row%8));
];
```

§**9. Swapping Blank Bits.** When sorting a table, we obviously need to swap rows from time to time; if any of its columns have blanks bitmaps, then the relevant bits in them need to be swapped to match, and the following routine performs this operation for two rows in a given column.

```
[ TableSwapBlankBits tab row1 row2 col at1 at2 bit1 bit2;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rtrue;
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) return;
    row1--;
    at1 = ((tab-->col)-->2) + (row1/8);
    row2--;
    at2 = ((tab-->col)-->2) + (row2/8);
    bit1 = ((TB_Blanks->at1) & (CheckTableEntryIsBlank_LU->(row1%8)));
    bit2 = ((TB_Blanks->at2) & (CheckTableEntryIsBlank_LU->(row2%8)));
    if (bit1) bit1 = true;
    if (bit2) bit2 = true;
    if (bit1 == bit2) return;
    if (bit1) {
        (TB_Blanks->at1)
            = (TB_Blanks->at1) & (CheckTableEntryIsNonBlank_LU->(row1%8));
        (TB_Blanks->at2)
            = (TB_Blanks->at2) | (CheckTableEntryIsBlank_LU->(row2%8));
    } else {
        (TB_Blanks->at1)
            = (TB_Blanks->at1) | (CheckTableEntryIsBlank_LU->(row1%8));
        (TB_Blanks->at2)
            = (TB_Blanks->at2) & (CheckTableEntryIsNonBlank_LU->(row2%8));
    }
];
```

§**10. Moving Blank Bits Down.** Another common table operation is to compress it by moving all the blank rows down to the bottom, so that non-blank rows occur in a contiguous block at the top: this means table sorting can be done without having to refer continually to the blanks bitmaps. The following operation is useful for keeping the blanks bitmaps up to date when blank rows are moved down.

```
[ TableMoveBlankBitsDown tab row1 row2 col at atp1 bit rx;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rtrue;
    if (((tab-->col)-->1) & TB_COLUMN_NOBLANKBITS) return;
    row1--; row2--;
    ! Read blank bit for row1:
    at = ((tab-->col)-->2) + (row1/8);
    bit = ((TB_Blanks->at) & (CheckTableEntryIsBlank_LU->(row1%8)));
    if (bit) bit = true;
    ! Loop through, setting each blank bit to the next:
    for (rx=row1:rx<row2:rx++) {
        atp1 = ((tab-->col)-->2) + ((rx+1)/8);
        at = ((tab-->col)-->2) + (rx/8);
        if ((TB_Blanks->atp1) & (CheckTableEntryIsBlank_LU->((rx+1)%8))) {
            (TB_Blanks->at)
                = (TB_Blanks->at) | (CheckTableEntryIsBlank_LU->(rx%8));
        } else {
            (TB_Blanks->at)
```

```
                = (TB_Blanks->at) & (CheckTableEntryIsNonBlank_LU->(rx%8));
        }
    }
    ! Write bit to blank bit for row2:
    at = ((tab-->col)-->2) + (row2/8);
    if (bit) {
        (TB_Blanks->at)
            = (TB_Blanks->at) | (CheckTableEntryIsBlank_LU->(row2%8));
    } else {
        (TB_Blanks->at)
            = (TB_Blanks->at) & (CheckTableEntryIsNonBlank_LU->(row2%8));
    }
];
```

§**11. Table Row Corresponding.** `TableRowCorr(T, C, V)` returns the first row on which value `V` appears in column `C` of table `T`, or prints an error if it doesn't.

`ExistsTableRowCorr(T, C, V)` returns the first row on which `V` appears in column `C` of table `T`, or 0 if `V` does not occur at all. If the column is a topic, then we match the entry as a snippet against the value as a general parsing routine.

```
[ TableRowCorr tab col lookup_value lookup_col i j f;
    if (col >= 100) col=TableFindCol(tab, col, true);
    lookup_col = tab-->col;
    j = lookup_col-->0 - COL_HSIZE;
    f=0;
    if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED) f=1;
    for (i=1:i<=j:i++) {
        if ((lookup_value == TABLE_NOVALUE) &&
            (CheckTableEntryIsBlank(tab,col,i))) continue;
        if (f) {
            if (BlkValueCompare(lookup_col-->(i+COL_HSIZE), lookup_value) == 0)
                return i;
        } else {
            if (lookup_col-->(i+COL_HSIZE) == lookup_value) return i;
        }
    }
    return RunTimeProblem(RTP_TABLE_NOCORR, tab);
];
[ ExistsTableRowCorr tab col entry i k v f kov;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rfalse;
    f=0;
    if (((tab-->col)-->1) & TB_COLUMN_TOPIC) f=1;
    else if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED) f=2;
    k = TableRows(tab);
    for (i=1:i<=k:i++) {
        v = (tab-->col)-->(i+COL_HSIZE);
        if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i))) continue;
        switch (f) {
            1: if ((v)(entry/100, entry%100) ~= GPR_FAIL) return i;
            2: if (BlkValueCompare(v, entry) == 0) return i;
            default: if (v == entry) return i;
```

```
        }
    }
    ! print "Giving up^";
    return 0;
];
```

§12. Table Look Up Corresponding Row. `TableLookUpCorr(T, C1, C2, V)` finds the first row on which value `V` appears in column `C2`, and returns the corresponding value in `C1`, or prints an error if the value `V` cannot be found or has no corresponding value in `C1`.

`ExistsTableLookUpCorr(T, C1, C2, V)` returns `true` if the operation `TableLookUpCorr(T, C1, C2, V)` can be done, `false` otherwise.

```
[ TableLookUpCorr tab col1 col2 lookup_value write_flag write_value cola1 cola2 i j v f;
    if (col1 >= 100) col1=TableFindCol(tab, col1, true);
    if (col2 >= 100) col2=TableFindCol(tab, col2, true);
    cola1 = tab-->col1;
    cola2 = tab-->col2;
    j = cola2-->0;
    f=0;
    if (((tab-->col2)-->1) & TB_COLUMN_ALLOCATED) f=1;
    if (((tab-->col2)-->1) & TB_COLUMN_TOPIC) f=2;
    for (i=1+COL_HSIZE:i<=j:i++) {
        v = cola2-->i;
        if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col2,i-COL_HSIZE))) continue;
        if (f == 1) {
            if (BlkValueCompare(v, lookup_value) ~= 0) continue;
        } else if (f == 2) {
            if ((v)(lookup_value/100, lookup_value%100) == GPR_FAIL) continue;
        } else {
            if (v ~= lookup_value) continue;
        }
        if (write_flag) {
            ForceTableEntryNonBlank(tab,col1,i-COL_HSIZE);
            switch (write_flag) {
                2: cola1-->i = cola1-->i + write_value;
                3: cola1-->i = cola1-->i - write_value;
                default: cola1-->i = write_value;
            }
            rfalse;
        }
        v = cola1-->i;
        if ((v == TABLE_NOVALUE) &&
            (CheckTableEntryIsBlank(tab,col1,i-COL_HSIZE))) continue;
        return v;
    }
    return RunTimeProblem(RTP_TABLE_NOCORR, tab);
];
[ ExistsTableLookUpCorr tab col1 col2 lookup_value cola1 cola2 i j f;
    if (col1 >= 100) col1=TableFindCol(tab, col1, false);
    if (col2 >= 100) col2=TableFindCol(tab, col2, false);
    if (col1*col2 == 0) rfalse;
    cola1 = tab-->col1; cola2 = tab-->col2;
```

```
    j = cola2-->0;
    f=0;
    if (((tab-->col2)-->1) & TB_COLUMN_ALLOCATED) f=1;
    if (((tab-->col2)-->1) & TB_COLUMN_TOPIC) f=2;
    for (i=1+COL_HSIZE:i<=j:i++) {
        if (f == 1) {
            if (BlkValueCompare(cola2-->i, lookup_value) ~= 0) continue;
        } else if (f == 2) {
            if ((cola2-->i)(lookup_value/100, lookup_value%100) == GPR_FAIL) continue;
        } else {
            if (cola2-->i ~= lookup_value) continue;
        }
        if ((cola1-->i == TABLE_NOVALUE) &&
            (CheckTableEntryIsBlank(tab,col1,i-COL_HSIZE))) continue;
        rtrue;
    }
    rfalse;
];
```

§13. Table Look Up Entry. `TableLookUpEntry(T, C, R)` returns the value at column `C`, row `R`, printing an error if that doesn't exist.

`ExistsTableLookUpEntry(T, C, R)` returns true if a value exists at column `C`, row `R`, false otherwise.

```
[ TableLookUpEntry tab col index write_flag write_value v;
    if (col >= 100) col=TableFindCol(tab, col, true);
    if ((index < 1) || (index > TableRows(tab)))
        return RunTimeProblem(RTP_TABLE_NOROW, tab, index);
    if (write_flag) {
        switch(write_flag) {
            1: ForceTableEntryNonBlank(tab,col,index);
                (tab-->col)-->(index+COL_HSIZE) = write_value;
            2: ForceTableEntryNonBlank(tab,col,index);
                (tab-->col)-->(index+COL_HSIZE) =
                    ((tab-->col)-->(index+COL_HSIZE)) + write_value;
            3: ForceTableEntryNonBlank(tab,col,index);
                (tab-->col)-->(index+COL_HSIZE) =
                    ((tab-->col)-->(index+COL_HSIZE)) - write_value;
            4: ForceTableEntryBlank(tab,col,index);
            5: ForceTableEntryNonBlank(tab,col,index);
                return ((tab-->col)-->(index+COL_HSIZE));
        }
        rfalse;
    }
    v = ((tab-->col)-->(index+COL_HSIZE));
    if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,index)))
        return RunTimeProblem(RTP_TABLE_NOENTRY, tab, col, index);
    return v;
];
[ ExistsTableLookUpEntry tab col index v;
    if (col >= 100) col=TableFindCol(tab, col);
    if (col == 0) rfalse;
    if ((index<1) || (index > TableRows(tab))) rfalse;
```

```
    v = ((tab-->col)-->(index+COL_HSIZE));
    if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,index)))
        rfalse;
    rtrue;
];
```

§**14. Blank Rows.** `TableRowIsBlank(T, R)` returns true if row `R` of table `T` is blank. (`R` must be a legal row number.)

`TableBlankOutRow(T, R)` fills row `R` of table `T` with blanks. (`R` must be a legal row number.)

`TableBlankRows(T)` returns the number of blank rows in `T`.

`TableFilledRows(T)` returns the number of non-blank rows in `T`.

`TableBlankRow(T)` finds the first blank row in `T`.

```
[ TableRowIsBlank tab j k;
    for (k=1:k<=tab-->0:k++) {
        if (((tab-->k)-->(j+COL_HSIZE)) ~= TABLE_NOVALUE) rfalse;
        if (CheckTableEntryIsBlank(tab, k, j) == false) rfalse;
    }
    rtrue;
];
[ TableBlankOutRow tab row k;
    if (tab==0) return RunTimeProblem(RTP_TABLE_NOTABLE, tab);
    for (k=1:k<=tab-->0:k++)
        ForceTableEntryBlank(tab, k, row);
];
[ TableBlankOutColumn tab col n k;
    if (tab==0) return RunTimeProblem(RTP_TABLE_NOTABLE, tab);
    n = TableRows(tab);
    for (k=1:k<=n:k++)
        ForceTableEntryBlank(tab, col, k);
];
[ TableBlankOutAll tab n k;
    if (tab==0) return RunTimeProblem(RTP_TABLE_NOTABLE, tab);
    n = TableRows(tab);
    for (k=1:k<=n:k++)
        TableBlankOutRow(tab, k);
];
[ TableBlankRows tab i j c;
    i = TableRows(tab); !print i, " rows^";
    for (j=1:j<=i:j++)
        if (TableRowIsBlank(tab, j)) c++;
    !print c, " blank^";
    return c;
];
[ TableFilledRows tab;
    return TableRows(tab) - TableBlankRows(tab);
];
[ TableBlankRow tab i j;
    i = TableRows(tab);
    for (j=1:j<=i:j++)
```

```
        if (TableRowIsBlank(tab, j)) return j;
    RunTimeProblem(RTP_TABLE_NOMOREBLANKS, tab);
    return i;
];
```

§**15. Random Row.** `TableRandomRow(T)` chooses a random non-blank row in `T`.

```
[ TableRandomRow tab i j k;
    i = TableRows(tab);
    j = TableFilledRows(tab);
    if (j==0) return RunTimeProblem(RTP_TABLE_NOROWS, tab);
    if (j>1) j = random(j);
    for (k=1:k<=i:k++) {
        if (TableRowIsBlank(tab, k) == false) j--;
        if (j==0) return k;
    }
];
```

§**16. Swap Rows.** `TableSwapRows(T, R1, R2)` exchanges rows `R1` and `R2`.

```
[ TableSwapRows tab i j k l v1 v2;
    if (i==j) return;
    l = tab-->0;
    for (k=1:k<=l:k++) {
        v1 = (tab-->k)-->(i+COL_HSIZE);
        v2 = (tab-->k)-->(j+COL_HSIZE);
        (tab-->k)-->(i+COL_HSIZE) = v2;
        (tab-->k)-->(j+COL_HSIZE) = v1;
        if ((v1 == TABLE_NOVALUE) || (v2 == TABLE_NOVALUE))
            TableSwapBlankBits(tab, i, j, k);
    }
];
```

§**17. Compare Rows.** `TableCompareRows(T, C, R1, R2, D)` returns:

(a) +1 if the entry at row R1 of column C is > the entry at row R2,
(b) 0 if they are equal, and
(c) −1 if entry at R1 < entry at R2.

When $D = +1$, a blank value is greater than all other values, so that in an ascending sort the blanks come last; when $D = -1$, a blank value is less than all others, so that once again blanks are last. Finally, a wholly blank row is always placed after a row in which the entry in C is blank but where other entries are not.

```
[ TableCompareRows tab col row1 row2 dir val1 val2 bl1 bl2 f;
    if (col >= 100) col=TableFindCol(tab, col, false);
    val1 = (tab-->col)-->(row1+COL_HSIZE);
    val2 = (tab-->col)-->(row2+COL_HSIZE);
    if (val1 == TABLE_NOVALUE) bl1 = CheckTableEntryIsBlank(tab,col,row1);
    if (val2 == TABLE_NOVALUE) bl2 = CheckTableEntryIsBlank(tab,col,row2);
    if ((val1 == val2) && (bl1 == bl2)) {
        if (val1 ~= TABLE_NOVALUE) return 0;
        if (bl1 == false) return 0;
        ! The two entries are both blank:
```

```
        if (TableRowIsBlank(tab, row1)) {
            if (TableRowIsBlank(tab, row2)) return 0;
            return -1*dir;
        }
        if (TableRowIsBlank(tab, row2)) return dir;
        return 0;
    }
    if (bl1) return dir;
    if (bl2) return -1*dir;
    f = ((tab-->col)-->1);
    if (f & TB_COLUMN_ALLOCATED) {
        if (BlkValueCompare(val2, val1) < 0) return 1;
        return -1;
    } else if (f & TB_COLUMN_SIGNED) {
        if (val1 > val2) return 1;
        return -1;
    } else {
        if (UnsignedCompare(val1, val2) > 0) return 1;
        return -1;
    }
];
```

§18. Move Row Down.

```
[ TableMoveRowDown tab r1 r2 rx k l m v f;
    if (r1==r2) return;
    l = tab-->0;
    for (k=1:k<=l:k++) {
        f = false;
        m = (tab-->k)-->(r1+COL_HSIZE);
        if (m == TABLE_NOVALUE) f = true;
        for (rx=r1:rx<r2:rx++) {
            v = (tab-->k)-->(rx+COL_HSIZE+1);
            (tab-->k)-->(rx+COL_HSIZE) = v;
            if (v == TABLE_NOVALUE) f = true;
        }
        (tab-->k)-->(r2+COL_HSIZE) = m;
        if (f) TableMoveBlankBitsDown(tab, r1, r2, k);
    }
];
```

§19. Shuffle.

`TableShuffle(T)` sorts `T` into random row order.

```
[ TableShuffle tab i to;
    TableMoveBlanksToBack(tab, 1, TableRows(tab));
    to = TableFilledRows(tab);
    for (i=2:i<=to:i++) TableSwapRows(tab, i, random(i));
];
```

§20. Next Row. `TableNextRow(T, C, R, D)` is used when scanning through a table in order of the values in column `C`: ascending order if `D = 1`, descending if `D = -1`. The current position is row `R` of column `C`, or `R = 0` if we have not yet found the first row. The return value is the row number for the next value, or 0 if we are already at the final value. Note that if there are several equal values in the column, they will be run through in turn, in order of their physical row numbers - ascending if `D = 1`, descending if `D = -1`, so that using the routine with `D = -1` always produces the exact reverse ordering from using it with `D = 1` and the same parameters. Rows with blank entries in `C` are skipped.

```
for (R=TableNextRow(T,C,0,D): R : R=TableNextRow(T,C,R,D)) ...
```

will perform a loop of valid row numbers in order of column `C`.

```
[ TableNextRow tab col row dir i k val v dv min_dv min_at signed_arithmetic f;
    if (col >= 100) col=TableFindCol(tab, col, false);
    f = ((tab-->col)-->1);
    if (f & TB_COLUMN_ALLOCATED) RunTimeProblem(RTP_TABLE_CANTRUNTHROUGH, tab);
    signed_arithmetic = f & TB_COLUMN_SIGNED;
    #Iftrue (WORDSIZE == 2);
    if (row == 0) {
        if (signed_arithmetic) {
            if (dir == 1) val = $8000; else val = $7fff;
        } else {
            if (dir == 1) val = 0; else val = $ffff;
        }
    } else val = (tab-->col)-->(row+COL_HSIZE);
    if (signed_arithmetic) min_dv = $7fff; else min_dv = $ffff;
    #ifnot; ! WORDSIZE == 4
    if (row == 0) {
        if (signed_arithmetic) {
            if (dir == 1) val = $80000000; else val = $7fffffff;
        } else {
            if (dir == 1) val = 0; else val = $ffffffff;
        }
    } else val = (tab-->col)-->(row+COL_HSIZE);
    if (signed_arithmetic) min_dv = $7fffffff; else min_dv = $ffffffff;
    #endif;
    k = TableRows(tab);
    if (dir == 1) {
        for (i=1:i<=k:i++) {
            v = (tab-->col)-->(i+COL_HSIZE);
            if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i)))
                continue;
            dv = dir*v;
            if (signed_arithmetic)
            f = (((dv > dir*val) || ((v == val) && (i>row))) &&
                (dv < min_dv));
            else
            f = (((UnsignedCompare(dv, dir*val) > 0) || ((v == val) && (i>row))) &&
                (UnsignedCompare(dv, min_dv) < 0));
            if (f) { min_dv = dv; min_at = i; }
        }
    } else {
        for (i=k:i>=1:i--) {
            v = (tab-->col)-->(i+COL_HSIZE);
            if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i)))
```

```
                continue;
            dv = dir*v;
            if (signed_arithmetic)
            f = (((dv > dir*val) || ((v == val) && (i<row))) &&
                (dv < min_dv));
            else
            f = (((UnsignedCompare(dv, dir*val) > 0) || ((v == val) && (i<row))) &&
                (UnsignedCompare(dv, min_dv) < 0));
            if (f) { min_dv = dv; min_at = i; }
        }
    }
    return min_at;
];
```

§**21. Move Blanks to Back.**

```
[ TableMoveBlanksToBack tab fromrow torow i fbl lnbl blc;
    if (torow < fromrow) return;
    fbl = 0; lnbl = 0;
    for (i=fromrow: i<=torow: i++)
        if (TableRowIsBlank(tab, i)) {
            if (fbl == 0) fbl = i;
            blc++;
        } else {
            lnbl = i;
        }
    if ((fbl>0) && (lnbl>0) && (fbl < lnbl)) {
        TableMoveRowDown(tab, fbl, lnbl); ! Move first blank just past last nonblank
        TableMoveBlanksToBack(tab, fbl, lnbl-1);
    }
    return torow-blc; ! Final non-blank row
];
```

§**22. Sort.** This is really only a front-end: it calls the sorting code at "Sort.i6t".

```
[ TableSort tab col dir test_flag algorithm i j k f;
    for (i=1:i<=tab-->0:i++) {
        j = tab-->i; ! Address of column table
        if ((j-->1) & TB_COLUMN_DONTSORTME)
            return RunTimeProblem(RTP_TABLE_CANTSORT, tab);
    }
    if (col >= 100) col=TableFindCol(tab, col, false);
    k = TableRows(tab);
    k = TableMoveBlanksToBack(tab, 1, k);
    if (test_flag) {
        print "After moving blanks to back:^"; TableColumnDebug(tab, col);
    }
    SetSortDomain(TableSwapRows, TableCompareRows);
    SortArray(tab, col, dir, k, test_flag, algorithm);
    if (test_flag) {
        print "Final state:^"; TableColumnDebug(tab, col);
    }
];
```

§**23. Print Table Name.** NI fills this in: it's used to say the "table" kind of value.

```
[ PrintTableName T;
    switch(T) {
{-call:Data::Tables::compile_print_table_names}
        default: print "** No such table **";
    }
];
```

§**24. Print Table to File.** This is how we serialise a table to an external file, though the writing is done by printing characters in the standard way; it's just that the output stream will be an external file rather than the screen when this routine is called.

```
[ TablePrint tab i j k row col v tc kov;
    for (i=1:i<=tab-->0:i++) {
        j = tab-->i; ! Address of column table
        if (((j-->1) & TB_COLUMN_CANEXCHANGE) == 0)
            rtrue;
    }
    k = TableRows(tab);
    k = TableMoveBlanksToBack(tab, 1, k);
    print "! ", (PrintTableName) tab, " (", k, ")^";
    for (row=1:row<=k:row++) {
        for (col=1:col<=tab-->0:col++) {
            tc = ((tab-->col)-->1) & TB_COLUMN_NUMBER;
            kov = KindAtomic(TC_KOV(tc));
            if (kov == UNKNOWN_TY) kov = NUMBER_TY;
            v = (tab-->col)-->(row+COL_HSIZE);
            if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,row)))
                print "-- ";
            else {
                if (BlkValueWriteToFile(v, kov) == false) print v;
                print " ";
            }
        }
        print "^";
    }
    rfalse;
];
```

§**25. Read Table from File.** And this is how we unserialise again. It makes sense only on Glulx.

```
#ifdef TARGET_GLULX;
[ TableRead tab auxf row maxrow col ch v sgn dg j tc kov;
    for (col=1:col<=tab-->0:col++) {
        j = tab-->col; ! Address of column table
        if (((j-->1) & TB_COLUMN_CANEXCHANGE) == 0)
            return RunTimeProblem(RTP_TABLE_CANTSAVE, tab);
    }
    maxrow = TableRows(tab);
    !print maxrow, " rows available.^";
    for (row=1: row<=maxrow: row++) {
        TableBlankOutRow(tab, row);
    }
    for (row=1: row<=maxrow: row++) {
        !print "Reading row ", row, "^";
        ch = FileIO_GetC(auxf);
        if (ch == '!') {
            while (ch ~= -1 or 10 or 13) ch = FileIO_GetC(auxf);
            while (ch == 10 or 13) ch = FileIO_GetC(auxf);
        }
        for (col=1: col<=tab-->0: col++) {
            if (ch == -1) { row++; jump NoMore; }
            if (ch == 10 or 13) break;
            tc = ((tab-->col)-->1) & TB_COLUMN_NUMBER;
            kov = KindAtomic(TC_KOV(tc));
            if (kov == UNKNOWN_TY) kov = NUMBER_TY;
            !print "tc = ", tc, " kov = ", kov, "^";
            sgn = 1;
            if (ch == '-') {
                ch = FileIO_GetC(auxf);
                if (ch == -1) jump NotTable;
                if (ch == '-') { ch = FileIO_GetC(auxf); jump EntryDone; }
                sgn = -1;
            }
            if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED)
                ForceTableEntryNonBlank(tab, col, row);
            !print "A";
            v = BlkValueReadFromFile(0, 0, -1, kov);
            if (v) {
                if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED)
                    v = BlkValueReadFromFile(TableLookUpEntry(tab, col, row),
                        auxf, ch, kov);
                else
                    v = BlkValueReadFromFile(0, auxf, ch, kov);
                ch = 32;
            } else {
                dg = ch - '0';
                if ((dg < 0) || (dg > 9)) jump NotTable;
                v = dg;
                for (::) {
                    ch = FileIO_GetC(auxf);
                    dg = ch - '0';
```

```
                    if ((dg < 0) || (dg > 9)) break;
                    v = 10*v + dg;
                }
                v = v*sgn;
            }
            !print "v=", v, " ";
            if (((tab-->col)-->1) & TB_COLUMN_ALLOCATED == 0)
                TableLookUpEntry(tab, col, row, true, v);
            .EntryDone;
            !print "First nd is ", ch, "^";
            while (ch == 9 or 32) ch = FileIO_GetC(auxf);
        }
        while (ch ~= -1 or 10 or 13) {
            if ((ch ~= '-') && (((ch-'0')<0) || ((ch-'0')>9))) jump NotTable;
            if (ch ~= 9 or 32) jump WontFit;
            ch = FileIO_GetC(auxf);
        }
    }
    .NoMore;
    while (ch == 9 or 32 or 10 or 13) ch = FileIO_GetC(auxf);
    if (ch == -1) return;
    .WontFit;
    return RunTimeProblem(RTP_TABLE_WONTFIT, tab);
    .NotTable;
    return RunTimeProblem(RTP_TABLE_BADFILE, tab);
];
#ENDIF; ! TARGET_GLULX
```

§**26. Print Rank.** The table of scoring ranks is a residue from the ancient times of early IF: it gets a tiny amount of special treatment here, even though I7 works tend not to use these now dated conventions.

```
[ PrintRank i j v;
#ifdef RANKING_TABLE;
    L__M(##Score, 3);
    j = TableRows(RANKING_TABLE);
    for (i=j:i>=1:i--)
        if (score >= TableLookUpEntry(RANKING_TABLE, 1, i)) {
            v = TableLookUpEntry(RANKING_TABLE, 2, i);
            if (v ofclass String) print (string) v;
            else v();
            ".";
        }
#endif;
    ".";
];
```

§**27. Debugging.** A routine to print the state of a table, for debugging purposes only.

```
[ TableColumnDebug tab col k i v;
    if (col >= 100) col=TableFindCol(tab, col, false);
    k = TableRows(tab);
    print "Table col ", col, ": ";
    for (i=1:i<=k:i++) {
        v = (tab-->col)-->(i+COL_HSIZE);
        if ((v == TABLE_NOVALUE) && (CheckTableEntryIsBlank(tab,col,i)))
            print "BLANK ";
        else
            print v, " ";
    }
    print "*^";
];
```