Technical Report 843

TEMPEST: A Template Editor for Structured Text

Peter J. Sterpe

MIT Artificial Intelligence Laboratory

This blank page was inserted to preserve pagination.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

A.I. T.R. No. 843 June 1985

TEMPEST:
A Template Editor for Structured Text

by

Peter J. Sterpe

Abstract

TEMPEST is a full-screen text editor that incorporates a structural paradigm in addition to the more traditional textual paradigm provided by most editors. While the textual paradigm treats the text as a sequence of characters, the structural paradigm treats it as a collection of named *blocks* which the user can define, group, and manipulate. Blocks can be defined to correspond to the structural features of the text, thereby providing more meaningful objects to operate on than characters or lines.

The structural representation of the text is kept in the background, giving TEMPEST the appearance of a typical text editor. The structural and textual interfaces coexist equally, however, so one can always operate on the text from either point of view.

TEMPEST's representation scheme provides no semantic understanding of structure. This approach sacrifices depth, but affords a broad range of applicability and requires very little computational overhead. A prototype has been implemented to illustrate the feasibility and potential areas of application of the central ideas. It was developed and runs on an IBM Personal Computer.

Copyright (c) Massachusetts Institute of Technology, 1985

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grant MCS-8117633, in part by the International Business Machines Corporation, and in part by Honeywell Information Systems, Incorporated.

The views and conclusions contained in this document are those of the author, and should not be interpreted as representing the policies, either expressed or implied, of the Department of Defense, of the National Science Foundation, of the International Business Machines Corporation, or of Honeywell Information Systems, Incorporated.

This report is a revised version of a thesis submitted to the Department of Electrical Engineering and Computer Science on June 3, 1985 in partial fulfillment of the requirements for the degree of Master of Science.

Table of Contents

1. The Basic Ideas **1**

1.1 The Textual Paradigm 1
1.2 The Structural Paradigm 2
1.3 Combining the Two Paradigms 4
1.4 Uses for Structure 6
1.5 Related Work 6
1.6 Fundamental Limitations 7

2. Scenario **9**

2.1 Defining Blocks and Groups 9
2.2 Navigation Commands 12
2.3 Constraints 15
2.4 Flexible Viewing Commands 18
2.5 Reuse Commands 24
2.6 Structural Manipulation Commands 29

3. Implementation Issues **33**

3.1 The Host Environment 33
3.2 Representing Structure 33
3.3 Auxiliary Files 34
3.4 Constraints 34
3.5 The Delete Buffer 35
3.6 Command Interface 35
3.7 Buffer Markers 36
3.8 The Show Buffer 36
3.9 Automatic Framing and Unframing 36

4. Future Directions **37**

References **39**

1. The Basic Ideas

Although text editors span a wide spectrum of sophistication, most share approximately the same paradigm. Under this *textual* paradigm, the text is presented to the user as a sequence of characters. Editing consists of modifying this sequence. Characters are the only accessible units of this paradigm.

Since characters are often too fine a subdivision, most editors extend the textual paradigm by making it possible to operate on objects which are more meaningful. EMACS [Stallman 81], for example, allows one to operate on words, lines, sentences, and paragraphs. These are identified as needed by dynamically parsing the character sequence.

TEMPEST augments the traditional textual paradigm by incorporating a *structural* paradigm which allows the user to define arbitrary units to operate on. Under the structural paradigm, the user can name these units, specify their extent, and group them to correspond to the logical components of the text, as opposed to the mechanical components such as characters and lines. Although TEMPEST does not attach meaning to these objects, it does make them accessible to the user who understands their meaning and can manipulate them in a useful way.

This is a weaker concept of structure than syntax-directed editors have, for example, but it is also broader, since it is not restricted by a grammar. Furthermore, TEMPEST, unlike most syntax-directed editors, does not abandon textual commands in order to provide structural commands. Rather, both coexist.

The major result of this research is a running prototype which has proved useful in a number of ways -- it has demonstrated the feasibility of the ideas behind TEMPEST; it has been a vehicle for discovering areas of application for the structural commands and features; and it has suggested ways to integrate and even overlap the two paradigms.

The prototype was built on an IBM Personal Computer by modifying and extending a commercially available editor. It runs quickly, without any optimization having been performed. These facts argue not only for the feasibility of such an editor, but also for its desirability -- TEMPEST is a significantly more powerful tool than the editor it was built on, yet is not significantly more expensive, either in terms of implementation time or computational overhead.

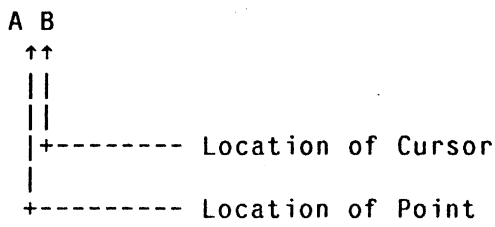
1.1 The Textual Paradigm

TEMPEST supports the textual paradigm in the same way EMACS does. The text is presented as a sequence of characters, which for display purposes is broken into lines by "newline" characters. It is not treated as a set of lines, however -- the newline characters are part of the sequence and can be edited. The interface is "full-screen," which means the characters are directly accessible by their position on the screen.

The Basic Mechanisms

All operations under the textual paradigm are based on four key objects: the point, the mark, the region, and the delete buffer. Below is a description of each, as well as a description of the more important commands that will be referred to in subsequent chapters.

Point -- The point is the location in the character sequence where the next insertion or deletion will take place. It is thought of as being between characters, in particular, to the "left" of where the cursor is. Thus, if the cursor is on the "B" below, the point is between the "A" and the "B":



Mark -- The mark indicates some previous location of the point. Like the point, it is not directly visible; it is simply a place in the character sequence. One speaks of "setting the mark," which causes the mark to refer to the location of the point at that time. In the editor underlying TEMPEST, there is one mark. (Some EMACS implementations simulate multiple marks by pushing the various locations of the mark on a stack.)

Region -- The region is all the text between the point and the mark. Two particularly important commands which operate on the region are KILL-REGION and COPY-REGION. The former deletes all the text in the region and saves it in the delete buffer; the latter copies the text of the region into the delete buffer without deleting it.

Delete Buffer -- The delete buffer corresponds approximately to what in some implementations of EMACS is known as the "kill ring." Commands that delete a word, line, sentence, paragraph, or the region save that text in the delete buffer. Text from successive deletions is concatenated. The contents of the delete buffer can be inserted into the text at any time with the YANK command. A common way to move pieces of text is to kill and then yank them. Replication is accomplished by successive invocations of YANK.

Limitations of the Textual Paradigm

The textual paradigm that TEMPEST uses is reasonably powerful, mostly because it allows one to manipulate the text in familiar terms -- words, sentences, and paragraphs. In addition, the full-screen interface is analogous to the familiar notion of a page of text. What the paradigm lacks, however, is a representation for the structure of the text being edited.

The dynamic parsing of the character sequence to recognize words, etc., is a step toward making the text's structure accessible. Unfortunately, these parsed objects are a weak representation for structure because they are at too low a level. They reflect the mechanical structure of the text, but not its logical structure, which actually guides the composition and revision of a body of text. Independent of computer text editors, one conceptualizes text in terms of meaningful units, for example, "Chapter One," "Figure 2," "the background discussion." An editing paradigm that lacks the ability to operate on such units fails to capture a natural and therefore powerful way to approach the editing process.

1.2 The Structural Paradigm

The structural paradigm provides a way to represent the structure of text. In TEMPEST, saying that text has "structure" means it contains a number of logical units which constitute some abstract view of it. In the vocabulary of the structural paradigm, these units are known as *blocks* and *groups*. TEMPEST does not consider the text to be *built* out of structural units, however. Instead, the structure is considered to be a kind of annotation that can be superimposed upon the text. This is illustrated in Figure 1-1.

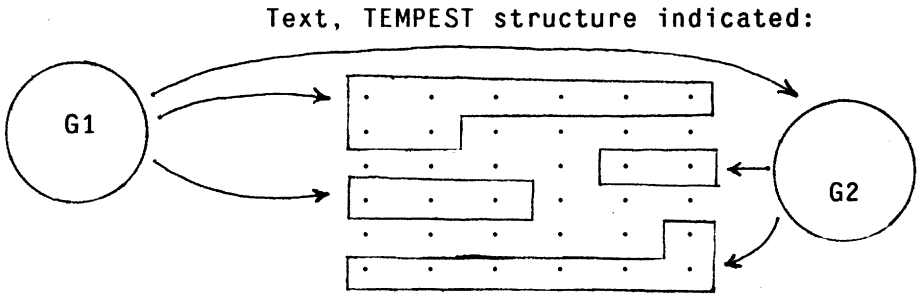


Figure 1-1: Abstract View of TEMPEST Structure

There are two forms of structure in this picture: the boxes that enclose text, and the circles that point to boxes and other circles. The boxes correspond to blocks; the circles to groups of blocks which are related. This is elaborated upon below.

The Basic Mechanisms

The purpose of blocks is to identify and delimit logical units in the text. If blocks are chosen and named appropriately, they can represent the meaning and purpose of the text they enclose. For example, this subsection could be enclosed in a block named "basics," and this paragraph in a block named "purpose-description." TEMPEST provides a suite of commands which operate on blocks, thereby enabling the user to edit in terms of structure.

Physically, a block is a sequence of characters that can begin and end anywhere in the text. Blocks can contain other blocks; the only restriction upon this is that they must nest properly. Blocks are user-defined, that is, the user specifies where a block begins and ends, and also the name it is to have (names are optional). The text contained in a block is called its *value.*

One can create blocks at any time. They need not be created first and filled later, although that is one method of using them. It is entirely possible to put all the blocks in place after typing the text, thereby superimposing structure where appropriate. Blocks can thus be used both for construction and annotation.

The other unit of structure in TEMPEST is the group, which is a named collection of blocks and/or other groups. Groups collect together related structural entities, for example, all the figures in a document or all the sections in a chapter. Unlike blocks, which have a physical correspondence to the text, groups are conceptual and are not anywhere "in" the text.

The blocks of a group are not constrained to be physically contiguous. Note Group G1 in Figure 1-1. In fact, blocks from different groups can interleave, since membership is by name and not location. More important, any number of groups can exist without interfering with each other physically. Thus, one can impose several structures on the same text, each having a different point of view or level of granularity.

Because groups can contain other groups, they can be used to represent hierarchical structures such as sectioned documents. One is not limited to hierarchical arrangements, however, since an entity can be a member of more than one group. The resulting hierarchies, then, can actually be graphs, however they are required to be acyclic. For example, a section heading can plausibly belong to two groups: a group containing its particular section, and a group containing all section headings.

TEMPEST uses two kinds of groups, explicit and computed. For explicit groups, the user explicitly declares which other blocks and groups are to become members. Members can be added or subtracted at any time, but this, too, must be done explicitly. The membership of a computed group, on the other hand, is determined dynamically based upon a pre-specified computation. TEMPEST currently uses one computed group: the group of all blocks that are textually contained within a block. A typical example of this is a "section" block which contains a number of "paragraph" blocks. Such a relationship should be dynamic rather than static, since it would become invalid if, for example, a paragraph were moved to another section.

Limitations of the Structural Paradigm

TEMPEST's structural paradigm is missing semantics. Blocks and groups convey information about the identity, location, and extent of structural components, but not about their relationships or content. They do provide a useful handle for manipulating structure. For representing structure, however, they are useful primarily because they are given meaningful names. Meaningful names, however, are not equivalent to "meaning." It is the user who ultimately has the knowledge of what the structure actually means.

As a simple example of how the structural paradigm lacks semantics, suppose the user has created a block named "paragraph" and inserts a blank line in the middle of it, intending to split it in two. The result will *not* be two "paragraph" blocks; it will be one with a blank line in the middle. TEMPEST has no knowledge of what a "paragraph" is, so it cannot understand the user's action.

It is interesting that the editor underlying TEMPEST contains some semantic content, while TEMPEST's structural paradigm does not. This is because that editor goes beyond the textual paradigm by parsing for words, sentences, and paragraphs. This is relatively low-level, however. What prevents it from reaching a higher level is that one can only parse a fixed number of predefined objects, thereby limiting the flexibility and applicability of the editor. Furthermore, the capability to parse a broad and truly useful set of objects is

not trivial. Syntax-directed editors, on the other hand, have a greater understanding of structure, but can operate only in one restricted domain because they are grammar-based.

It is difficult to incorporate semantics without restricting the applicability of the editor. TEMPEST sacrifices semantics to retain breadth, yet still achieves considerable utility.

1.3 Combining the Two Paradigms

TEMPEST's structural paradigm is combined additively with the textual paradigm. This means that the addition of the structural commands has not detracted from the original core of textual commands. As far as possible, neither paradigm conflicts with the other.

The significance behind this additivity principle is that the inclusion of the structural paradigm has not changed the already familiar textual paradigm. Contrast this with syntax-directed editors, for example, whose support of structure typically requires one always to think in terms of nodes and parse trees. These editors abandon textual commands in favor of structural ones. TEMPEST's choice is to offer both paradigms, allowing the user to use whichever is most convenient for a particular operation. This approach is discussed in detail in [Waters 82a].

A consequence of supporting two paradigms is that they will interact. Particularly important in TEMPEST are insertion and deletion operations which often have structural side-effects. These are described below.

Interactions Between the Paradigms

Although blocks are objects within the structural paradigm, they are strongly related to the textual paradigm, since they correspond directly to sequences of characters in the text. It would be wrong to make them accessible only through structural commands. For example, one should not have to issue a structural command to add sentences to a block that encloses a paragraph. Therefore, TEMPEST makes it possible to expand, shrink, move, and even delete blocks using the usual text insertion and deletion commands. These operations are described below.

Consider the illustration in Figure 1-2. Each letter represents a character in the buffer. The braces, "{" and "}," indicate the beginning and end of a block, respectively, but are not in the buffer.

The block contains the text "ABC"; all other text is outside it. The numbered arrows indicate five locations of interest where insertions or deletions could take place. (The arrows are between characters because insertions and deletions take place at the point, which is thought of as being between characters.)

Deletion would have the same effect at each of the five locations, deleting the character to the right of the point. Thus, one can shrink a block by deleting a character within it or at its beginning boundary (Location 2).

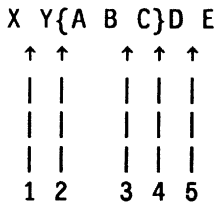


Figure 1-2: Relevant Insertion/Deletion Locations

Insertion, however, is not as predictable, and its effect at each location is summarized in Figure 1-3. Note that insertion at Locations 1 and 5 has no effect on the block, since neither is within it or on a boundary. One can expand the block by inserting a character at Location 3, since this is within the block.

Insertion at Locations 2 and 4 is ambiguous. TEMPEST's default behavior is to insert inside the block at Location 2 and outside it at Location 4. (This can be overridden using the FRAME command, explained in Screen 2-13.)

Location	Effect of Insertion on Block
1	None.
2	Ambiguous. Default: character inserted inside the block, just before the "A."
3	Character inserted inside the block.
4	Ambiguous. Default: character inserted outside the block, just before the "D."
5	None.

Figure 1-3: Effects of Direct Editing on Blocks

Direct editing can also be used to move or delete a block, but this cannot be done with single-character deletion commands. Instead, it is done with commands that delete a region of text, for example, KILL-WORD or KILL-LINE. To delete a block, one need only delete a region which contains it. Doing so causes the deleted text to migrate to the delete buffer. To move a block, one need only delete it, and then move the point before executing YANK. (For a region to contain a block, the point must be before the beginning boundary and the mark after the end boundary. This can be accomplished conveniently with the MARK-BY-NAME command, which prompts for the name of a block and places the point and mark around it.)

The YANK command has interesting side-effects itself. The first time a deleted block is yanked, TEMPEST assumes it is being moved, and essentially "undeletes" it. If a block is yanked repeatedly, however, TEMPEST assumes it is being copied and places the yanked version in all the groups the original block was a member of. In some sense, the copies are made "siblings" of the original.

Remembering Structure

For example, TEMPEST "remembers" structure -- any blocks and groups defined in a file will still exist when that file is edited in a later session. This supports the structural paradigm, which expects structure to exist until it is explicitly deleted. On the other hand, text in disk files does not appear structured in any way. This supports the textual paradigm, which expects saved files to be usable by other programs. In fact, any program can use a structured text file as long as it does not modify it. If the file is modified, the structure will be lost.

1.4 Uses for Structure

TEMPEST's commands and features can be divided into five areas of application, each of which is summarized below. They are illustrated and explained fully in the next chapter.

Navigation -- Moving through the text using blocks as landmarks. Navigation by structure complements navigation by string search by basing the search on the name of the block (and thus its purpose) rather than its contents.

Constraints -- Action routines that are triggered by operating on blocks. Constraints can be used to enforce syntax or semantics, but more generally to effect any kind of change on the editing environment, for example, propagating text from one block to another.

Flexible Viewing -- Viewing the text in different ways based on its structure. For example, obtaining an outline of the currently defined blocks and groups.

Structural Manipulation -- Rearrangement or replacement of blocks and groups. Commands in this area are powerful because they effect large-scale changes with only one command invocation.

Reuse -- Editing with pieces of structured text known as *templates* which represent standard fragments. Templates contain empty blocks, referred to as *slots*, which are placeholders for text to be filled in by the user. Text can be built up by inserting templates and filling their slots.

1.5 Related Work

Text editors span a broad spectrum of purpose and degree of sophistication, which one might characterize roughly as follows:

* Text-only Editors -- These treat the text as consisting only of characters. They have no provisions for recognizing or manipulating structure, except at a very low level (e.g., parsing for words or sentences). Editors in this class have broad applicability and are not specialized for any one kind of editing.

* Structure Editors -- This term is used loosely to indicate editors that allow some form of structure to be accessed and operated upon. Generally the members of this class have a small repertoire of structure objects whose relationships they understand. For example, document editors know about chapters, sections, etc., and syntax-directed editors know about programming statements. These editors usually have a restricted domain and exhibit greater depth than breadth.

* Semantic Editors -- A good example is the KBE [Waters 82b], a program editor which understands the control and data flow of a library of programming fragments. While the KBE has great depth, its domain is narrow, limited only to programs.

TEMPEST resides near the middle of this spectrum, mostly because it shares certain elements with a number of systems which are also near the middle. There is a fundamental difference between TEMPEST and its nearest neighbors in the spectrum, however. The other programs have generally applied structure to increase editing capability in a particular domain, for example, the Cornell Program Synthesizer [Teitelbaum 79]. In doing so they have had to trade breadth of applicability for depth. TEMPEST has attempted to depart from this trend by retaining breadth while seeking depth.

In spirit, TEMPEST is most closely related to the Knowledge Based Editor (KBE), from which some of its basic ideas were drawn. Templates and slots are directly analogous to the cliches and roles of the KBE. A major part of the KBE for which there is no analog in TEMPEST, however, is the idea of a *plan*. A plan is an abstract representation of a program in which data flow and control flow knowledge is explicitly represented. KBE commands, which seemingly operate on the text of a program, actually operate on its plan.

TEMPEST cannot exhibit the depth of the KBE because its representation scheme embodies no semantics. It does not need to exhibit such depth, however, in order to support the paradigm of editing with templates. Many of this paradigm's operations are mechanical, requiring no special knowledge on the part of

the editor. TEMPEST makes a contribution simply by giving the user access to the text's structure and allowing the user to manage the semantic details.

Some of TEMPEST's interface concepts were inspired by similar features in other systems. In particular, the flexible viewing command SHOW is derived from the concept of selecting collections in CREF [Pitman 85]. The flexible viewing concept in general is found in systems such as CREF, AUGMENT [Engelbart 84], and Hypertext [Nelson 81], where it is almost a necessity, due to the structural complexity allowed by those systems. In TEMPEST, it is used not so much to walk one's way through a complex structure as to bring the structure to the fore, given that it's usually in the background.

A somewhat different interface issue in TEMPEST was that it have the look and feel of a "text-only" editor. Certain document editing programs, particularly the Document Editor [Walker 81] and Etude [Hammer 81], influenced that decision. "Tree-walking" type systems, such as ED3 [Stromfors 81], provided negative examples of what the interface should be like.

TEMPEST's constraint facility was inspired by two very different sources: syntax-directed editors, and spreadsheet programs. The notion of constraining slot values came from syntax editors. The idea of using the facility more generally to propagate values and produce wide-spread changes came from the commercially available spreadsheet programs.

1.6 Fundamental Limitations

TEMPEST's fundamental weakness is its lack of semantic understanding. Unfortunately, this is not a problem of the implementation. It is an inherent consequence of the text-based approach taken. A place where this limitation is particularly noticeable is in inserting templates into the buffer. TEMPEST is only capable of placing the designated text at the cursor, directly adjacent to whatever text surrounds it. This often produces unaesthetic results.

Fortunately there is a tool that works very well in tandem with TEMPEST to smooth over such flaws -- the document compiler SCRIBE. In fact, the text in the scenario in Chapter 2 incorporated SCRIBE commands purposely to illustrate how TEMPEST can best be used. Since TEMPEST has no knowledge of how to "glue together" templates, it needs to be used with some system that can accept ragged input and produce correct output. SCRIBE, and formatters in general, are perfectly suited for this use.

In order for TEMPEST to have broad applicability, it should also be useful for editing programs. One could incorporate a pretty-printer to fulfill a role analogous to that of SCRIBE. A more important issue than text format, however, is the impact of naming conflicts on program semantics, particularly conflicts in variable names between programming templates. Although there are ways to work around this problem, some semantic understanding of programming templates is required to handle it properly.

In general, future applicability of a tool like TEMPEST would depend on input-insensitive applications, such as editing with the help of formatters and pretty-printers. A template-like paradigm could also be used as an interface to an operating system, for example, since command monitors are usually not excessively restrictive about the form of their input.

This empty page was substituted for a blank page in the original document.

2. Scenario

This chapter presents a scenario illustrating the use of TEMPEST's commands. The scenario is divided into sections according to TEMPEST's areas of application.

In the scenario, the user is in the middle of editing the reference manual for a mythical program called "Softwhere." Each step of the scenario corresponds to one or two command invocations, and is represented by a view of the editor screen after the commands are executed.

2.1 Defining Blocks and Groups

Defining Blocks

Blocks are defined by typing their printed representation into the text. This is a concise interface, since it eliminates the need for a command to create blocks. It is also intuitive -- to create a block, one simply types in text that looks like a block. The most powerful feature of this interface, however, is that it is not temporal. The user is not constrained first to create empty blocks and then to fill them. Rather, he can make any portion of the text into a block at any time simply by enclosing it in braces. This type of interface is discussed in [Ciccarelli 84].

A block's representation indicates four pieces of information about it: its beginning, its end, its name, and its default. An example is shown below.

```
{|bas|foo}
```

The opening and closing braces ("{}") mark the block's beginning and end, respectively. The text after the first vertical bar ("|") but before the second is the block's default, which is optional. The block shown above has default "bas." To omit a default, one omits the first "|". The text after the second vertical bar is the block's name, in this case, "foo." The name is also optional. However, the "|" preceding it is mandatory. Following are the possible configurations for names and defaults:

`{|foo}`	A block named "foo" with no default.
`{|}`	An unnamed block with no default. The ("|") is mandatory.
`{|bas|foo}`	A block named "foo" with default value "bas." Both vertical bars are needed -- the first indicates the presence of a default; the second indicates the beginning of the name.
`{|bas|}`	An unnamed block with default value "bas." Again, both vertical bars are needed.

All of the blocks shown above are *empty* -- there is no text before the first vertical bar. Empty blocks are referred to as *slots.* Any text appearing before the first "|" is known as the block's *value.* A block containing a value is said to be *filled.*

Empty blocks are always displayed *framed,* that is, with braces surrounding the default and name. Filled blocks, however, are displayed *unframed* -- only the value appears. TEMPEST unframes a filled block as soon as the cursor is moved off of it. Should it become empty again, TEMPEST will reframe it.

In Screen 2-1 the user is in the middle of editing the title page of the Softwhere reference manual. A summary of the dialogue between TEMPEST and the user is shown above the screen. What the user types is in the normal type face; TEMPEST's prompts are in boldface; commentary is in italics. The notation "C-Z G" means "Control-Z" followed by "G." (All TEMPEST commands are invoked as a C-Z sequence.) Also, "<CR>" represents a carriage return. In each screen of the scenario, any characters that have changed from the previous screen are shown in boldface. The cursor is represented by a hollow square (□).

In addition to the text of the manual itself, the file also contains several SCRIBE formatting commands [Reid 80]. The SCRIBE commands are preceded by the "@" character. SCRIBE is a high level text formatter with which the user creates documents of types known to SCRIBE through a database of predefined *document types*. The parts of a document are called its *environments*. The document to be created is specified with the Make command (e.g., @Make(Manual)), and each environment is filled by enclosing its text in named delimiters (e.g., @Begin(TitlePage)<CR>*text of title page*<CR>@End(TitlePage)). The user must specify the environments in the proper order, but SCRIBE does the rest, e.g., justification, margin control, pagination, etc.

The file being edited in Screen 2-1 is `MANUAL.MSS`, as indicated on the mode line (the bottom line of the screen). SCRIBE input files must have ".MSS" as their file name extension. The user has just typed "`{|title-env}`" to define a slot that will contain the text of the `TitlePage` environment. Note in the dialogue that TEMPEST prompted **`Add to Group?`** This gave the user the option to add the block to an explicit group. Had the user named a group that did not exist, TEMPEST would have created it.

`{|title-env}` **`Defined title-env. Add to Group?`** `<CR>` **`Done.`**

```
@Make(Manual)
@Begin(TitlePage)
{|title-env}□
@End(TitlePage)

Note:  This manual applies to Softwhere Version {|version-num}.
```

```
TEMPEST  main:MANUAL.MSS
```

Screen 2-1

Defining and Building Groups

Screen 2-2 illustrates the ADD-TO-GROUP command, which prompts for the name of a target group and an object to add to it. One can respond to the second prompt with the name of either a block or group. If the target group does not exist, TEMPEST creates it.

Notice that it is possible to create groups as side-effects of other operations. Since creating an empty group is an intermediate step, TEMPEST does it implicitly rather than requiring the user to do it. In fact, TEMPEST does not provide a command whose function is to create an empty group.

In this screen, the user adds block `title-env` to group `env-group`. This has no effect on the text in the buffer.

```
C-Z A Add-to-Group Group? env-group<CR> Object to Add? title-env<CR>
```

```
@Make(Manual)
@Begin(TitlePage)
{|title-env}□
@End(TitlePage)

Note:  This manual applies to Softwhere Version {|version-num}.




TEMPEST  main:MANUAL.MSS
```

Screen 2-2

2.2 Navigation Commands

Navigation refers to the ability to move throughout the buffer to find some particular place in the text. Editors that operate under a strictly textual paradigm usually support navigation with a string search facility. The editor underlying TEMPEST provides string search. In addition, TEMPEST also supports navigation using blocks and groups as landmarks. The advantage of this approach is the ability to identify a piece of text by its purpose rather than its contents.

A significant drawback of navigating by string search is that a unique search string must be specified to land in the right place. For example, if one were writing a sectioned document and wanted to work on section "4.2," then a likely search string to use would be "4.2". What would often happen in such a case is that the editor would first find "1.4.2," and then "2.4.2" and "3.4.2," and perhaps several forward references to "Section 4.2" before finally landing at the start of the proper section.

The real flaw with string search is that it bases its search on textual content, which is difficult or even impossible to specify. For example, how does one determine a search string when editing somebody else's file? Even the owner of a file would have trouble pinpointing something as abstract as the "chase scene" or the "hashing function."

Often, a piece of the text can be best identified by its role. In TEMPEST, it is possible to indicate this role by enclosing the text in a block with an appropriate name. Then, navigation on the basis of role is simply a matter of searching for the right block rather than the right text string. TEMPEST provides such a navigation facility as a complement to string search.

GOTO

The GOTO command prompts for the name of a block and puts the cursor on its first character. If a group is named, the cursor is placed on the block within that group that appears earliest in the buffer. In this screen the user requests TEMPEST to GOTO the block named `i/o-example`. Note that executing GOTO has brought an unseen part of the text into view.

This screen employs a few new SCRIBE commands, @Tag and @Caption, which are associated with figures. @Tag provides a label for use when referring to the figure; @Caption provides the caption.

```
C-Z G Goto? i/o-example<CR>
```

```
You can direct Softwhere output to the screen, the printer, or a disk
file.  These options are presented in the Output Menu, shown below:
@Begin(Figure)
□                              OUTPUT MENU

                       Current File: LETTER.TXT

                       (1)    Display on Screen
                       (2)    Send to Printer
                       (3)    Write to File

                            Choice:  ( )
@Tag(outputfig)
@Caption(Options Available in the Output Menu)
@End(Figure)


TEMPEST  main:MANUAL.MSS
```

Screen 2-3

NEXT

NEXT moves the cursor to the next empty block. If necessary, the display is scrolled to bring it into view. No action is taken if no empty blocks remain in the file. (The NEXT command has an inverse, called PREV, that searches for the previous empty block.)

In this screen NEXT places the cursor on block `menu-tree`. This command execution has scrolled the display to a part of the text previously unseen.

C-Z N *Next*

```
Below is a diagram of Softwhere's command menus and their relationship:
@Begin(Figure)
{|menu-tree}
@Tag(treefig)
@Caption(Relationship Among Softwhere's Command Menus)
@End(Figure)


@Section(Summary)
This chapter presented an overview of Softwhere's command menus.
The diagram representing them is reproduced below for convenience:
@Begin(Figure)
{|tree-recap}
@Tag(recapfig)
@Caption(Recap of Command Menus)
@End(Figure)


TEMPEST  main:MANUAL.MSS
```

Screen 2-4

2.3 Constraints

Constraints in TEMPEST take the form of action routines that are triggered by operating on blocks. TEMPEST employs a general scheme that associates an action routine with a list of blocks which are its arguments. The blocks are called *partners* and are said to be *linked* under that constraint. The action routines are C procedures which are compiled as part of the editor.

Constraints were motivated by the desire to enforce correctness upon slot values, much as syntax-directed editors enforce the syntactic validity of expressions. The scheme employed affords even more power than this, however. As with syntax-directed editors, constraints can be written which enforce syntax and semantics, but one can also write routines which produce side-effects upon the editing environment, for example, propagating text from one block to another.

There are two classes of constraints, "enter" and "exit." Enter constraints fire when the cursor enters a constrained block, and exit constraints fire when the cursor leaves a constrained block. TEMPEST currently implements two constraints, equality and number, both of which are exit constraints.

Specifying Constraints

A constraint is attached to one or more blocks with the ATTACH-CONSTRAINT command, which prompts for the name of a constraint routine and a list of blocks to be its arguments. In this screen the user attaches the equality constraint to blocks `menu-tree` and `tree-recap`. This operation does not change the buffer contents; its effect is indicated in the dialogue at the top of the screen.

```
C-Z C Constraint? equality<CR> Arguments? menu-tree,tree-recap<CR>
```

```
  Below is a diagram of Softwhere's command menus and their relationship:
  @Begin(Figure)
  {|menu-tree}
  @Tag(treefig)
  @Caption(Relationship Among Softwhere's Command Menus)
  @End(Figure)


  @Section(Summary)
  This chapter presented an overview of Softwhere's command menus.
  The diagram representing them is reproduced below for convenience:
  @Begin(Figure)
  {|tree-recap}
  @Tag(recapfig)
  @Caption(Recap of Command Menus)
  @End(Figure)


  TEMPEST  main:MANUAL.MSS
```

Screen 2-5

The Equality Constraint

The goal of the equality constraint is to propagate the contents of a changed block to its constraint partners. The trivial use of this constraint is as a convenience feature. For example, one could save typing by linking the "return address" slot and the "signature" slot of a business letter, since the person who sends a letter usually also signs it. The equality constraint can be used more substantially, however, particularly to ensure consistency among related blocks.

For example, consider the case of slots `menu-tree` and `tree-recap` from Screen 2-4. These are both intended to contain the same figure -- `tree-recap` is included for convenience so the reader won't have to flip pages when reading the summary. Certainly these slots should always contain the same text, so the user linked them under the equality constraint in the previous screen.

In this screen the user has begun to type the figure into the `menu-tree` slot, which now contains the text `"MAIN MENU"` surrounded by a box. Note that the cursor has been moved off of the `menu-tree` slot, and this slot's text has propagated to the `tree-recap` slot in the lower part of the screen.

Direct Editing C-N *Cursor to Next Line*

```
Below is a diagram of Softwhere's command menus and their relationship:
@Begin(Figure)
              +----------------------+
              |      MAIN   MENU     |
              +----------------------+
@Tag(treefig)□
@Caption(Relationship Among Softwhere's Command Menus)
@End(Figure)


@Section(Summary)
This chapter presented an overview of Softwhere's command menus.
The diagram representing them is reproduced below for convenience:
@Begin(Figure)
              +----------------------+
              |      MAIN   MENU     |
              +----------------------+
@Tag(recapfig)
@Caption(Recap of Command Menus)
@End(Figure)

TEMPEST  main:MANUAL.MSS
```

Screen 2-6

The Number Constraint

The number constraint ensures that the contents of a block is a number. In particular, only non-negative real numbers in standard form are accepted. Thus, "02139" is unacceptable because leading zeroes are normally suppressed when writing numbers.

In this screen the user returns to the text first seen in Screen 2-1 and attempts to fill the **version-num** block. The user accidentally holds down the shift key and types a non-numeric value, and is warned with a message. Note that this operation fills the block, causing it to be unframed when the cursor is moved to the next line.

```
C-Z G Goto? version-num<CR> !#% C-N Non-numeric value entered.
```

```
@Make(Manual)
@Begin(TitlePage)
{|title-env}
@End(TitlePage)

Note:  This manual applies to Softwhere Version !#%.
□


TEMPEST  main:MANUAL.MSS
```

Screen 2-7

2.4 Flexible Viewing Commands

Flexible viewing refers to the ability to see the text from different points of view. In particular, it means modifying the display to reflect the structure of the text. Since the blocks and groups defined in a piece of text provide an alternate way of conceptualizing it, one sometimes wants to focus on this structure, rather than the text's content. The flexible viewing commands make this possible.

HILIGHT

This command underlines the text in the designated block and goes to the block, as with the GOTO command. If a group is named rather than a block, TEMPEST underlines the text in all member blocks, and forces the one that occurs first in the buffer to appear on the screen.

HILIGHT is useful for viewing structure that should be seen in context, for example, figure captions. A side-effect of this command is that it modifies the way the NEXT command works. After HILIGHT has been executed, NEXT causes the cursor to move to the next hilighted block. Thus, these two commands work in tandem to let one focus attention upon a set of blocks and then browse through the set.

In Screen 2-8, the user has requested TEMPEST to HILIGHT the group `figure-captions`. As a result, the text of all blocks in this group is underlined. Currently, not all of them are visible. As others become visible, they, too will be underlined. Note the display has been adjusted to make the block visible on the screen.

Hilighting can be removed with the UNHILIGHT command. This also causes NEXT to revert to its normal mode of operation.

```
C-Z H Hilight? figure-captions<CR>
```

```
                    (1)    Display on Screen
                    (2)    Send to Printer
                    (3)    Write to File

                         Choice:   ( )
@Tag(outputfig)
@Caption(Options Available in the Output Menu)
@End(Figure)

Below is a diagram of Softwhere's command menus and their relationship:
@Begin(Figure)
             +----------------------+
             |      MAIN   MENU     |
             +----------------------+
@Tag(recapfig)
@Caption(Relationship Among Softwhere's Command Menus)
@End(Figure)

TEMPEST  main:MANUAL.MSS
```

Screen 2-8

SHOW

The SHOW command is analogous to HILIGHT in that it focuses on a particular block or group. Whereas the HILIGHT command leaves the designated block in context, SHOW removes it from context and displays it in a separate "SHOW" buffer. SHOW consolidates on one screen text that is likely to be distributed throughout the buffer.

The important feature of the SHOW buffer is that it contains blocks which can be edited directly. When the SHOW buffer is exited (via the EXIT SHOW command), its blocks are placed back into the original text, *all changes intact.* Effectively, this command makes it possible to block out the text one doesn't want to see, edit the relevant text, and then restore the display to normal.

In this screen, the user wishes to focus on the members of group `figure-captions`. Notice that horizontal lines (`"====="`) divide the screen into three parts. Each part contains one block from the group being shown. The lines are simply visual cues to separate the various blocks on display. They do not demarcate separate editing windows -- this is a one-window display.

Notice on the dialogue line that the user first executes UNHILIGHT.

```
C-Z U Unhilight C-Z S Show? figure-captions<CR>
```

```
@Caption(Options Available in the Output Menu)

==========

@Caption(Relationship Among Softwhere's Command Menus)

==========

@Caption(Recap of Command Menus)□


TEMPEST  SHOW:
```

Screen 2-9

In this screen, the user edits the contents of the SHOW buffer. The first caption now reads "Output Menu Options".

Direct Editing

```
@Caption(Output Menu Options)

=============================================================

@Caption(Relationship Among Softwhere's Command Menus)

=============================================================

@Caption(Recap of Command Menus)



TEMPEST  SHOW:
```

Screen 2-10

EXIT-SHOW

In this screen, the user has returned to the original buffer via the EXIT-SHOW command. The changes made in the SHOW buffer have migrated to this buffer.

```
C-Z E Exit-Show
                    (1)    Display on Screen
                    (2)    Send to Printer
                    (3)    Write to File

                         Choice:   ( )
@Tag(outputfig)
@Caption(Output Menu Options)
@End(Figure)

Below is a diagram of Softwhere's command menus and their relationship:
@Begin(Figure)
             +----------------------+
             |      MAIN   MENU     |
             +----------------------+
@Tag(recapfig)
@Caption(Relationship Among Softwhere's Command Menus)
@End(Figure)

TEMPEST  main:MANUAL.MSS
```

Screen 2-11

OUTLINE

This command displays a list representing the currently defined group relationships, followed by a list of all blocks in the buffer in order of their appearance.

The list of groups is ordered on the basis of the first block in each group. Indentation is used to represent explicit containment.

In the list of blocks, indentation is used to reflect textual containment of one block in another. Top-level blocks (i.e., those not contained in any other) are always listed in the leftmost column. Unnamed blocks are represented by an asterisk ("*").

This screen presents the outline for the current buffer. Note on the mode line that this display is presented in its own buffer, named **OUTLINE**.

C-Z O *Outline*

```
EXPLICIT GROUPS:
title-env
figures
    figure-captions
    figure-bodies

BLOCKS:
titlpage
    title-text
    author-name
    author-address
i/o-example
*  (figure-captions)
menu-tree
*  (figure-captions)
tree-recap
*  (figure-captions)
□

TEMPEST OUTLINE:
```

Screen 2-12

FRAME

FRAME toggles a mode in which blocks are displayed *framed*, that is, as they were originally typed, with the boundaries indicated by the "{" and "}" characters and the name and default inside the braces. Empty blocks are always displayed framed, so this command only affects the appearance of filled blocks.

Although FRAME can be used simply to remind oneself of the names and locations of blocks, its primary use is to control the effect of an insertion which takes place on the boundary of a block by making explicit the destination of each character typed. (The problem of inserting on a block boundary was discussed in Chapter 1.)

In Screen 2-13 three blocks have been framed: `i/o-example` and the two figure captions, which are unnamed.

```
C-Z B<CR> Return from Outline Buffer C-Z { Frame

                    (1)    Display on Screen
                    (2)    Send to Printer
                    (3)    Write to File

                         Choice:    ( )|i/o-example}
  @Tag(outputfig)
  {@Caption(Output Menu Options)|}
  @End(Figure)

  Below is a diagram of Softwhere's command menus and their relationship:
  @Begin(Figure)
               +----------------------+
               |      MAIN   MENU     |
               +----------------------+
  @Tag(recapfig)
  {@Caption(Relationship Among Softwhere's Command Menus)|}
  @End(Figure)

  TEMPEST  main:MANUAL.MSS
```

Screen 2-13

2.5 Reuse Commands

TEMPEST's reuse commands are organized around the idea that users frequently edit the same kinds of text. For example, business letters, memoranda, proposals, and reports are standard fragments that come up repeatedly. The concept behind reuse is to represent these fragments as files of structured text which can be instantiated when they need to be used.

These files of structured text are known as *templates*. A template is a skeletal form for a piece of text, a pattern in which some of the text is supplied and the rest must be filled in by the user. Templates, however, are not special objects in TEMPEST as are blocks and groups. In fact, they are simply text files containing structure. A special term is used for them because they are meant for a specific use, and consequently are thought of differently than regular text files.

Illustrated below is a simple example of a template. It represents a personal address, as might be used in a business letter. Notice that it contains several empty blocks. In templates, the empty blocks are called *slots*. The special term is used to suggest something empty that is to be filled -- slots are placeholders for the parts of a template to be filled in by the user. Because slots are just empty blocks, they are always displayed framed.

```
{|title} {|first-name} {|initial} {|last-name}
{|number-and-street}
{|city}, {|state} {|zip}
```

Although any piece of text with structure can be saved in a file and later inserted into the buffer, templates are a special paradigm because they allow reuse to be customized through filling slots. Slots are placeholders for text that is likely to change from one reuse to the next. Because of this, templates are not just "boilerplate" text. And although templates resemble forms, they are not static like forms are -- they can be edited freely. Furthermore, once a template has been modified to suit a particular purpose, it need only be saved under a new name in order to define another template.

INSERT

In this screen, the user has moved to the beginning of the manual with the GOTO command, and has used the INSERT command to fill block `title-env` with a standard title page template in file `TITLPAGE.MSS`. (The name is not spelled out because the operating system imposes an 8 character restriction on file names.) INSERT places the text of the designated file into the buffer at the cursor. This template contains four slots, `title-text`, `subtitle`, `author-name`, and `author-address`. The `author-name` slot contains a default; the others do not.

The INSERT command is useful for building a document up from smaller pieces. It can be used to insert any text file, regardless of whether that file contains structure. If the file does contain structure, those definitions become accessible.

Note that the user first executes FRAME to turn off framing.

```
C-Z { C-Z G Goto? title-env<CR> C-Z I Insert? titlpage.mss<CR>
```

```
@Make(Manual)
@Begin(TitlePage)
@Begin(Title)
{|title-text}
{|subtitle}
@End(Title)
@Begin(Author)
{|HDC Systems, Inc.|author-name}
@End(Author)
@Begin(Address)
{|author-address}
@End(Address)□
@End(TitlePage)

Note:  This manual applies to Softwhere Version !#%.


TEMPEST  main:MANUAL.MSS
```

Screen 2-14

FILL

To fill a slot is to put text into it. One way of doing this is the FILL command, which prompts for two pieces of information: the name of the slot to be filled, and the name of the file whose contents will be placed in the slot. FILL is equivalent to moving the cursor to the desired slot and executing the INSERT command.

In this screen the `author-address` slot is filled with file `PUBLADDR.TXT`, which contains the address at which the company produces technical publications. Since companies use different addresses to handle different forms of correspondence, it is advantageous to use a slot for the `author-address` rather than "hardwire" this text into the template.

```
C-Z F Fill Goto? author-address<CR> Insert? publaddr.txt<CR>
```

```
@Make(Manual)
@Begin(TitlePage)
@Begin(Title)
{|title-text}
{|subtitle}
@End(Title)
@Begin(Author)
{|HDC Systems, Inc.|author-name}
@End(Author)
@Begin(Address)
HDC Systems, Inc.
P.O. Box 9999
Cambridge, MA  02139
Attention: Technical Publications□
@End(Address)
@End(TitlePage)

Note:  This manual applies to Softwhere Version !#%.

TEMPEST  main:MANUAL.MSS
```

Screen 2-15

ACCEPT

This command is for keeping a default. It unframes the slot, leaving behind only the default value. Since the default mechanism is purely syntactic, the acceptance process simply edits out everything but this text. To override a default, one types the desired value in front of the first vertical bar. When the block is exited, it will be unframed and the default will disappear.

The primary reason TEMPEST provides the ability to specify and accept a default is convenience. The purpose of templates is to reduce work by allowing reuse. It would be counterproductive if each reuse resulted in extra typing to account for text that is unlikely to change anyway. Another reason for defaults is to indicate a slot's intended contents.

In this screen the default for the `author-name` slot is accepted.

```
C-Z X Accept? author-name<CR>
```

```
@Make(Manual)
@Begin(TitlePage)
@Begin(Title)
{|title-text}
{|subtitle}
@End(Title)
@Begin(Author)
HDC Systems, Inc.□
@End(Author)
@Begin(Address)
HDC Systems, Inc.
P.O. Box 9999
Cambridge, MA  02139
Attention: Technical Publications
@End(Address)
@End(TitlePage)

Note:  This manual applies to Softwhere Version !#%.


TEMPEST  main:MANUAL.MSS
```

Screen 2-16

DELETE

The DELETE command, when applied to a block, deletes it along with all blocks it contains. When applied to a group, DELETE deletes all member blocks, and all blocks of all member groups.

In this screen, DELETE is used to remove the **subtitle block.**

```
C-Z D Delete? subtitle<CR>
```

```
@Make(Manual)
@Begin(TitlePage)
@Begin(Title)
{|title-text}
@End(Title)
@Begin(Author)
HDC Systems, Inc.
@End(Author)
@Begin(Address)
HDC Systems, Inc.
P.O. Box 9999
Cambridge, MA  02139
Attention: Technical Publications
@End(Address)
@End(TitlePage)

Note:  This manual applies to Softwhere Version 1#%.

TEMPEST, main:MANUAL.MSS
```

Screen 2-17

2.6 Structural Manipulation Commands

Structural manipulation refers to the rearrangement or replacement of blocks and groups. This is the least well developed area of application, containing only two commands. Nevertheless, commands of this nature can be very powerful because they combine many steps into one, thereby effecting a big change with only a little user effort.

REPLACE

The first representative of this area is the REPLACE command. REPLACE prompts for the name of a block and the name of a template. The contents of the block are replaced by the contents of the template, with the following effect (assume "A" is replaced with "B"):

* the text of A is deleted, as are any blocks contained within A;

* template B is inserted in place of A;

* any blocks in template B which bear the same name as blocks that were contained in A retain whatever text was originally in the A blocks;

* any other blocks in template B remain unfilled.

REPLACE is illustrated on the following page.

In this screen the user has replaced the **titlpage** block (from the TITLPAGE.MSS template) with a template named TECHTITL.MSS which represents a title page to be used especially for technical publications. Note that the **author-address** slot has retained the same value, as has the **title-text** slot (i.e., no value at all). The **author-name** slot, however, has disappeared entirely, since template **techtitl** does not contain such a slot. Note also the appearance of a slot not seen before: **date**. This slot has no counterpart in the block that was replaced, so it remains unfilled.

```
C-Z R Replace? titlpage<CR> With? techtitl.mss<CR>
```

```
  @Make(Manual)
  @Begin(TitlePage)
  @Begin(Title)
  {|title-text}
  @End(Title)
  @Begin(DocumentDate)
  {|date}
  @End(DocumentDate)□
  @Begin(Address)
  HDC Systems, Inc.
  P.O. Box 9999
  Cambridge, MA  02139
  Attention: Technical Publications
  @End(Address)
  @End(TitlePage)

  Note:  This manual applies to Softwhere Version !#%.



  TEMPEST  main:MANUAL.MSS
```

Screen 2-18

ANOTHER

This command copies the designated block and places the copy at the cursor. The copy includes any blocks contained in the original. ANOTHER does more than simply create new instances of the blocks it copies; it also copies their attributes. Thus, the copies are made members of the same groups and are given the same constraints as their original counterparts. They also retain the same names.

ANOTHER is equivalent to copying a block with COPY-REGION and yanking it back into the text. In this screen, it is used to reproduce the "Output Menu" figure at some other location in the Softwhere reference manual.

```
C-Z + Another? i/o-example<CR>
```

```
  Recall the form of the Output Menu:


                    (1)    Display on Screen
                    (2)    Send to Printer
                    (3)    Write to File

                         Choice:   ( )□
```

```
  TEMPEST  main:MANUAL.MSS
```

Screen 2-19

This empty page was substituted for a blank page in the original document.

3. Implementation Issues

3.1 The Host Environment

TEMPEST was built on an IBM Personal Computer by extending and modifying a commercially available editor called MINCE [Mince 81]. MINCE is a partial EMACS implementation, written in C. Extending an existing editor rather than building a new one allowed most of the time and effort to be focused on developing the ideas behind TEMPEST. Very little time had to be spent preparing an environment in which to experiment with them. The drawback of this approach is that MINCE's existing data structures and routines were not always well suited for supporting TEMPEST's features. They were sufficient to develop a prototype, however.

3.2 Representing Structure

TEMPEST's representation of structure is simple. A block is represented by two two-byte delimiters embedded in the buffer as follows:

```
<id><START> ... <id><END>
```

The first byte of each delimiter, represented as "<id>" in the illustration, is an identifier for that block. The id values span the range between 128 and 255, decimal. (MINCE actually has a printed representation for all values between 0 and 255, but the buffer display routines were modified to give special treatment to the characters beyond 127.) The second byte of each delimiter, represented as "<START>" and "<END>" in the illustration, marks the start or end of the block.

Each buffer in TEMPEST maintains a data structure called its "entity table" that stores information about the blocks and groups defined in that buffer. For a block, the identifier stored in the first byte of its buffer delimiter is also an index to its entry in the entity table. The table also contains an entry for each explicit group. Groups, like blocks, can be uniquely identified by their table index.

Each table entry contains several pieces of information. For a block these include its name and a list of the constraints it is included in; for a group, its name and a list of its members. Because blocks are explicitly delimited in the buffer, it is not necessary to store location information, such as an offset, for example. A block can be found by searching the buffer for its identifier.

When an object is deleted, its table entry is marked "deleted," but is not reclaimed. Since each entity table has a fixed size, this places a sharp restriction on the number of objects definable in TEMPEST. Furthermore, TEMPEST does not reuse identifiers. Thus, even with arbitrarily large tables, there would still be a fixed number of blocks and groups definable. These decisions were made in order to simplify the implementation, and are acceptable for the purposes of a prototype. A real implementation, however, would require larger tables, a larger identifier space, and some mechanism for garbage collecting used resources.

An approach that embeds characters in the buffer to delimit blocks is not an intuitive first choice. It means considerable modification to the editor, both to prevent the delimiters from being deleted, and to ensure that the display routines work correctly in spite of them. An alternative approach is to maintain a table of block locations. This approach is very good for finding a block in the buffer, because the table stores buffer offsets. To put the point in a block, one simply assigns the current offset to the point variable. The delimiter approach is much slower in this instance, since one must search the buffer to find the correct identifier byte.

Although the location table scheme is good for finding blocks, it is poor overall because of the overhead required to track blocks as their offsets change. Every insertion and deletion affects the offset of every block that comes after it in the buffer. A table of locations would have to be modified to account for these insertions and deletions, including operations that kill or yank an entire region. Contrast this with the delimiter scheme. Because the delimiters are in the buffer, there is no tracking to be done at all -- it is

handled as a side-effect of ordinary editing. There is only the relatively small overhead of protecting the delimiters from deletion.

The delimiter scheme was chosen over the table scheme primarily due to the difficulty of tracking in the latter. The drawback is not only in the difficulty of managing the table, but also in the frequency with which it would have to be updated. If almost every keystroke required a table update, TEMPEST would be virtually unusable. Although the delimiter scheme is slower at finding blocks, this only has to be done when executing some TEMPEST commands, and not after every insertion or deletion.

3.3 Auxiliary Files

In order for TEMPEST to remember structure from one session to another, it must be able to reconstruct both the contents of a buffer and that buffer's entity table. Toward this end, TEMPEST writes an auxiliary file each time a buffer is saved to disk. This file bears the name of the text file that was being edited, but has a different extension (".TSF", for "TEMPEST Structure File"). When TEMPEST reads a file, it also looks for a ".TSF" file; if none is found, the text is assumed to have no structural definitions.

The auxiliary file has two parts. One is the entity table for the buffer that was saved. The other is a list of the location and value of every delimiter in that buffer. The reason this information must be recorded in the auxiliary file is that the delimiters cannot be saved with the text. If they were, other programs would see them as garbage when accessing the user's files.

Auxiliary files have the obvious limitation that if they are deleted, all structural knowledge goes with them. This would not be such a problem if the files could be protected, but the personal computer operating system (PC-DOS, in this case) is not that sophisticated. Furthermore, because the structure information is in an auxiliary file, a text file cannot be modified by any other programs. On the other hand, maintaining structure information independent of the text file does allow that file to be read by other programs. This advantage far outweighs the potential disadvantages, since without it, it would be impossible to compile, format, or even print one's text files.

An interesting problem arises when a file of structured text is inserted into a buffer that already contains some structural definitions. Its auxiliary file contains the entity table and identifiers that were active at the time it was saved. These may overlap with the identifiers in the current buffer, however. The solution is to determine the next identifier to be assigned in the current buffer and rename all the saved identifiers as they are read in. This also means modifying all the entries in the entity table as they are read in.

3.4 Constraints

TEMPEST's constraints are action routines which are triggered by operating on blocks. The difficulty in designing the constraint mechanism was in deciding what sort of operations on blocks would trigger constraints. The solution chosen was to adopt two kinds of constraints: "enter" constraints, which fire when the cursor enters a constrained block, and "exit" constraints, which fire when the cursor leaves a constrained block. A block can have either or both kind of constraint.

In part, reduction of overhead motivated this solution. The overhead comes from monitoring every keystroke typed to determine which block, if any, it applies to. Because blocks are part of the text, editing operations are constantly modifying them without the user ever issuing a direct command to do so. Thus, each keystroke must be screened to see if it applies to a constrained block. The enter-exit scheme provides a reduction in this screening due to the following observations:

* most keystrokes will be character insertions;

* insertion cannot cause the cursor to cross a block boundary.

The consequence of these two facts is that, by being sensitive only to boundary crossings, it is not necessary to screen a majority of the keystrokes typed. This is a significant gain, since one must still screen all cursor movement and deletion commands.

A further motivation for triggering constraints on boundary crossings is the issue of consistency. While it is being edited, a block is likely to pass through many inconsistent states which would often force needless warnings, particularly from syntax checking constraints.

TEMPEST maintains a list of all constraint relationships. Each time ATTACH-CONSTRAINT is invoked, an entry is made in this list noting which constraint routine to use and which blocks to apply it to. For each block that is a partner in the constraint, a pointer is maintained to this list entry so the correct routine can be run if the block is operated on. If a block is a partner in more than one constraint, all of its routines will be run when it is operated on.

Constraint routines are written in C as part of the editor's source code. Thus, the constraint facility in the prototype is not extensible without access to the source code. Chapter 4 discusses the possibility of supplying a pseudo-language with which users could write constraints.

3.5 The Delete Buffer

The delete buffer is where MINCE keeps text that has been deleted by such commands as KILL-WORD, KILL-LINE, KILL-REGION, etc. It is possible for all of these commands to delete text that contains blocks. TEMPEST has to keep track of which blocks are in the delete buffer so it can do the right thing if they are yanked back into the text.

It is unfortunately not possible to allow blocks to be yanked as if they were raw text. Some monitoring is necessary because blocks are delimited by identifiers that must not conflict. Yanking a block from the delete buffer involves the following steps:

* if the block's entity table entry is marked deleted, then assume the block is just being moved -- reinsert it with the same identifier and remove the deleted status from its table entry;

* if the block's entity table entry is *not* marked deleted, then the block has already been yanked -- reinsert it with a new identifier and make a new table entry for it;

  Also, make this instance of the block a member of all the groups the original version was a member of, anticipating that the user meant to copy the block by yanking it more than once;

* if the block is being yanked into a different buffer than it came from, give it a new identifier and make an entry for it in the destination buffer's entity table.

TEMPEST maintains a data structure, called the "delete buffer table," in which it records enough information about all blocks in the DELETE buffer to determine their buffer of origin and the index of their entity table entry in that buffer.

In order for TEMPEST to determine when a block is migrating to the delete buffer, it must scan the text of the region to be deleted. If a block is completely contained in this region, it is allowed to pass to the delete buffer and an entry is made for it in the delete buffer table. If, on the other hand, only one of a block's delimiters is contained in the region, that delimiter is not deleted. Instead, it is kept in the buffer and the rest of the region is migrated to the delete buffer without it. TEMPEST must perform this check to prevent the user's buffer from containing unbalanced block delimiters.

3.6 Command Interface

All TEMPEST commands are grouped under the prefix character C-Z, and are invoked as the C-X commands would be. This is the standard EMACS command interface.

3.7 Buffer Markers

TEMPEST uses special marking characters in the buffer to delimit blocks and to indicate which blocks should be underlined due to the HILIGHT command. This section outlines the modifications necessary to support markers.

To begin, the display routines had to be modified to work correctly with markers. Most of the solution was in fooling the display routines into thinking that markers had zero width and did not print. That modification allowed markers to be ignored. Another modification was required to cause them to be significant in the case of underlining, where they were used as toggles to turn it on or off. For this, the redisplay algorithm was forced to scan for these toggling markers, even in lines that did not need to be redisplayed, in order to produce correct underlining.

Markers did not require only display modifications, however. In fact, almost every routine that operates on the buffer had to be modified. Deletion routines, for example, had to be altered so as not to delete markers (except, of course, when both of a block's delimiters were contained in the region to be deleted). Routines that move the point had to be altered to treat a series of marker bytes as a unit. For example, there are no insertions allowed between the bytes of a block's two-byte delimiters.

3.8 The Show Buffer

The show buffer is where TEMPEST displays blocks selected by the SHOW command. Each selected block is copied to the show buffer, and an entry is made for it in the "show buffer table." When the EXIT SHOW command is issued, TEMPEST searches the show buffer for every block listed in the show buffer table, and copies each one back to the originating buffer, deleting the old version and replacing it with the version from the show buffer. Thus, any changes made to a block in the show buffer will appear in the originating buffer.

TEMPEST does not simply treat the show buffer contents as raw text. The show buffer contains blocks which can be operated on by both structural and textual commands. This is possible because the show buffer uses the entity table of the buffer from which the SHOW command was executed. Thus, the show buffer blocks can be edited as if they were in their home buffer.

The use of a show buffer is an approximate solution to the SHOW command. Ideally, the SHOW command would not change buffers -- it would simply blank out the text not selected, leaving only the selected text visible for editing. Unfortunately, this is very difficult in MINCE. Although it is easy to prevent certain section of the buffer from being displayed, it is hard to prevent the point from ever entering those sections. Thus, the show buffer was chosen as a compromise solution.

A drawback of the show buffer is that constraints cannot operate there reliably. Constraint routines are written with the assumption that their argument blocks will all be in the same buffer. There is no guarantee that a block's constraint partners will appear in the show buffer with it. The current solution is to prevent constraints from triggering in the show buffer. A good extension would be to allow constraints to operate on more than one buffer.

3.9 Automatic Framing and Unframing

Normally TEMPEST displays empty blocks framed and non-empty blocks unframed. (An exception is when the FRAME command is used. Then, even non-empty blocks are framed.) When a block's status changes, its mode of display must change accordingly. TEMPEST accomplishes automatic framing and unframing through its own constraint facility.

To each block is attached an exit constraint which parses the block to determine if it has been left empty or non-empty. The routine then makes the appropriate display changes if any are called for.

4. Future Directions

Name Disambiguation

When dealing with named structures, name duplication is almost unavoidable. It could be legislated out of existence, but it would be burdensome for the user to have to name everything uniquely, or worse yet, to keep track of artificially generated names. TEMPEST allows duplication because it is most natural to apply the same name to the various instances of something. For example, it is reasonable that all blocks representing occurrences of the variable "X" should be named "X."

The problem with name duplication is, of course, that it results in ambiguous references. Although the current prototype does not implement any schemes for disambiguating references, a plausible one that would not require any major modifications is discussed below.

One approach to disambiguation is for TEMPEST to make some reasonable guesses, inform the user of these choices, and ask the user to pick one. Some possible heuristics for disambiguating a reference to block "X" are:

* if there is an X on the screen, and it is the only one on the screen, choose it;

* if there is no X on screen, or if there is more than one, choose the one nearest the cursor;

* if there is an X that has been previously referenced, choose that one -- perhaps "locality of reference" will succeed.

Clearly there are a large number of heuristics which vary in complexity -- these three simple ones are shown to suggest that this is a plausible scheme which could be implemented relatively easily. An extension to this scheme would be to allow the user to specify a pathname based on group membership or lexical containment; for example, "the foo in group bar" or "the foo within block bar".

Computed Groups

Chapter 1 introduced the notion of a computed group -- a group whose members are determined by computation rather than explicit declaration. Computed groups are not currently used very extensively in TEMPEST, but could be used effectively to extend the mechanism for referring to blocks and groups.

The current reference mechanism understands single names only, that is, one can request TEMPEST to "GOTO X" or "SHOW Y." It is not possible, however, to make more expressive references such as "ALL BLOCKS NAMED 'HEADING'." The only way to operate on all blocks named "HEADING," for example, would be to place them all in an explicit group. That is not only inconvenient, but is also prone to error -- the user is likely to forget to add a new "HEADING" block to the group. References of this kind are a natural application for computed groups.

This extension would involve a simple language for quantifying references. For instance, the example reference above could be expressed "ALL HEADING." If the keyword "IN" were included, one could construct more complex references, such as "ALL HEADING IN SECTION-3." Such a facility would have two advantages. First, it would reduce the number of explicit groups to be created by the user. Second, it would reduce the number of ambiguous references in command arguments.

Structural Manipulation

As has been stated, structural manipulation commands permit one to operate on several blocks at a time by consolidating many steps into few. The OUTLINE command, which is normally used only to view structure, could be extended to permit a particularly useful structural manipulation.

The OUTLINE command presents an abstract representation of the buffer in terms of blocks and the groups

they belong to. The extension would be to treat this display as a sort of map which could be edited. When one finished editing the map, any changes would be reflected in the buffer. For example, one could reorder two blocks by switching their order in the outline. This is a powerful alternative to directly moving the text of blocks. (This command could itself be extended to allow one to edit group relationships.)

In general, this would not be an easy extension to implement. The difficulty is in deciding what to do with the text between blocks. One could simply reorder the blocks and leave the surrounding text untouched. However, the text surrounding a block might be conceptually related to it, in which case it should move with the block.

An additional structural manipulation command might rearrange the slots in a specified region according to the format of a particular template. This would function identically to REPLACE, except that it would leave in place, rather than delete, any slots in the region with no counterpart in the template. This sort of command would allow the user to work out of order and later restructure according to a stored format.

Constraints

TEMPEST's constraint facility is currently limited by requiring constraint actions to be written as C functions and compiled as part of the editor. While this approach allows one to define any number of constraints, it has the drawback that one must have detailed knowledge of the implementation of the editor in order to write routines that correctly manipulate the buffer.

A more flexible approach would be to provide a pseudo-language and a set of primitives for manipulating the buffer that could be used to write constraint actions. The primitives would have to take block names as arguments, since this is how constraint routines work. A minimal set of primitives would have to provide operations to access and change the contents of blocks. There would also need to be more specialized operations such as converting strings to numbers, for example. Although this keeps constraints in the realm of the programmer, it improves on the current approach by eliminating the need to understand the inner workings of the editor. It also does not require access to the source code.

There are many constraints that could be written for TEMPEST. One such class could attempt to enforce certain syntactic and even semantic conventions, perhaps for programming or for a restricted class of documents. It would certainly be possible to emulate spreadsheet programs by writing mathematical constraints. Whereas spreadsheets have some difficulty in integrating text, it would be relatively easy for TEMPEST to integrate numerical constraints on slots.

A Template Library

Since templates are files of structured text, they are referred to by their file system names. While this is a workable solution, it can be improved by introducing a layer of abstraction between templates and the file system. This is the role that a template library would fill. In the library, templates could have arbitrarily long names, independent of file system restrictions. Furthermore, templates could be grouped hierarchically according to logical purpose. For example, programming templates for C could be under a separate node from the PASCAL templates. This would allow the existence of templates with the same name, just as files of the same name can exist in different directories. Also, the name of the actual file that represents a template could be changed without forcing users to keep up with such changes.

Adding a library would also mean adding some utilities. For example, one would need a way to tell TEMPEST where in the library tree to begin its searches (analogous to specifying a working directory). One would also need mechanisms for modifying and browsing through the library. There is a further complication -- the library insulates the user from the file system only during TEMPEST sessions, but cannot control access to it at other times. Thus, the library needs to be kept consistent with the file system.

A library would not be a trivial extension to implement, but would be a useful abstraction feature, particularly for managing a very large set of templates.

References

[Ciccarelli 84]
Ciccarelli, Eugene C..
Presentation Based User Interfaces.
Master's thesis, Massachusetts Institute of Technology, August, 1984.

[Engelbart 84]
Engelbart,Douglas C..
Authorship Provisions in AUGMENT.
Technical Report, Tymshare, Inc., 1984.

[Hammer 81]
Hammer, Michael et. al.
The Implementation of Etude, An Integrated and Interactive Document Production System.
In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 137-146. ACM, 1981.

[Mince 81]
Mince.
Mark of the Unicorn, Inc., Cambridge, MA, 1981.

[Nelson 81]
Nelson, T.
Literary Machines.
Published privately, 1981.
Copies may be obtained from Ted Nelson, Box 128, Swarthmore, PA 19081

[Pitman 85]
Pitman, Kent M.
CREF: An Editing Facility for Managing Structured Text.
Technical Report, MIT Artificial Intelligence Laboratory, February, 1985.
A.I. Memo #829

[Reid 80]
Reid, Brian K. and Walker, Janet H.
SCRIBE Introductory User's Manual.
Unilogic, Ltd., 1980.

[Stallman 81]
Stallman, Richard M.
EMACS Manual for TWENEX Users.
MIT Artificial Intelligence Laboratory, 1981.

[Stromfors 81]
Stromfors, O. and Jonesjo, L.
The Implementation and Experiences of a Structure-Oriented Text Editor.
In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 22-27. ACM, 1981.

[Teitelbaum 79]
Teitelbaum, Tim .
The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS.
Technical Report, Department of Computer Science, Cornell University, 1979.

[Walker 81]
Walker, Janet H.
The Document Editor: A Support Environment for Preparing Technical Documents.
In *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*, pages 44-50. ACM, 1981.

[Waters 82a]
Waters, Richard C.
Program Editors Should Not Abandon Text Oriented Commands.
SIGPLANNotices 17(7), July, 1982.

[Waters 82b]
Waters, Richard C.
The Programmer's Apprentice: Knowledge Based Program Editing.
IEEE SE-8(1), January, 1982.

CS-TR Scanning Project
Document Control Form

Date : 1/25/96

Report # AI-TR-843

Each of the following should be identified by a checkmark:
Originating Department:

☒ Artificial Intellegence Laboratory (AI)
☐ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 42 (49 images)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☒ Single-sided or
☐ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form (2) ☐ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages(by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description :	Page Number:
IMAGE MAP: (1-42) UN#'ED TITLE PAGE, i, 1-7 UN#BLANK, 9-31, UN#'ED BLANK, 33-40	
(43-49) SCANCONTROL, COVER, DOD(2), TRGT'S(3)	

Scanning Agent Signoff:
Date Received: 1/25/96 Date Scanned: 1/31/96 Date Returned: 2/1/96

Scanning Agent Signature: Michael W. Cook

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 843	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) TEMPEST: A Template Editor for Structured Text		5. TYPE OF REPORT & PERIOD COVERED Technical Report
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Peter J. Sterpe		8. CONTRACT OR GRANT NUMBER(s) In part by ARPA, NSF, IBM and Honeywell, Inc.
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE June, 1985 13. NUMBER OF PAGES 42
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

text editors, structured text, templates, reuse

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

TEMPEST is a full screen editor that incorporates a structural paradigm in addition to the more traditional textual paradigm provided by most editors. While the textual paradigm treats the text as a sequence of characters, the structural paradigm treats it as a collection of named *blocks* which the user can define, group, and manipulate. Blocks can be defined to correspond to the structural features of the text, thereby providing more meaningful objects to operate on than characters or lines.

(continued on back page)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Continuation of Abstract:

The structural representation of the text is kept in the background, giving TEMPEST the appearance of a typical text editor. The structural and textual interfaces coexist equally, however, so one can always operate on the text from either point of view.

TEMPEST's representation scheme provides no semantic understanding of structure. This approach sacrifices depth, but affords a broad range of applicability and requires very little computational overhead. A prototype has been implemented to illustrate the feasibility and potential areas of application of the central ideas. It was developed and runs on an IBM personal computer.

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 1/31/1996

M.I.T. Libraries
Document Services