XEROX

BUSINESS SYSTEMS

Systems Development Division

August 14, 1978

To: Advanced Design and User Prototypes Group

C: Larry Clark

From: Dave Smith, Bob Shur / SD at Palo Alto

Subject: Recording and playing back selections in Stored Command Sequences

Filed on: <davesmith>star>macro-selections.bravo

This memo addresses some issues concerning the representation of selections in Star's stored command sequences. It deals with Star and the real implementation, not necessarily the Desktop prototype. (How the prototype differs from this theory, if at all, is undecided.) These issues may continue to evolve as we gain more experience with a working system, real or prototype.

The following topics are discussed: issues involved in recording selections, the syntactic form of a recorded selection, the identification of Star objects, defining variables, the pattern-matching algorithm for using variables, recordable objects and actions, and playback.

Recording selections in Star

It is clear that there is little difficulty in recording or playing back command operators. We simply record the text on a key cap, e.g. "MOVE", or in a menu, e.g. "CLOSE". On playback, we have a case statement which dispatches on the command string.

For operands to commands, however, the situation is more interesting. There are two questions:

> What is an adequate representation for a selection?
>
> How can we generalize the selection representation to include variables?

A comparison with Bravo illustrates these questions. Bravo has very few objects, mainly windows and characters. Star, on the other hand, has many kinds of objects (printers, file drawers, mail boxes, calculators, ...), many kinds of text containers (documents, fields, text scrolls), and many kinds of primitive objects (characters, frames, graphic symbols, tables, ...). Star actions can deal with any of these objects and be recorded doing so. Therefore we need a flexible representation that can encompass Star's richness.

Bravo always starts a macro execution (replay) at the Alto executive level. It requires that the playback state be *identical* to the define state. Bravo records *exactly* what is done at define time, e.g. *exactly which* characters in *which* windows are selected. This means that Bravo can never execute a recorded typescript on anything but the original document. (If you try, you'll be sorry! However Bravo can run short startup and quit macros on different documents.) In Star we attempt to ease this restriction by a powerful representation and by permitting users to define "variables" at record time and to "bind" these variables to different values at playback time. The most interesting and challenging problem for stored command sequences is how to use these variables. When the user makes a selection that is "related" to a previously defined variable, what do we record?

After conversations with most of ADUPG, we have designed solutions to these problems. (These solutions will likely undergo refinement, but things seems to be progressing.) The first solution is a complete syntactic description of recorded selections. The description incorporates the needed power. The second solution is a pattern matching algorithm describing how and when to generalize the parts of a selection.

One thing to note is that while a selection representation may be quite long, it is a simple structure made up of simple parts, and therefore is amenable to manipulation by a machine.

Syntax of recorded selections

The overall representation of any selection in Star can be described in a single line:

```
SelectionRepresentation  ::= "[ "  VariablePart*  ConstantPart*  "] "
```

Here "*" means "any number of", but at least one variable or constant part must be present. Also the parts of a selection are separated by commas. The detailed description of the parts follows.

```
VariablePart             ::= ObjectType  Variable

ConstantPart             ::= ObjectType  ObjectIdentification

ObjectType               ::= FunctionIcon
                         |   "Folder"
                         |   "Document"
                         |   "Record File"
                         |   "Frame"
                         |   "Footnote"
                         |   "Field"
                         |   "Table"
                         |   "Row"
                         |   "Column"
                         |   GraphicElement
                         |   "Text Scroll"
                         |   "Record"
                         |   "Text"
```

```
                        |  "Caret"
                        |  "X-Y"
                        |  "Sheet"
                        |  "Option"

FunctionIcon            ::= "Directory"
                        |  "File Drawer"
                        |  "Printer"
                        |  "Calculator"
                        |  ...

GraphicElement          ::= "Point"
                        |  "Symbol"
                        |  "Group"
                        |  "Multiple"
                        |  "Guiding"
                        |  "Pinned"

Variable                ::= "*"  Number

ObjectIdentification
    ::=  Name  <Pilot number>                                          (icon)
    |    Name                        (document-unique)                 (field)
    |    Number                      (document-unique)                 (frame, footnote)
    |    Number                      (frame-unique)                    (table, graphic element,
                                                                       text scroll)
    |    Number                      (table-relative)                  (row, column)
    |    CharNumber ":" CharNumber   (document/scroll-relative)        (text)
    |    CharNumber                  (document/scroll-relative)        (text, caret)
    |    Integer  Integer            (desktop/frame-relative)          (x-y)
    |    Name                        (of property/option sheet)        (sheet)
    |    Name  <value>               (of property/option)              (option)
    |    Name                        (record file-unique)              (record)
    |    Name  Number                (record file-unique)              (record element)
    |    GraphicList                                                   (graphic group,
                                                                       multiple selection)

Name                    ::= <arbitrary string of characters, possibly empty>

Number                  ::= <positive integer starting with 1>

Integer                 ::= <positive or negative integer>

CharNumber              ::= Number
                        |  "0"
                        |  "last"
```

```
GraphicList          ::= "(" GraphicSymbol {"," GraphicSymbol}* ")"

GraphicSymbol        ::= GraphicElement Variable
                      |  GraphicElement ObjectIdentification
```

"Field" above includes two kinds of fields, form fields and form letter sequences, which are not distinguished in command sequences. "Text" includes formatting characters, words, sentences, paragraphs and frames. "Row" and "Column" can occur in either order in table identifications. X-Y points can be negative, indicating a place to the left or above a frame origin; they are in units of screen dots.

Examples of recorded selections:

```
[File Drawer "Dave Smith" #]                # represents a 64-bit Pilot number
[Printer "Daisy" #]
[Folder "Purchase Orders" #]
[Record File "Help Data Base" #]
[Document "My Memo" #]
[Document "My Memo" #, Frame 1]
[Document "My Memo" #, Footnote 3]
[Document "My Memo" #, Field "subject"]
[Document "My Memo" #, Frame 1, Table 1]
[Document "My Memo" #, Frame 1, Table 1, Row 5]
[Document "My Memo" #, Frame 1, Table 1, Column 3]
[Document "My Memo" #, Frame 1, Table 1, Row 5, Column 3]
[Document "My Memo" #, Frame 1, Table 1, Column 3, Row 5]
[Document "My Memo" #, Frame 1, Symbol 14, Guiding 1]
[Document "My Memo" #, Frame 1, Symbol 14, Guiding 1, Pinned 3]
[Document "My Memo" #, Frame 1, Group 15, Guiding (Symbol 6, Point 1),
    Pinned (Symbol 6, Point 2)]
[Document "My Memo" #, Frame 1, Multiple (Symbol 3, Symbol 7, Group 15),
    Guiding (Symbol 3, Point 3)]
[Document "My Memo" #, Frame 1, Scroll 4]
[Document "My Memo" #, Frame 1, Scroll 4, 1:15]
[Document "My Memo" #, Text 23:45]
[Document "My Memo" #, Text 1:last]
[Document "My Memo" #, Caret 23]
[Document "My Memo" #, Caret last]
[Document "My Memo" #, Frame 1, X-Y 100 -25]
[X-Y 100 200]
[Record File "Parts Inventory" #, Record "Part Name" 1]
```

Note that the object selected is *implicitly* described by this representation. However the representation is unambiguous and determinable. For example, the fact that a Pinned point is included in a graphic selection means that a control point is selected; a Guiding point without a Pinned point means that a whole graphic symbol is selected.

Star object identification

It has become clear that every selectable object in Star, with the exception of text characters, needs a permanent, invariable identification. This was briefly addressed in version 4 of the Functional Spec., but many objects were left out. The above list is more complete, though it may still have one or two omissions. What many of these identifications should be has crystallized only after working with the Desktop prototype and thinking about scenarios.

Each object identification can be thought of as either "absolute" or "relative", depending on whether or not it needs other object identifications to be a complete description. All inter-document objects (icons and only icons) have "absolute" identifications; i.e. they can stand alone. Examples:

[File Drawer "Dave Smith" #]	# represents a 64-bit Pilot number
[Printer "Daisy" #]	
[Folder "Purchase Orders" #]	
[Record File "Help Data Base" #]	
[Document "My Memo" #]	

These identifications are unique. No more description need be given in order to unambiguously resolve them. On the other hand, all intra-document objects have "relative" identifications; they need additional information in order to be unambiguously resolved. Examples:

[Document "My Memo" #, Frame 1]
[Document "My Memo" #, Footnote 3]
[Document "My Memo" #, Field "subject"]
[Document "My Memo" #, Frame 1, Table 1]
[Document "My Memo" #, Frame 1, Symbol 14, Guiding 1]
[Document "My Memo" #, Frame 1, Scroll 4]
[Document "My Memo" #, Text 23:45]
[Document "My Memo" #, Caret last]
[Record File "Parts Inventory" #, Record "Part Name" 1]

The question arises as to the extent to which users will have to know about and deal with these identifications. I think stored command sequences will be successful only if the answer is "almost nothing". The proper approach is that all of these identifications are system generated and stored; users will not understand them or deal with them. (It is simply not practical to expect them to be able to change "Symbol 14" to "Symbol 23"; it's unlikely any of us could!) The only exception is that the names associated with icons, fields and records are user-generated and therefore are user-changable.

Each identification requires an entry in the object's internal data structure. For intra-document objects, a single word (16 bits) should be enough. For icons, the Pilot number is used. In every case, with the exception of text, *all object identifications have the following characteristics:*

- An identifying number is *permanently associated with an object for the life of the object in its container.* For example, if a transfer symbol is moved to a new frame, it receives a new number; otherwise it keeps its old number forever.

• Identifying numbers always *increase monotonically.* They never decrease. They are never reused, even if objects are deleted. They start with 1.

(While Pilot numbers are described as being used for icons, they may not meet the command sequence goals. It may be necessary to use an alternative number assigned by the Star applications software. Personally, I think that the real way to identify icons is by text icons.)

Admittedly the goals for object identifications are not fully developed. It is likely that they will develop only as we gain experience with the system, particularly with the Desktop prototype. However the *structure* of each identification is known, we believe: some objects have user-assigned names (e.g. fields) and some have internally-assigned numbers (e.g. frames). What is not fully understood is the precise *content* of these identifications, e.g. whether document-unique or frame-unique numbers are better. But any interpretation will "work" in the sense that the system will be able to operate; we need to evolve to the most natural assignments for the user. Thus we believe enough is known to permit us to proceed with the implementation of command sequences, both in the Desktop prototype and in the actual Star system.

Some known goals for selection representations are:

• to be able to execute macros on different Desktops and different machines, so that Star users can exchange programs

• to not require that the run-time environment be *identical* to the define-time environment; in particular, a document at run-time should need be only *similar* to the document used at define-time, for some definition of "similar"

• to permit variables to be defined and used

• to not require icons to be open at run-time in order to be used (e.g. the Calculator)

• to not require windows to be opened in a particular order; to run regardless of how many windows are open

• to be able to manipulate fields and frames regardless of how many characters are in a document

• to be able to manipulate graphic symbols and constructions regardless of how many elements are in a frame; in general to be able to manipulate an object regardless of how many objects of the same or different types are in its container (e.g. frame, folder)

• to be able to insert at the beginning or end of a document, field or text scroll regardless of how many characters are in it

Creating variables

First we will give a brief review of the theory of variables in command sequences. Then we will present an algorithm for using variables to generalize selections.

A variable is created whenever one of the following four commands is executed:

```
*1 = PAUSE "prompt"
```

The definer can explicitly tell the system to stop and accept an object at run time. For example,

```
*1 = PAUSE "Please select an address list to use"
```

Each time the macro is run, the user (the macro runner) can designate different objects to use. This is the primary way of introducing variables into macros.

```
*1 = COPY [selection] TO [destination]
```

Whenever a COPY is done, the result is a new object. Since objects created in this way do not exist at define time, some way must be provided to deal with them. The solution is to make the results of COPYs be variables and to refer to those variables in the rest of the macro.

```
FIND [selection]
...parameter setting actions...
*1 = START
```

The Search command finds an object at run time based on an associative description. The object found, of course, may vary from run to run. To permit macros to deal with such objects, the results of searches are made variables.

```
*1 = FOR EACH [selection]
```

The definer can tell the system that a sub-sequence of commands is to be done for each element in a repeating structure (repeating field group, table or folder). The FOR EACH command sets the variable to each element in turn, then executes its associated "body" of commands. When all the elements in the repeating structure have been processed, the macro will continue.

Variables are recorded in command sequences as *declarations:*

```
*1 = PAUSE "prompt"
    means "*1 is defined to be the result of a PAUSE"
*1 = COPY [selection] TO [destination]
    means "*1 is defined to be the result of a COPY"
*1 = START
    means "*1 is defined to be the result of a FIND"
*1 = FOR EACH [selection]
    means "*1 is defined to be each element of the repeating structure: [selection]"
```

The results of these operations are not explicitly in the typescript, of course, because they vary from run to run. However the system does remember what the result of each operation was. If, during the definition, the same object or any part of it is subsequently selected, the system records the new selection in a generalized representation.

For example suppose I, the macro writer, wants to permit some object, say a table, to be different from run to run. I might invoke a PAUSE and give it the text string "Please select a table entry to use.". This would be recorded as:

*1 = PAUSE "Please select a table entry to use."

Every time the macro is run, it will pause and print out the specified message. The macro runner should select a table entry and invoke CONTINUE. Since definition is programming by example, I, the macro definer, must also do the same thing. So I select a table entry and invoke CONTINUE. This now *binds* an actual value to the variable. Internally variables are kept in a list (or stack or array). So the following entry is made in the list:

*1: [Document "foo" #, Frame 1, Table 2, Row 3, Column 4]

Such variable values have the standard selection representation, just like any other. Now the interesting question is what happens if I, the definer, select some other element in the same table, or something else in the same frame, or even something in the same document? According to the pattern matching algorithm presented below, this new selection is generalized in so far as its selection representation matches the current value of *1. So if I select an entry in the same row but a different column, the selection might be recorded as:

[Document *1, Frame *1, Table *1, Row *1, Column 5].

If I select an entirely different entry in the table:

[Document *1, Frame *1, Table *1, Row 6, Column 7].

If I select something else in the frame:

[Document *1, Frame *1, Symbol 18, Guiding 1].

If I select something else in the document:

[Document *1, Frame 2, Table 2, Row 3, Column 4].

Note that as soon as a selection representation differs from a variable value, no further generalization is done. Thus in the last example, only the document is the same and is generalized, even though both selections designate the second tables in their respective frames and row 3, column 4. This makes generalization a hierarchical process, in terms of containment. It eliminates accidental generalization in which object identifications just happen to match without any real connection between them.

Finally, a variable value can itself contain variables:

*2: [Document *1, Frame *1, Table *1, Row *1, Column 3].

A pattern-matching algorithm for generalizing selections

The following is the algorithm which generalizes Star selections (i.e. turns them into variables or partial variables).

Every variable always has a value. Variable values are stored in a list (or stack or array) ordered from most-recently-set to least-recently-set.

> (1) When a selection is made, a constant representation is built for it according to the syntax presented earlier. This representation is then matched against the variable list, from front to back.
>
> (2) For each variable value, matching proceeds from left to right in the representation. As long as the selection representation exactly matches the variable representation, matching continues. As soon as they differ, matching stops. The number of parts that matched is remembered.
>
> (3) If the selection completely matches (is the same as) a variable value, the algorithm stops immediately. The selection representation is replaced by the variable number, e.g. *10, which becomes the new representation for the selection.
>
> (4) If no variable value matches completely but some partially match, then the longest match is used. In case of ties, the most recent value is used. For each part of the selection representation that matched, the object identification is replaced by the variable number, e.g. Document *10. This becomes the new selection representation.
>
> (5) If no variable parts match, then the selection representation remains a constant.

If a variable value itself contains a variable, as in the last example in the preceeding section, then each variable object identification is expanded to a constant before matching.

Recordable objects and actions

The following Star objects and actions can be recorded and will appear in command sequence text:

Commands and property sheet changes

- Keyboard and menu commands are recorded in the form:

  COMMAND {[selection]} {TO [destination]}   where {} => optional

Examples:

DESKTOP
DELETE [selection]
MOVE TO [destination]
MOVE [selection] TO [destination].

• Property sheet changes and commands which involve option sheets are recorded in the order in which actions are done. Any parameters not changed are left alone. This normally produces different results from run to run because the starting states are different. However this is expected to be the desired situation. Syntax:

```
COMMAND [selection]
...parameter setting actions...
DONE or START
```

Examples:

```
PROPERTIES [selection]
SELECT [Sheet "Character Properties", Option "Face" BOLD]
SELECT [Sheet "Character Properties", Option "Name", Caret 0]
|Chapter Title|
DONE

FIND [selection]
COPY [selection] TO [Sheet "Find", Option "Search for", Caret 0]
SELECT [Sheet "Find", Option "Range" CURRENT SELECTION]
SELECT [Sheet "Find", Option "Match on" TEXT]
START
```

• Typed-in text, which does not require a command since Star uses modeless insertion, is recorded in the form:

"|" <any typable characters possibly including frames> "|"

where "|" is some non-typable character to be determined.

Examples:

|Now is the time for all good men *to come to the aid of their party.*|

|Here is some type-in containing a frame. *Such frames always float vertically and horizontally in stored command sequence text.*|

Selections

Discused above.

Destinations

A text destination is either a character or a caret. In either case it is recorded as a text selection, indicating the character *after which* insertion is to occur; e.g. Text 73, Caret 73. "Last" is used to indicate the last character in a document, field or text scroll; e.g. Text last, Caret last. "Caret 0" indicates a caret placed before the first character.

A point destination is a pair of x-y coordinates relative to the Desktop or to the origin of a frame if inside a frame. X-Y points are in screen dot units. Every frame has an "origin", coordinates (0,0), which is initially the upper left corner. However the top and/or left border of a frame may be altered by MOVEing the frame's control points. Since the origin always remains fixed at (0,0), the upper left corner may no longer be at (0,0) and may even have negative coordinates. Coordinates on the Desktop are always positive. Examples:

[X-Y 100 200]	a destination on the Desktop
[Document "foo" #, Frame 1, X-Y 93 -17]	a destination in a frame

Playback

When a recorded command sequence is executed, a question arises as to what the run-time environment must be. If an icon was on the Desktop when a macro was defined, must it be on the Desktop at run time? What if it was open? What if it was in the directory, or in a folder, or in a file drawer? The example of the Calculator most clearly illustrates the answer. As a macro definer, I may very well want the Calculator open on the screen so that I can use its arithmetic capabilities. This is particularly true if I am defining a field fill-in rule. However I certainly do *not* want the Calculator popping up on the screen every time I fill in a form. Nor do I want to require that the Calculator first be opened before I can fill in a form, since the Calculator has nothing to do with the form as far as the user is concerned. The form filler-iner certainly isn't going to use it.

The answer is that the Calculator and all other icons *must be usable by command sequences even when in a closed state.* In general this requires the following steps:

- gain access to the icon
- open it
- access the icon contents
- close it
- return the icon to its previous location

In practice, one or more of these steps might already be done, e.g. the icon might already be open (in which case, of course, it wouldn't be closed either). Thus the form designer could have the Calculator on his Desktop and make heavy use of it. The clerk filling in the form could leave the Calculator in the directory (and might never even know about it). Both would be able to use the same macro.

There are various issues having to do with sharing. If a document used in a command sequence is in a shared file drawer, the document and/or file drawer may not be available at run time. But if the above sequence is followed by the implementation -- gain access, open, access contents, close, put back -- then the standard Star mechanisms can be used. No additional macro mechanisms will have to be added. At the most the user will see an occasional message of the flavor: "File Drawer so-and-so is presently in use by B. Jones. Do you wish to wait for it to become available or stop this command sequence? Select one: WAIT STOP"