StoredAction Template B/stact

Purpose

Code to support the stored action kind of value.

§**1. Head.** As ever: if there is no heap, there are no stored actions.

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

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

```
[ STORED_ACTION_TY_Support task arg1 arg2 arg3;
    switch(task) {
        CREATE_KOVS:     return STORED_ACTION_TY_Create();
        CAST_KOVS:       rfalse;
        DESTROY_KOVS:    return STORED_ACTION_TY_Destroy(arg1);
        PRECOPY_KOVS:    rfalse;
        COPY_KOVS:       return STORED_ACTION_TY_Copy(arg1, arg2);
        COMPARE_KOVS:    return STORED_ACTION_TY_Compare(arg1, arg2);
        READ_FILE_KOVS:  rfalse;
        WRITE_FILE_KOVS: rfalse;
        HASH_KOVS:       return STORED_ACTION_TY_Hash(arg1);
    }
];
```

§**3. Creation.** A stored action block has fixed size, so this is a single-block KOV: its data consists of six words, laid out as shown in the following routine. Note that it initialises to the default value for this KOV, an action in which the player waits.

An action which involves a topic – such as the one produced by the command LOOK UP JIM MCDIVITT IN ENCYCLOPAEDIA – cannot be tried without the text of that topic (JIM MCDIVITT) being available. That's no problem if the action is tried in the same turn in which it is generated, because the text will still be in the command buffer. But once we store actions up for future use it becomes an issue. So when we store an action involving a topic, we record the actual text typed at the time when it is stored, and this goes into array entry 5 of the block. Because that in turn is indexed text, and therefore a block value on the heap in its own right, we have to be a little more careful about destroying and copying stored actions than we otherwise would be.

Note that entries 1 and 2 are values whose kind depends on the action in entry 0: but they are never block values, because actions are not allowed to apply to block values. This simplifies matters considerably.

```
[ STORED_ACTION_TY_Create stora;
    stora = BlkAllocate(6*WORDSIZE, STORED_ACTION_TY, BLK_FLAG_WORD);
    BlkValueWrite(stora, 0, ##Wait); ! action
    BlkValueWrite(stora, 1, 0); ! noun
    BlkValueWrite(stora, 2, 0); ! second
    BlkValueWrite(stora, 3, player); ! actor
    BlkValueWrite(stora, 4, false); ! whether a request
    BlkValueWrite(stora, 5, 0); ! indexed text of command if necessary, 0 if not
    return stora;
];
```

§**4. Setting Up.** In practice it's convenient for NI to have a routine which creates a stored action with a given slate of action variables, rather than have to set them all one at a time, so the following is provided as a shorthand form.

```
[ STORED_ACTION_TY_New a n s ac req  stora;
    if (stora == 0) stora = STORED_ACTION_TY_Create();
    BlkValueWrite(stora, 0, a);
    BlkValueWrite(stora, 1, n);
    BlkValueWrite(stora, 2, s);
    BlkValueWrite(stora, 3, ac);
    BlkValueWrite(stora, 4, req);
    BlkValueWrite(stora, 5, 0);
    return stora;
];
```

§**5. Destruction.** Entries 0 to 4 are forgettable non-block values: only the optional indexed text requires destruction.

```
[ STORED_ACTION_TY_Destroy stora toc;
    toc = BlkValueRead(stora, 5);
    if (toc) BlkFree(toc);
];
```

§**6. Copying.** The only entry needing attention is, again, entry 5: if this is non-zero in the source, then we need to create a new indexed text block to hold a duplicate copy of the text.

```
[ STORED_ACTION_TY_Copy storato storafrom tocfrom tocto;
    tocfrom = BlkValueRead(storafrom, 5);
    if (tocfrom == 0) return;
    tocto = INDEXED_TEXT_TY_Support(CREATE_KOVS);
    BlkValueCopy(tocto, tocfrom);
    BlkValueWrite(storato, 5, tocto);
];
```

§**7. Comparison.** There is no very convincing ordering on stored actions, but we need to devise a comparison which will exhaustively determine whether two actions are or are not different.

```
[ STORED_ACTION_TY_Compare storaleft storaright delta itleft itright;
    delta = BlkValueRead(storaleft, 0) - BlkValueRead(storaright, 0);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 1) - BlkValueRead(storaright, 1);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 2) - BlkValueRead(storaright, 2);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 3) - BlkValueRead(storaright, 3);
    if (delta) return delta;
    delta = BlkValueRead(storaleft, 4) - BlkValueRead(storaright, 4);
    if (delta) return delta;
    itleft = BlkValueRead(storaleft, 5);
    itright = BlkValueRead(storaright, 5);
    if ((itleft ~= 0) && (itright ~= 0))
        return INDEXED_TEXT_TY_Support(COMPARE_KOVS, itleft, itright);
    return itleft - itright;
];
[ STORED_ACTION_TY_Distinguish stora1 stora2;
    if (STORED_ACTION_TY_Compare(stora1, stora2) == 0) rfalse;
    rtrue;
];
```

§**8. Hashing.**

```
[ STORED_ACTION_TY_Hash stora  rv it;
    rv = BlkValueRead(stora, 0);
    rv = rv * 33 + BlkValueRead(stora, 1);
    rv = rv * 33 + BlkValueRead(stora, 2);
    rv = rv * 33 + BlkValueRead(stora, 3);
    rv = rv * 33 + BlkValueRead(stora, 4);
    it = BlkValueRead(stora, 5);
    if (it ~= 0)
        rv = rv * 33 + INDEXED_TEXT_TY_Support(HASH_KOVS, it);
    return rv;
];
```

§**9. Printing.** We share some code here with the routines originally written for the ACTIONS testing command. (The `DB` in `DB_Action` stands for "debugging".) When printing a topic, it prints the relevant words from the player's command: so if our stored action is one which contains an entry 5, then we have to temporarily adopt this as the player's command, and restore the old player's command once printing is done. To do this, we need to save the old player's command, and we do that by creating an indexed text for the duration.

```
[ STORED_ACTION_TY_Say stora text_of_command saved_command saved_pn saved_action K1 K2 at;
    if ((stora==0) || (BlkType(stora) ~= STORED_ACTION_TY)) return;
    text_of_command = BlkValueRead(stora, 5);
    if (text_of_command) {
        saved_command = INDEXED_TEXT_TY_Support(CREATE_KOVS);
        INDEXED_TEXT_TY_Support(CAST_KOVS, players_command, SNIPPET_TY, saved_command);
        SetPlayersCommand(text_of_command);
    }
    saved_pn = parsed_number; saved_action = action;
    action = BlkValueRead(stora, 0);
    at = FindAction(-1);
    K1 = ActionData-->(at+AD_NOUN_KOV);
    K2 = ActionData-->(at+AD_SECOND_KOV);
    if (K1 ~= OBJECT_TY) {
        parsed_number = BlkValueRead(stora, 1);
        if ((K1 == UNDERSTANDING_TY) && (text_of_command == 0)) {
            if (saved_command == 0) saved_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(players_command, SNIPPET_TY, saved_command);
            text_of_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(parsed_number, TEXT_TY, text_of_command);
            SetPlayersCommand(text_of_command);
            parsed_number = players_command;
        }
    }
    if (K2 ~= OBJECT_TY) {
        parsed_number = BlkValueRead(stora, 2);
        if ((K2 == UNDERSTANDING_TY) && (text_of_command == 0)) {
            if (saved_command == 0) saved_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(players_command, SNIPPET_TY, saved_command);
            text_of_command = INDEXED_TEXT_TY_Create();
            INDEXED_TEXT_TY_Cast(parsed_number, TEXT_TY, text_of_command);
            SetPlayersCommand(text_of_command);
            parsed_number = players_command;
        }
    }
    DB_Action(BlkValueRead(stora, 3), BlkValueRead(stora, 4), BlkValueRead(stora, 0),
        BlkValueRead(stora, 1), BlkValueRead(stora, 2), true);
    parsed_number = saved_pn; action = saved_action;
    if (text_of_command) {
        SetPlayersCommand(saved_command);
        BlkFree(saved_command);
    }
];
```

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

An action "involves" an object if it appears as either the actor or the first or second noun.

```
[ STORED_ACTION_TY_Involves stora item at;
    at = FindAction(BlkValueRead(stora, 0));
    if (at) {
        if ((ActionData-->(at+AD_NOUN_KOV) == OBJECT_TY) &&
            (BlkValueRead(stora, 1) == item)) rtrue;
        if ((ActionData-->(at+AD_SECOND_KOV) == OBJECT_TY) &&
            (BlkValueRead(stora, 2) == item)) rtrue;
    }
    if (BlkValueRead(stora, 3) == item) rtrue;
    rfalse;
];
```

§**11. Nouns.** Extracting the noun or second noun from an action is a delicate business because simply returning the values in entries 1 and 2 would not be type-safe; it would fail to be an object if the stored action did not apply to objects. So the following returns `nothing` if requested to produce noun or second noun for such an action.

```
[ STORED_ACTION_TY_Part stora ind at ado;
    if (ind == 1 or 2) {
        if (ind == 1) ado = AD_NOUN_KOV; else ado = AD_SECOND_KOV;
        at = FindAction(BlkValueRead(stora, 0));
        if ((at) && (ActionData-->(at+ado) == OBJECT_TY)) return BlkValueRead(stora, ind);
        return nothing;
    }
    return BlkValueRead(stora, ind);
];
```

§**12. Pattern Matching.** In order to apply an action pattern such as "doing something with the kazoo" to a stored action, it needs to be the current action, because the code which compiles conditions like this looks at the `action`, `noun`, ..., variables. We don't want to do anything as disruptive as temporarily starting the stored action and then halting it again, so instead we simply "adopt" it, saving the slate of action variables and setting them from the stored action: almost immediately after – the moment the condition has been tested – we "unadopt" it again, restoring the stored values. Since the action pattern cannot itself refer to a stored action, the following code won't be nested, and we don't need to worry about stacking up saved copies of the action variables.

`SAT_Tmp-->0` stores the outcome of the condition, and is set in code compiled by NI.

```
Array SAT_Tmp-->7;
[ STORED_ACTION_TY_Adopt stora at;
    SAT_Tmp-->1 = action;
    SAT_Tmp-->2 = noun;
    SAT_Tmp-->3 = second;
    SAT_Tmp-->4 = actor;
    SAT_Tmp-->5 = act_requester;
    SAT_Tmp-->6 = parsed_number;
    action = BlkValueRead(stora, 0);
    at = FindAction(-1);
    if (ActionData-->(at+AD_NOUN_KOV) == OBJECT_TY) noun = BlkValueRead(stora, 1);
```

```
    else { parsed_number = BlkValueRead(stora, 1); noun = nothing; }
    if (ActionData-->(at+AD_SECOND_KOV) == OBJECT_TY) second = BlkValueRead(stora, 2);
    else { parsed_number = BlkValueRead(stora, 2); second = nothing; }
    actor = BlkValueRead(stora, 3);
    if (BlkValueRead(stora, 4)) act_requester = player; else act_requester = nothing;
];
[ STORED_ACTION_TY_Unadopt;
    action = SAT_Tmp-->1;
    noun = SAT_Tmp-->2;
    second = SAT_Tmp-->3;
    actor = SAT_Tmp-->4;
    act_requester = SAT_Tmp-->5;
    parsed_number = SAT_Tmp-->6;
    return SAT_Tmp-->0;
];
```

§**13. Current Action.** Although we never cast other values to stored actions, because none of them really imply an action (not even an action name, since that gives no help as to what the nouns might be), there is of course one action almost always present within a story file at run-time, even if it is not a single value as such: the action which is currently running. The following routine translates that into a stored action – thus allowing us to store it.

This is the place where we look to see if the action applies to a topic as either its noun or second noun, and if it does, we copy the player's command into an indexed text block-value in entry 5.

```
[ STORED_ACTION_TY_Current stora at text_of_command;
    if ((stora==0) || (BlkType(stora) ~= STORED_ACTION_TY)) return 0;
    BlkValueWrite(stora, 0, action);
    at = FindAction(-1);
    if (ActionData-->(at+AD_NOUN_KOV) == OBJECT_TY) BlkValueWrite(stora, 1, noun);
    else BlkValueWrite(stora, 1, parsed_number);
    if (ActionData-->(at+AD_SECOND_KOV) == OBJECT_TY) BlkValueWrite(stora, 2, second);
    else BlkValueWrite(stora, 2, parsed_number);
    BlkValueWrite(stora, 3, actor);
    if (act_requester) BlkValueWrite(stora, 4, true); else BlkValueWrite(stora, 4, false);
    if ((at) && ((ActionData-->(at+AD_NOUN_KOV) == UNDERSTANDING_TY) ||
            (ActionData-->(at+AD_SECOND_KOV) == UNDERSTANDING_TY))) {
        text_of_command = BlkValueRead(stora, 5);
        if (text_of_command == 0) {
            text_of_command = INDEXED_TEXT_TY_Support(CREATE_KOVS);
            BlkValueWrite(stora, 5, text_of_command);
        }
        INDEXED_TEXT_TY_Support(CAST_KOVS, players_command, SNIPPET_TY, text_of_command);
    } else BlkValueWrite(stora, 5, 0);
    return stora;
];
```

§**14. Trying.** Finally: having stored an action for perhaps many turns, we now let it happen, either silently or not.

```
[ STORED_ACTION_TY_Try stora ks  text_of_command saved_command;
    if ((stora==0) || (BlkType(stora) ~= STORED_ACTION_TY)) return;
    if (ks) { @push keep_silent; keep_silent=1; }
    text_of_command = BlkValueRead(stora, 5);
    if (text_of_command) {
        saved_command = INDEXED_TEXT_TY_Support(CREATE_KOVS);
        INDEXED_TEXT_TY_Support(CAST_KOVS, players_command, SNIPPET_TY, saved_command);
        SetPlayersCommand(text_of_command);
    }
    TryAction(BlkValueRead(stora, 4), BlkValueRead(stora, 3),
        BlkValueRead(stora, 0), BlkValueRead(stora, 1), BlkValueRead(stora, 2));
    if (text_of_command) {
        SetPlayersCommand(saved_command);
        BlkFree(saved_command);
    }
    if (ks) { @pull keep_silent; }
];
```

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

```
#IFNOT; ! IFDEF MEMORY_HEAP_SIZE
[ STORED_ACTION_TY_Say stora; ];
[ STORED_ACTION_TY_New a n s ac req stora; return false; ];
[ STORED_ACTION_TY_Support t a b c; rfalse; ];
#ENDIF; ! IFDEF MEMORY_HEAP_SIZE
```