IncTeX : An Incremental Document Processing System *

Michael A. Harrison
Derluen Pan
Computer Science Division
University of California
Berkeley, CA 94720

April 1, 1991

Abstract

IncTeX is an incremental document processing system for TeX documents. The system is editor independent and has been used in a variety of applications such as VorTeX. The formatter state is periodically compressed and checkpointed to allow incremental processing. Pages are selectively reformatted by analyzing the input and doing state comparison to decide when to halt.

*Supported by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contract N00039-88-C-0292.

1 Introduction

INCTEX is an editor-independent incremental formatter which combines the formatting capabilities of batch formatters with the incremental processing of direct manipulation systems. Batch formatting systems such as TEX [Knu84a, Knu86] provide sophisticated formatting algorithms but must reprocess the entire document each time even if all the changes are on a single page. Direct manipulation document systems such as Lilac [Bro88], InterLeaf [Int85], and VORTEX [CCH+86, Che88, CH88], provide interactive, incremental response. WYSIWYG (what you see is what you get) direct manipulation systems display and allow editing of the document in its finished form and the changes are reformatted incrementally. Direct manipulation systems rely on a directly coupled editor which identifies document changes. Interactive response is provided by incrementally reformatting only what is necessary and by giving processing priority to visible regions. However, interactive response in direct manipulation systems exacts a price in the form of less sophisticated formatting, fewer features, and less flexibility than batch formatting.

INCTEX occupies a new location in the spectrum between batch formatting and direct manipulation systems. It reformats incrementally but is not directly coupled to an editor. Any text editor can be used to edit the input document, as with batch formatters. Formatter execution is checkpointed and formatting is restarted where there are changes and stopped when the formatting of the next region will be identical. Checkpoints of the formatter state are compressed by taking advantage of state change characteristics which result from fundamental document use patterns.

INCTEX is based on TEX, a widely used document system which provides excellent formatting and a powerful macro language. It provides the highest quality linebreaking available by computing optimal linebreaks an entire paragraph at a time [KP82]. Hyphenation is also very sophisticated and achieves high quality with relatively small memory use [Lia83]. The macro language is powerful enough to create document styles ranging from letterheads to professional mathematical journals and books. TEX provides file inclusion so that input can be read from multiple files and also provides the ability to write arbitrary text to multiple auxiliary output files.

The VORTEX system [CCH+86], to which INCTEX is related, is a page-level incremental WYSIWYG document editing system based on TEX. The VORTEX system provides direct manipulation editing on the formatted view of the document as well as the source language view, but does not compress state checkpoints nor provide a comprehensive solution for quiescence checking. InforTEX is a non-incremental document editing system which includes special editor support for TEX document entry and provided document previewing [Sch87].

The development of an editor-independent incremental formatter requires the solution of several problems. Formatter output must be incrementally generated. Input has to be analyzed to find changes and an appropriate point must be chosen to restart formatting.

Execution has to be checkpointed and restarted. Determining when reformatting can be discontinued and output will be identical is another serious issue.

INCTEX will discontinue reformatting if it determines that it has reached a point in the document where the remainder of the formatter output will be the same as before. This condition is termed *quiescence*. TEX was not designed to provide for quiescence detection and has no mechanism for tracing formatting data dependencies which would support quiescence detection. However, another solution can be found by viewing the formatter as a deterministic automaton whose processing (formatting) is distributed over time. Execution points can be associated with particular program states and output points (page breaks) and input points. The execution of the machine from any point into the future is determined by the program state and remaining input. Quiescence can be identified if an execution point, composed of a program state and the remaining input, ever repeats an execution point from a previous session.

The extent to which comparing formatting states will be equivalent to checking for quiescence depends on the degree to which the state is free of unnecessary dependence on the input encountered so far. If it depends too much on the previous input, any trivial difference in the input will result in a state difference and quiescence will never be detected. In INCTEX, this technique allows quiescence to be successfully detected in cases where the number of lines on a page remain unchanged and the document structure is not altered (for example, no sections are added or removed). This success is directly dependent on the design of TEX, which reads input lines until it reaches the end of a paragraph, determines paragraph linebreaks, until it decides a paragraph should not be placed on the remaining page and a page break should be made (which may actually precede the last input line by some distance). This means that linebreaking data structures do not hold information necessary for future formatting (at a page break) and do not need to be compared for quiescence detection. The result is that quiescence detection does not depend on the exact linebreaks (and thus words) on a page. However, if the number of lines does change, then the set of pending lines for the next page saved in dynamic memory *will be* different and the algorithm will correctly detect that quiescence has not occurred yet. On the other hand, document structure is encoded in a table which *is* necessary for future formatting and cannot be ignored. State comparisons fail even if a structural change is made which will not change the appearance of future pages, for example, if a subsection is added to the end of a section, but all later pages still have the same sections and text. This type of change causes the quiescence check to fail unnecessarily, at least until an appendix appears and the structural labels are redefined. One conclusion that can be drawn from the INCTEX project is that, for more general quiescence detection, the formatting data dependencies themselves must be checked and the formatter must be designed to identify and trace these data dependencies. Quiescence should be detected by checking dependencies instead of comparing states.

2 Design Of INCTEX

INCTEX was adapted from TEX by treating it as a deterministic multi-tape Turing machine which is checkpointed and restarted at various execution points during processing. INCTEX is page incremental; checkpoints are made at the execution points where output pages are produced. TEX pages are the natural unit of granularity. Input is processed until a page break is determined and then a description of the formatted output page is generated.

INCTEX maps execution points at page breaks to locations in the input and output by recording the current input and output file offsets at page breaks. A change at any particular point in the input can be matched to an execution point where formatting diverges from the previous version. INCTEX compares the new document against the previous version for changes and locates a page checkpoint which precedes the changes from which to restart. Input file offsets and output file contents are restored to their state at that checkpoint and formatting is restarted.

This approach to incremental processing can be applied to any deterministic program where the processing is distributed over the course of reading the input. If multiple passes over the input are made, execution must be restarted at an execution point during the first pass and sufficient processing must occur during the first pass so that the savings justify the overhead.

Several problems must be solved in order to restart TEX incrementally. First, formatter output must be incrementally generated. It was fairly simple to divide the formatted device-independent (DVI) output file into separate DVI files for each page. For details about the structure of DVI files, see [Fuc81, Fuc82]. The input document must be analyzed for changes and an appropriate point must be chosen to restart formatting. A method of checkpointing the formatter execution state and restarting it must be developed. The final problem which must be solved for incremental processing is determining when reformatting can be discontinued.

Analyzing the document for changes and locating an appropriate restart point is complicated by line buffering and nested input files. Nested input files require that multiple input files be analyzed for changes. In order to decide where to restart, INCTEX must treat a change in a file as if a change has also occurred in the line in the file where the command to include the file is found.

Compression techniques allow state checkpointing to be performed more efficiently. The state which must be saved to restart formatting for the next page occupies 400 Kbytes of memory. Analysis shows that macro definitions, font descriptions, and style parameters rarely change during TEX formatting (which is probably true as well in other formatting systems). INCTEX uses compression techniques which rely on this behavior.

Reformatting can be halted when the formatter has reached a point where the rest of the formatting output will be the same as before. Detecting this situation, termed *quiescence*, was not fully solved in VORTEX because of the potentially unrestricted scope of side-effects due to macro redefinitions. The solution taken in INCTEX is that document formatting can be guaranteed to be identical if the formatter state has reconverged to a state identical to one in the previous document (this is a sufficient but not theoretically necessary condition; it is over restrictive). As new pages are formatted, the state checkpoints are compared against the previous checkpoint for the same page. Execution is halted if quiescence is found. The next change in the remaining input, if any, is located and formatting is restarted. The largest problem in verifying quiescence is the internal dynamic memory management in TEX. INCTEX parses the contents of dynamic memory to determine if two states are equivalent.

The initial version of INCTEX was derived from VORTEX. This technical report describes work which was performed to develop a quiescence detection algorithm and improve state compression from 4–1 to 40–1. Checkpoint size varies depending on document complexity, but can occupy as little as 10 Kbytes in the best case, despite the fact that TEX's state occupies 400 Kbytes (a complex page can occupy up to 150 Kbytes). Input document analysis was also refined so that fewer files are copied. The standard TRIP acceptance test for TEX [Knu84b] was also generalized to an incremental form since validation is a large part of any software project (see Appendix D).

The design of INCTEX allows the reuse of existing code for a TEX formatter to develop an incremental formatter. The techniques used to accomplish the initial part of this was reported in [CHM88]. Most of INCTEX consists of external code to the formatter itself which allows state to be saved and restored and checked for quiescence. A few minor internal changes were made to allow execution to be halted and restarted at each page break and to allow state changes to be recorded for state compression. INCTEX is based on a public domain C language version of TEX written by Pat Monardo. The original system was written in PASCAL with the WEB software design system [Knu83, Knu86]. TEX consists of about 25,000 lines of C code, and INCTEX consists of approximately 10,000 lines of additional C code.

We will describe INCTEX by first explaining how the input files are analyzed to locate changes and determine the correct restart point, describing formatter state checkpointing and compression, quiescence detection, and finally explain how these solutions are combined for incremental formatting.

3 Input Analysis

Before any incremental processing can be performed, INCTEX must identify the changes in the document and determine where to restart formatting. The information needed to do this is recorded in an incremental page-mapping file which maps each page to a point in the input and the output file. At the end of each page, the input file which is currently being read is recorded. The offset describing the last line which was read is also recorded (all output files and offsets are also recorded). It also records the time each file was last modified, the parent of each file (the file from which it is included), and the end of the line in the parent file where the include command occurs is recorded in the page mapping file. Each input file is also copied. The important problem for input analysis is that the last line has been read into the buffer but may not have been fully processed yet. However, changes in the set of input files are always reflected as changes in some of the original input files because each file to be read has to be specified somewhere, starting in the original source document file.

When a new document is submitted, INCTEX finds out which files have been modified since the last time. It compares these files against the previous versions and finds the offset where the first changes occur. If there is only one input file, it simply scans sequentially from the first to the last page looking for the first page whose input line offset exceeds the offset of the change. However, if there are nested input files, it must consider a change in one file as equivalent to a change to the parents of that file on the lines where the file is included. It must find the page where that file, or any line leading to that file's inclusion, is read. For each page, therefore, the system checks whether the end of the page exceeds the offset of the change on this file, the offset of the end of the line in the parent file, and so on for any succeeding parents. The checkpoint which precedes all changes to all files is loaded.

Copies of the input files are stored in a subdirectory called **INC** which also holds all other permanent information needed to incrementally reformat the document. Input files are copied as the same relative pathnames within the **INC** directory, so that files in different directories with the same leaf name will not collide. Files, timestamps, and page offsets are recorded in the incremental page mapping file, which is called **doc.inc** where **doc** is the root document name. The format of this file is described in Appendix C.

In order to reduce file copying, read-only files are not copied. The time stamp is recorded, and if the file is modified, the analysis algorithms treats it as if the entire file has changed. Files ending in **.sty** are also treated in this way, since by convention [Lam86, page 85] these are usually system or user style files which usually never change and almost always are read at the beginning. Since they are read at the very start and usually contain macro definitions, a change would normally force complete reformatting and this does not impair the ability to incrementally reformat.

4 State Checkpointing

INCTEX checkpoints the formatter execution state at different execution points so that formatting can be incrementally restarted. The execution state is very large, even when only the portion which must be carried over to the next page is considered. However, outside of the page description in dynamic memory, very little of this formatter state changes from page to page. These characteristics are explained in part by the way in which users write documents and in part by the design of TEX. Users rarely modify font characters or page dimensions and, in general, document style parameters rarely change in mid-stream even though TEX allows them to be altered. Macros are also rarely redefined. Often they are defined by a style package which is read from a system library before the user document is processed. These characteristics suggest that recording the changes to the state might be a compact way of describing the current state.

INCTEX describes the formatter state as changes relative to a reference state, instead of describing the state itself. The state after the initial style file is loaded is used as the reference state. Each state checkpoint is described as the difference relative to this reference point either by recording changes to data structures or by computing the difference. The style initialization point has a special advantage as the reference state since it can be created simply by loading the style file at the start of execution and therefore it does not need to be recorded by some special means. The state of any page is then restored simply by loading the differences recorded in the page checkpoint. Dynamic memory is the only region to which this difference technique is not applied. The only compression attempted on the dynamic memory region is the removal of unused areas.

The state required to continue formatting from page to page consists of about 400 Kbytes divided as follows:

1. Global variables, grouped into a single region.
2. A hash table which is add-only. Access is random, and few additions are made.
3. Symbol tables. Entries can be changed at random.
4. A stack for saving symbol definitions. The stack is simply saved.
5. Font description arrays. Segments can be added onto the end occasionally, and rarely a random font character description will be changed.
6. A string pool which is append-only.
7. Two regions of internally allocated dynamic memory:
    - node memory (contains the page description, etc).
    - token memory (contains macro descriptions).

Changes to randomly accessed entries in data structures (the hash table, symbol tables, and font arrays) are recorded in cumulative difference lists by inserting software probes into the code which modifies these entries. The current value of each different entry is written in the checkpoint. When a checkpoint is loaded, these difference lists are restored, as if the changes during processing of preceding pages had been logged.

Modifications to the font arrays are separated into changes to current entries, which are recorded in the difference list, and additions to the end of the arrays to describe new fonts, which are captured in a per-document file so that they will not be described redundantly in each page checkpoint. Any new additions are saved each time a checkpoint is saved and the current array limits are recorded in each checkpoint. The correct font state is restored by loading the initial segment values from the font files and then loading any changes from the font difference list.

String pool limits are similarly recorded in each checkpoint while the additions to the string pool themselves are saved in a separate per-document file. However, since entries are never changed, the string pool is just saved at the end of formatting.

The difference in the global variables relative to the initial reference state are computed by comparing the global region word by word against a copy of the initial global region. The difference is saved as a difference list like those for the hash or symbol tables.

The save stack, a small stack used to save symbol table definitions, is saved in its entirety.

Dynamic memory is scanned and the occupied regions are saved in the checkpoint. Token memory allocation was modified so the region allocated during style file initialization does not have to be saved. The free list is reset to the outer unallocated region after initialization so that no locations in the initial token region are reused. With this modification, the contents of this region never changes and does not have to be saved.

A state checkpoint thus consists of dynamic memory descriptions, difference lists, the save stack, and the string pool and font segment descriptions. The string pool and font segments are only saved once per document. The compression ratio varies depending primarily on the complexity of the page description and macro description. The uncompressed state occupies 400 Kbytes, and a minimum checkpoint occupies 10 Kbytes, giving a maximum compression ration of 40–1. Checkpoints can vary up to 160 Kbytes.

The overhead for state checkpointing consists of 12 Kbytes of memory to copy the initial global region. With current workstation processor speeds (SUN 4 or SPARC), the cost of performing a comparison is found to be outweighed by the eliminating the costs (communication overhead and disk delay) of writing 12 Kbytes to a network disk. Software overhead for keeping the difference lists is small, although the memory overhead for these lists is proportional to their size. There is very little memory overhead for compressing dynamic memory, but unoccupied regions have to be found by scanning these regions, which are very large.

Details on the implementation of state checkpointing are given in Appendix A.

5 Quiescence Checking

5.1 Overview

The goal of an incremental formatter is to reformat the least amount necessary to reflect new editing changes. Practical issues such as output regeneration granularity and the computational and storage overhead for any particular reprocessing algorithm can lead to an implementation which is incremental, but performs more than the theoretically minimum amount of reformatting.

The TEX system was not designed with incremental formatting in mind and does not compute data dependencies that can be used to determine what must be reformatted. Furthermore, the TEX language, especially the macro definition capability, contains complicated features which make it difficult to deduce such dependencies. The problem of deciding which pages to reformat involves two questions: where the processor must restart formatting, and when formatting will repeat itself. The input analysis which answers the first question (where to restart) has already been described. INCTEX decides when formatting will repeat itself by comparing the formatting state against the previous state for the corresponding page. If they are equivalent, then the formatter will repeat execution and generate the same output for as long as the input repeats itself (this condition is termed quiescence). INCTEX halts reformatting when it finds an equivalent state. It reanalyzes the input to find the next difference and restarts at the last page which precedes it.

The quiescence algorithm in INCTEX detects a sufficient (but not necessary) condition for quiescence. INCTEX checks for quiescence and saves a state checkpoint after a page break has been chosen, and the current page description has been written and removed from memory. A combination of three techniques are used: direct (binary) state comparison, state parsing and comparison for dynamic memory structures, and ignoring temporarily irrelevant variables. The quiescence detection algorithm will be described in this section and in greater detail in Appendix B. Suppose a resubmitted document is quiescent after some set of changes: the remainder of the input will be the same as before at the page break following the changes, and any document formatting will also repeat itself after the page break. TEX reads entire paragraphs at a time and performs linebreaking (determining the positions of words, spaces, and line breaks) using a box and glue model. An entire paragraph will have just been read and linebroken; the positions have been chosen for all the boxes corresponding to the characters will have been determined. The linebreaking data itself is temporarily irrelevant and can be omitted from state comparisons because paragraph linebreaking has just been finished. Thus, linebreaking differences up to this point will not affect quiescence because the linebreaking state is not compared. Usually there is also some set of lines which overflow onto the next page; the text and box positions for these lines are saved in a page description list kept in dynamic memory. The page description for the pending page is parsed and compared so that dynamic memory allocation differences will also be ignored. Therefore, quiescence detection will fail only if the document changes have induced a change in some other part of the state (such as in a symbol or macro definition or a special counter for section numbers).

In practical terms, quiescence detection succeeds if the changes are confined to the text in paragraphs (the number of paragraphs can be altered without any effect). Quiescence can also be detected when entries in tables are altered. However, adding or removing a subsection title in LaTeX documents causes quiescence detection to fail until an appendix appears. This occurs because section numbers are recorded in formatter counters, and changes involving counter variables lead to a state difference preventing quiescence detection. An appendix in LaTeX resets the section numbering and permits state comparison to detect quiescence. Quiescence also fails when macro definitions change because the state describing the macros changes.

State comparison has two fundamental limitations with regard to detecting quiescence. One limitation arises if an input change has a state side-effect, for example if a macro definition changes. INCTeX does not know what parts of the document are affected, or conversely, which parts of the document are unaffected. It must assume the worst case and reformat the rest of the document until the state returns to an identical state as before. In the case of a macro change, keeping dependencies would permit INCTeX to determine if and where any references to a macro occcur, and the regions of the document that must be reformatted. The second limitation is that quiescence can only be detected *if the remaining input is the same as before*, since INCTeX really compares finite machine states and a state consists of the formatter state and the input (and, technically, output) state. Specific examples of problems with state side-effects and input file dependency will be discussed at the end of this section to demonstrate how the implementation of the LaTeX style macros create difficulties for state-comparison based quiescence detection.

State comparisons are more complicated than simple comparison of the binary state values because of two problems, dynamic memory allocation and state compression. TeX contains its own privately managed dynamic memory pool. A different allocation sequence can cause the pool to be allocated differently in two states even though it contains the same contents semantically. INCTeX parses and compares the dynamic memory contents and the technique that allows pointers to dynamic memory items to be compared to decide if they point to equivalent items will be described. The second problem is that the current program state in memory must be compared against a compressed state description which actually describes changes to the state relative to an initial reference state. The mechanisms that perform this comparison will also be described.

Dynamic memory comparison is simplified slightly by the separation of dynamic memory into two separate regions in the implementation upon which INCTeX is based. Only the dynamic memory pool containing the page description is parsed, while the dynamic memory pool containing the macro definitions is simply compared for binary equivalence. Any macro changes are likely to result in failure to detect quiescence because of differences in memory allocation, but since there is no way to determine the side-effects of a macro change, quiescence would be difficult to determine even if dynamic macro memory was parsed and compared. The only practical gain which might be achieved would be the ability to detect quiescence if macros are defined in a different order. In practice, users rarely redefine macros. The decision not to parse macro memory does not therefore significantly impair quiescence detection and allows a reduction in the parsing requirements for quiescence checking.

The comparison of the various state regions will now described in further detail, as well, as how the remaining input is analyzed to detect quiescence. Details at the level of specific state variables will be given in Appendix B.

5.2 Global Variable Comparison

The global variables are grouped into a contiguous region and described by comparing them against the initial reference state values to create a difference list description. The way difference lists are compared will be given later; but a series of locations in the current state and the previous state are matched and compared. A variable in the current state is compared against a variable in the previous state checkpoint by binary comparison except if it is one of three special variable types: a dynamic node memory pointer, an ignoreable variable (a variable whose value does not need to be compared, such as a page breaking value which no longer matters after a page has been generated or a dynamic memory free list pointer), or the current file name string, which is only compared up to its current length. The addresses of these variables are entered into a special address list which is used to determine if a special comparison technique should be performed.

The entire global region is compared as if it were a series of single word items. When a difference is found, the half word where the difference begins is identified, because pointer variables to node memory actually occupy a half word, and a special address list is scanned to determine whether the location is a dynamic node memory pointer or other variable type requiring a special comparison technique. Each entry in the list specifies an address range and the type of comparison to be performed (to compare a dynamic memory pointer, ignore a variable, or check a name string). Comparison of dynamic memory pointers will be explained in the subsection on parsing dynamic memory.

Special treatment was required for global file descriptor pointers so that spurious differences would not be flagged. File descriptors are allocated from the UNIX dynamic memory and the descriptor pointers will have different values each time. INCTEX saves the pointers in temporary storage and clears them before state checkpointing and quiescence checking so the descriptor pointers will be identical. Input and output will still be compared correctly because file names and file offsets are checked explicitly during input analysis. The unused part of the global file name string buffer is also always cleared before quiescence checking because INCTEX uses the string buffer prior to restarting TEX and the buffer region past the end of the current string contains random characters.

5.3 Parsing Strategy

The contents of dynamic node memory is examined by traversing and parsing it as an ordered forest of trees. The contents are printed in a readable form to a temporary file and compared to the contents generated by parsing the previous checkpoint. Traversing the trees in dynamic memory in a strictly ordered manner allows any location in dynamic memory to be assigned a semantic meaning according to the order in which it is visited. No location is ever visited twice, although a visit order could still be defined which would allow this technique to be used. During parsing each dynamic memory location is assigned a number by its order in the forest traversal. Pointers to dynamic memory in the global variables or other data structures of two different states can now be compared by verifying that they point to nodes which have the same tree visit number.

Pointers to the roots of the forest of trees in dynamic memory are stored in several structures residing in the non-dynamic portion of memory (the semantic nest [page description stack], static [predefined] glue list, [word] hyphenation lists, equivalence [symbol] table, [symbol definition] save stack, font glue definitions, condition stack, and alignment stack [see appendix for further details]). Certain locations in dynamic memory are also reserved to point to special lists. Dynamic memory is parsed by starting from this group of pointers, which is smaller in number than the total set of pointers into dynamic memory.

Rather than develop an equivalent procedure to parse a compressed state description in a checkpoint file, the quiescence algorithm loads the descriptions of dynamic node memory and the structures described above which contain pointers to dynamic memory and parses the previous dynamic memory state and then restores the current state so that formatting can continue properly.

The parse descriptions are then compared to make sure the contents of dynamic memory are identical. INCTEX also verifies that parsing correctly covered the complete contents of memory in both states by checking that all occupied entries in dynamic memory were numbered during the parse traversal and all free locations were not numbered.

5.4 Comparison of Difference List Descriptions

State checkpoints are compressed by describing the changes to data structures as difference lists, instead of describing the data structure itself. The global variables, equivalence table, equivalence level table, and hash table are described by difference lists. Difference list descriptions allow fewer comparisons to be potentially performed since the number of changes is smaller than the number of elements in the data structure, although comparison becomes more complex. A difference list describes a set of changes as pairs consisting of a location and the current value of the location. No type of order is guaranteed, although changes are recorded in fact in the order they are made. A direct comparison of two lists is therefore equivalent to checking not only that the same locations have the same values but also that changes were made in the same order. Instead, an algorithm capable of handling ordering differences was developed.

Comparing the content of two data structures described by two arbitrary difference lists in any order requires additional complexity. The set of changed locations appearing in both difference lists can be compared to each other since the values of these locations are recorded in the checkpoint, but a mechanism is required to handle ordering differences in the two lists. Any set of changed locations unique to one list is not included in the other list because, by definition, it was never changed from its value at the initial reference state. Therefore these locations can only have the same values as the other state if their current values are the same as the initial reference state values. This can occur if a series of changes ends with the value being changed back to the original value.

The goal of the difference list comparison algorithm can be described more formally as follows:

Let N be the current state

Let P be the previous execution state

$L(N) = \{N_1 \bigcup C\}$ is the set of changed locations of N

$L(P) = \{P_1 \bigcup C\}$ is the set of changed locations of P

C is the set of locations changed in common for both states N and P

N_1 is the set of locations changed in N but not in P

P_1 is the set of locations changed in P but not in N

Then state N is equivalent to state P only if:

1. $V_P(C)$, the values in state P of locations C, is equal to $V_N(C)$, the values for state N (if a location is a pointer to dynamic memory, the parse number for the memory location must be the same).

2. $V_P(N_1) = V_I(N_1)$. The values of the locations unique to N must be equal to those of the initial state I (a location may have been changed back and forth to its original value). If this is true, $V_P(N_1) = V_N(N_1)$, since for state N these locations were not changed.

3. and similarly, if $V_P(P_1) = V_I(P_1)$, then $V_P(P_1) = V_N(P_1)$.

We therefore have to determine D, the changes unique to the current state, and O, the changes unique to the previous state, and verify that these values are the same as the initial state. This is accomplished by recording which entries in the current state have been changed and comparing them to the previous state checkpoint description. An entry in the difference list description from the checkpoint is changed in both states if it is also flagged as having been changed in the current state. The previous checkpoint value is then compared with the current value to see if the same change was made for both states and the flag is updated to record the result of each comparison. An entry is unique to the previous checkpoint if it is not flagged as changed in the current state, but the value can still be compared to the current state value, which is the same as the initial reference point value. This comparison thus requires no additional storage overhead. Any entries which are flagged different but never compared against a previous difference list entry are differences unique to the current state. The current values of the entries can be compared against the initial reference point values to see if they would be the same as the previous state's. This is not currently performed for any of the regions because of the storage overhead. It would not be useful for the global variables because the difference list describes a true set of exact differences from the initial reference state, not a set of cumulative changed locations which need not be different. Future research remains to determine if adding copies of the initial values would increase the set of situations where quiescence can be detected.

The flags are implemented in two ways, depending on the characteristics of the data structure. In most cases an array used. This allows an entry to be looked up in a single step but requires a linear scan to find the changes unique to the current state. The flags are implemented as a linked list of the changed locations for the font arrays because the size of the arrays is large and there are also a large number of arrays. This also requires a linear scan to check if an entry has been changed, but the number of such changes is small (often less than ten for a 50 page document) so only a small number of comparisons is required. Finding the changes unique to the current state is also linear but involves a much shorter search since the list length is the number of defferent locations while the array length is the number of locations in the structure. A more sophisticated set implementation could reduce storage overhead for the flags and the processing overhead required to find unique entries.

Four state regions are described by difference lists: the global variable region, equivalence table, equivalence level table, and hash table. These difference lists are compared as follows:

1. For each state region, we keep a set of flags, implemented as a simple array, with one flag per element describing its current state. Each flag starts in the state CLEAR, and can assume the states CHANGED, EQUIVCURR, EQUIVPREV, UNEQUIVCURR, or UNEQUIVPREV.

2. A flag is set to CHANGED when a state change is logged for that entry.

3. When a location is read from the difference list describing the previous state, we check the flag for that entry and...

   - If it has the value CHANGED then we know the location differs from the initial reference state in both states (it is in the set C described previously) and we simply compare the values and set the flag accordingly to EQUIVCURR or UNEQUIVCURR.
   - If the flag for the entry has the value CLEAR instead, the change is unique to the previous state (it is in P_1) and we compare it to the current state value which is still the same as the initial state value and set the flag to EQUIVPREV if it is the same (meaning the previous state is still equivalent to the current state) or UNEQUIVPREV.
   - The data structures are copied when the initial reference state is established so that the above comparison is possible.

4. If any flags remain in the state CHANGED after the difference list entries describing the previous state have been read, they must describe changes which are unique to the current state (the set N_1). We must verify these are equal to the initial state values in order for the current state to be equivalent to the previous state.

5. We can immediately determine nonquiescence whenever we set any entry UNEQUIV-CURR or UNEQUIVPREV.

6. After comparison, we have to reset the flags to the original CLEAR or CHANGED state so that they can be used for state comparisons for future pages. EQUIVCURR and UNEQUIVCURR entries are reset to CHANGED because they indicate entries which were changed in the current state. Entries set to EQUIVPREV or UNEQUIV-PREV are unique to the previous page state and unchanged in this state so they are reset to CLEAR.

7. The CLEAR/CHANGED flags are implemented differently for the font description arrays. These arrays can grow dynamically and are extremely large, but there are few random changes to them. When a location is read from the previous state checkpoint, we compare it to the changed locations for the current state by scanning through the list for the current state, keeping track of which entries have been compared. Entries which are unique to the previous state can be compared to the current state value. Because the list is short, this technique does not have a large cost.

When a global variable entry is compared it is first compared against a special address list which is used to determine if a special comparison technique should be performed. Binary comparison is used unless the variable is a dynamic node memory pointer, an ignoreable variable (a variable whose value will not affect future formatting, such as a page breaking variable which no longer matters after a page break, or a dynamic memory free list pointers), or the current file name string, which is only compared up to its current length. If a difference is found, the half word where the difference begins is identified because pointer variables to node memory actually occupy a half word and the dynamic memory tree traversal number is looked up and compared in the case of dynamic memory pointer variables.

Dynamic memory pointers in the equivalence table difference list are ignored because comparison of the dynamic memory parses checks them directly for equivalence.

5.5 Comparing additions to the string pool and font

The string pool and font array additions are saved in separate files when formatting is over. Their current contents must also be compared for quiescence checking. The contents from the previous pass are not replaced until the end of processing and are available for comparison. For each structure, the initial size (right after the style file is processed), size at the previous checkpoint, and current size is kept. These limits are also recorded in each checkpoint.

If a segment has been added to the string pool since the last checkpoint, the string pool file for the previous pass is read and compared for an identical segment. This algorithm always skips the initial region resulting from style file initialization which will always be identical in both checkpoints and allows the pool to be compared incrementally. The font arrays are compared similarly except that, because font arrays are saved as each segment is added, the comparison has to be done a segment at a time according to how the array segments were previously recorded.

If a segment in either the string pool or the font array is found to be different, then INCTEX can be sure that it will be different for all following page checkpoints as well, and keeps a flag indicating so, allowing it to forgo future comparisons. Currently the string pool and font array file is opened and closed for each state comparison and this could be made more efficient by maintaining a current cursor position, allowing each file to be opened only once during the entire session.

5.6 Input and Output Considerations

Assuming the current state of a page is determined to be equivalent to the previous state, execution will be identical and the same output will be generated for future pages for as long as the same input is encountered as before. The current input file is compared from the current offset at the end of the page against the previous offset in the previous version of the file to locate the next difference if any. The other input files are also re-analyzed for changes from the beginning. Formatting can be skipped between the current page and the last page which precedes any of these changes and restarted at before the changes.

The output which would have been generated must be recreated for the pages whose formatting is skipped. The segments written in each output file for these pages can be determined by looking at the output file offsets recorded in the page-mapping file. These segments are extracted from the copies of the last version of the output files stored in the **INC** subdirectory and copied to the current output. The primary formatted document file does not have to be regenerated in this way because it is separated into per-page files which are individually regenerated.

IncTeX turns off quiescence checking if a new input file is encountered which was not in the previous chain of input files. The input analysis routines are not able to compare different chains of input files.

5.7 Summary

The implementation of quiescence checking which has been described allows IncTeX to successfully skip further formatting if changes are made to paragraphs but the number of lines do not change. The overhead required to achieve this consists of several processing and storage components. Quiescence checking requires dynamic memory and the state referring to it to be parsed, saved so that the dynamic memory information from the previous checkpoint can be loaded and parsed, and finally restored so that formatting can continue. Parsing and comparing dynamic memory incurs processing overhead and storage overhead is incurred to save and restore dynamic memory. Additional structures and processing are also required to trace which entries in the font arrays, equivalence, equivalence level, and hash tables have changed so that it can be verified that equivalent sets of changes have occurred.

The success of our quiescence algorithm based on state comparison is affected by certain implementation choices. Section and appendix changes (see the next section on LaTeX) cause counter differences in the state which cause unnecessary quiescence failure. A different order of macro definitions will also confuse quiescence detection. This type of problem could be solved by analyzing and parsing the formatter state more thoroughly. For example, parsing the macro definitions in dynamic memory would solve the latter problem with macro definition order. Such solutions would increase the cost of quiescence detection however, although one saving grace is that determining quiescence is impossible can usually be achieved rapidly.

5.8 Quiescence problems with LaTeX

Two particular types of problems arose with LaTeX which made detecting quiescence more difficult.

The most common problem with quiescence occurs because of an auxiliary file which is read at the very beginning of the LaTeX document. Section titles and bibliography references are recorded in this file. The references are used when a citation occurs and at the bibliography section, which is usually inserted at the end of the document. The primary problem occurs with sections titles. Section title information includes the pages where the title occurs; it is used if there is a table of contents. A wide variety of editing changes cause section titles to move to different page, altering the auxiliary file so it differs on the next pass and forces reformatting at the beginning of the document. The entire document is reformatted, since quiescence fails on every page because the state is different, even if only a single section title has moved and only the range of pages between the previous position and the new position needs to be reformatted. For LaTeX documents, this auxiliary file change is the most common reason for unnecessary reformatting. Both types of limitations on quiescence detection arise because formatting data dependencies are not extracted and analyzed.

Quiescence detection also fails if a subsection title is added or removed in LaTeX documents, until some event, such as the appearance of an appendix resets the section counters. This phenomenon occurs because section numbers are recorded in formatter counters and changes involving counter variables lead to a state difference preventing quiescence detection. An appendix resets the section numbering and permits state comparison to detect quiescence. This is an example of a state side-effect which causes an unnecessary quiescence failure.

6 Restarting Strategy

6.1 Overview

Now that the components in incremental processing – input analysis, state checkpointing and compression, and quiescence checking – have been discussed, the way in which formatting as a whole is restarted and halted will be summarized. TeX is conveniently structured so that a single loop processes input and produces output pages. This loop was interrupted whenever a page is produced so that IncTeX can record state checkpoints as well as input and output offsets. The formatted output file is also split into separate files for each page so that pages can be regenerated individually. Pages are collated after execution is finished into a single document file by a separate program.

When a document is resubmitted, the style file and the source document are determined from the command line. The source document and any included input files are analyzed for changes and IncTeX locates a page checkpoint that precedes them from to restore the formatter state and restart formatting. Input positions are restored, and output files are restored to their state at that point, and formatting is restarted by entering the processing loop described above.

The incremental page-mapping file records the input and output file information for each page. It is retrieved from a file with the document rootname, followed by the suffix **.inc** in the checkpoint subdirectory called **INC**. This file also contains the style file which was used for the last pass. If a different style file or version is specified, the entire document is reformatted, otherwise the document input is analyzed to find the first change and the page checkpoint preceding the change is loaded (obviously, all processing preceding the change will be identical).

When INCTEX is invoked the first time, it creates all the checkpoint information in the **INC** subdirectory needed to restore formatting as follows:

1. When INCTEX is invoked the first time, it loads the initial style file, determines the initial regions in the string pool, dynamic token memory, and font description arrays which can be eliminated from future checkpoints, and copies the global variable state.

2. When an output page is produced, it records the current location in each input and output file (it records the current input and output files and offsets, allowing an execution point to be mapped to a point in the input and output) and saves a checkpoint. Each output page is saved as a separate file. INCTEX provides an option to checkpoint every fixed multiple of pages.

3. It writes this per-page input and output information to an incremental page-mapping file at the end of execution.

4. It records the last modified timestamp of all files and also copies all auxiliary output files into the **INC** subdirectory. Every input file which is user-writable is copied unless the file name ends in **.sty**, in which case only the time stamp is recorded. These are assumed to be style or macro files, which rarely change and are almost always read at the very beginning of the document. If an input file timestamp is recorded without copying the input file itself, INCTEX merely assumes the entire file has been changed when a different time stamp is encountered. If the style file is read at the beginning of the document, reformatting has to start from the beginning and it is likely that the dynamic macro memory definitions will be different so that quiescence can not be detected and the complete document will have to be reformatted.

5. All files are kept in a local directory called **INC**.

Incremental Passes:

1. When INCTEX is reinvoked, it loads the incremental page-mapping file describing where each page occurs relative to the input and output files and looks up the last modification time stamps of the input files from the mapping file.

2. If a file has a new timestamp, it scans the previous version of the file (if a copy was made) and compares it against the current file to see if it has changed. It locates the offset of the first difference, and finds the last page checkpoint which was made before that location in the file is read.
3. Each file with a new timestamp is scanned and reformatting must be started at the last page checkpoint preceding all changes. The checkpoint is loaded, input files are reopened and offsets restored, output files are restored by copying the previous versions up to the current offsets (except that the log file is not restored, so the user can see what happened specifically for this pass), and formatting is restarted.
4. Every time a page is produced, a new checkpoint is saved and quiescence is checked. To reduce overhead, quiescence checking is disabled if it fails more than five consecutive pages in a row.
5. If INCTEX detects quiescence, formatting is halted, and all input files are scanned for any later changes. If there are none, formatting is complete (all following pages will be the same as the previous ones). If the total number of document pages is less than the number from the previous reformatting runs, the excess page checkpoints are deleted. Otherwise, formatting is restarted at the checkpoint preceding the next change. The output in the auxiliary files for the pages which were skipped is regenerated by examing the output file offsets in the page-mapping information and recopying these output segments from the copies of the previous output into the current output files.
6. After formatting, another program collects each separate output page file and produces a single output document file.

7 Error Handling

The anticipation of errors is an important issue in the design of any software project. TEX separates errors into a warning class and a fatal class, but allows the user to force formatting to continue despite an error. We do not wish INCTEX to restart from a checkpoint made after a fatal error, since reformatting will certainly be garbled. A user abort or internal error may also interrupt the execution of INCTEX before the incremental page mapping file, page checkpoints, the string pool file, and the font array file are all properly saved. We thus also wish to verify that this set of files are consistent.

The formatting history code is saved in the incremental page mapping file. If a fatal error occurred in the last run, reformatting is forced from the beginning. The time when INCTEX started is also saved in the page mapping file, string pool file, and font array file. These files must contain the same time to be consistent. If not, execution must have aborted and total reformatting is also forced. The page checkpoints are written before these files, and in fact are selectively generated depending on which pages are reformatted, so no time is recorded in these files.

A missing end of document marker is considered a warning. INCTEX reformats the last page even if a document is considered unchanged if a warning occurred previously. The reason is that if an end of document is added, there is no difference which precedes the last character in the previous input document and no change in the document is indicated. Reformatting the last page will allow any addition to be properly reprocessed if there is a missing end of document. The alternative solution is to consider an addition past the previous end of document to be a change, even though the formatter would never scan it if the document was well formed.

In case of software revisions, a code indicating the checkpoint file format version is recorded in the page mapping file, string pool file, font array file, and each page checkpoint. INCTEX compares its internal format version code against the code in these files; if it is different because the format has been changed in a later software release then the files cannot be loaded and the document is reformatted from the beginning.

8 Performance

Benchmarks were run on a number of sample LATEX documents to measure storage and processing overhead in INCTEX. Benchmarks were run on a Sun SPARC 2 workstation with a local disk at off hours to reduce network traffic and file server delays and the documents were stored on the local disk. INCTEX was modified so that it would perform the quiescence check for every page. Three types of documents were tested, the simple seven page sample letter *sample.tex* in the standard LATEX distribution, a nine page article, and this fifty page technical report.

8.1 Storage overhead

The storage overhead for INCTEX consists of per-page checkpoints, per-document files consisting of the incremental page-mapping file, the font addition file, and the string pool addition file, and finally, copies of each input document which is user-writable. This overhead is given in table 1 (the overhead for the copies of the input is not given).

8.2 Execution time

The cpu and total elapsed formatting times for these documents were also measured in modes that allow the overhead for checkpointing, document analysis, and quiescence checking to be measured (see table 2). The modes are as follows:

	Per Page			Per-document
	min	av	max	
letter	21.6	22.9	24.2	4.2
article	30.8	33.4	36.7	5.0
report	25.0	34.6	47.6	9.1

Table 1: Incremental Formatting Disk Overhead (Kbytes)

Orig The execution time for the original unaltered TeX.

Batch The batch mode does not save any incremental information at all. Comparing this time to the original allows us to check that the software conversion was correctly performed and does not introduce additional overhead in the formatting code itself.

Virgin The document is processed as if for the first time. No document analysis or quiescence checking is performed, but state checkpoints are recorded, allowing the overhead for checkpointing to be measured alone.

Anal Document analysis and determination of a proper restarting page is performed in this mode. A comment at the beginning of the document is altered so that the entire document will be reformatted. The overhead of document analysis can be calculated by comparing it to virgin mode.

Quiesc Finally, quiescence checking is also enabled and the document is reformatted so that the additional overhead of quiescence can be calculated. A special version of INCTeX was used which checks every page for quiescence regardless of the number of failures. A dummy macro definition on the first page is altered so that quiescence detection always fails. This ensures that the formatter computation is virtually identical to the previous pass.

	letter		article		technical report	
mode	**cpu**	**total**	**cpu**	**total**	**cpu**	**total**
Orig	0.9	1.2	2.3	2.5	11.3	11.6
Batch	1.1	1.3	2.3	2.6	11.2	11.8
Virgin	1.5	2.6	3.1	4.4	16.5	27.2
Anal	1.6	2.8	3.1	4.5	16.5	27.5
Quiesc	2.5	4.8	4.7	7.1	21.8	45.0

Table 2: Incremental Formatting Time (seconds on Sun SPARC 2)

8.3 Discussion

The conversion from TEX to INCTEX did not alter the performance of the formatting code itself, as similarity in the processing times for the original (pure) version of TEX and the batch mode of INCTEX shows. State checkpointing incurs a 35% cpu overhead and a 70% time overhead for the nine page article document. Document analysis consumes very little cpu time, but can add up to as much as 30 msec for file access, as the difference between virgin mode and incremental analysis mode shows. Quiescence checking consumed about 70% more cpu overhead and about another 100% in real time. Total incremental formatting overhead adds another 100% in cpu overhead and 174% in real time. All overhead is measured relative to the time required in batch mode without any incremental processing.

Results show that input/output and network delays account for the overwhelming majority of the overall delay on a fairly powerful workstation. The CPU time is less than half of the overall execution time. Execution could be accelerated by expending greater CPU time to reduce input and output needs.

Disk and network delays and overhead for remote file system access have a significant effect on performance. The importance of IO performance on INCTEX is demonstrated by the effect of disk server performance which was observed during use. Server load was observed to affect execution time by as much as 50%. Moving the document directory from a unloaded remote SPARC 2 to the local workstation's local disk reduces the overall execution time for the technical report by about half.

9 Conclusions

INCTEX demonstrates that a complex batch program can be adapted for incremental processing, at least on a large (i.e. page) granularity level. TEX documents can be reformatted incrementally and quiescence successfully detected as long as the changes do not change the number of lines in a paragraph. Since INCTEX approaches incremental processing by checkpointing TEX as a deterministic machine, processing can be incremental at even smaller granularities such as input lines. However, the two major impediments are the linebreaking algorithm and storage overhead. Paragraph-level linebreaking algorithm makes reformatting below the paragraph granularity level difficult because linebreaks are computed for the entire paragraph at a time. Checkpointing below the page level would increase the checkpoint size because local data structures and variables used to determine the formatting of paragraphs and pages must be saved and a larger state must be saved.

Checkpointing overhead is a major problem in INCTEX. State compression could be improved further. Since IO and network delays outweigh cpu time by more than 100%, large improvements are still possible. Disk server speed and server load have a very large effect on execution time. Strategies could also be developed which attempt to predict change locality and save checkpoints at those pages or closely preceding them. However, the fundamental problem in INCTEX is that incrementality is based on an output structure, the page break, while for fine-grained incrementality it must be based on input entities. Identifying which input entities have been added, deleted, or changed is also essential for full-fledged quiescence detection. Without this information, for example, there is no way of deciding if a series of pages have been deleted or inserted except by comparing each checkpoint with every other checkpoint. Direct editor coupling is the only way to automatically capture this information. Furthermore, a fine-grained incremental processor must also be designed from the very beginning to maintain the data dependencies between input entities and output entities.

10 References

References

[Bro88] Kenneth P. Brooks. *A Two View Document Editor with User-Definable Document Structure.* PhD thesis, Stanford University, Stanford, California, 1988.

[CCH+86] Pehong Chen, John L. Coker, Michael A. Harrison, Jeffrey W. McCarrell, and Steven J. Procter. The VORTEX document preparation environment. pages 32–24, June 19–21 1986.

[CH88] Pehong Chen and Michael A. Harrison. Multiple representation document development. 21(1):15–31, January 1988.

[Che88] Pehong Chen. *A Multiple Representation Paradigm for Document Development.* PhD thesis, University of California at Berkeley, Berkeley, California, 1988.

[CHM88] Pehong Chen, Michael A. Harrison, and Ikuo Minakata. Incremental document formatting. In *Proc. of ACM Conference on Document Processing Systems*, pages 93–100, Santa Fe, New Mexico, Dec 5–9 1988.

[Fuc81] David Fuchs. The format of TEX's DVI files version 1. *TUGboat*, 2(2):12–16, July 1981.

[Fuc82] David Fuchs. Device independent file format. *TUGboat*, 3(2):14–19, October 1982.

[Int85] Interleaf, Inc. *Interleaf Publishing Systems Reference Manual, Release 2.0, Vol. 1: Editing and Vol. 2: Management*, June 1985.

[KF86] Donald E. Knuth and David R. Fuchs. Texware. Technical Report STAN-CS-86-1097, Stanford University, Stanford, California, 1986.

[Knu83] Donald E. Knuth. The WEB system for structured documentation, version 2.3. Technical Report STAN-CS-83-980, Stanford University, Stanford, California, September 1983.

[Knu84a] Donald E. Knuth. *The TEX Book.* 1984. Reprinted as Vol. A of *Computers & Typesetting*, 1986.

[Knu84b] Donald E. Knuth. A torture test for TEX, version 1.3. Technical Report STAN-CS-84-1027, Stanford University, Stanford, California, November 1984.

[Knu86] Donald E. Knuth. *TEX: The Program*, volume B of *Computers & Typesetting*. 1986.

[KP82] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. 11(11):1119–1184, November 1982.

[Lam86] Leslie Lamport. LATEX*: A Document Preparation System. User's Guide and Reference Manual.* 1986.

[Lia83] Frank Mark Liang. *Word Hyphenation by Computer.* PhD thesis, Stanford University, Stanford, California, June 1983. Available as technical report STAN-CS-83-977.

[Sch87] William F. Schelter. Sample infor display. Unpublished manuscript, 1987. Distributed with INFOR System.

A Detailed Description Of Checkpointing Strategy

A.1 Overview

This appendix describes in detail the changes which were made to TEX for state checkpointing. The reader is referred to the documentation on the original Pascal version of TEX in [Knu86], which describes the program and data structures. The C version of TEX by Pat Monardo is a faithful translation of the Pascal version; the same variable names, procedure names, and algorithms are retained.

The formatter state which must be carried over to continue formatting for the following pages occupies 400K. Most changes to this state occur in the dynamic memory section describing the current page description. Many of the other data structure changes are additions; the string pool stack is the purest example. Few changes to data structures are truly random access in nature. Even the global variables, which were identified and collected in one region, and was expected to be a target of frequent random changes, rarely change by more than 10%. The checkpointing strategy is to take advantage of these characteristics by describing the limited number of random changes in lists, describe large additions to data structures compactly, and describe dynamic memory, the most region which changes the most, fairly directly by simply eliminating unused regions. Control always returns to a special control routine called *main_control* in TEX after a page break occurs. A flag was added to indicate when a page is produced so that checkpointing (and quiescence detection) can be triggered in *main_control* after a page break.

TEX execution normally begins first with the reading of a standard TEX, LATEX, or other document style package, then the user's document is read and processed. The execution point after the style package has been read is chosen as an initial reference state. The state at succeeding pages is checkpointed as the **difference** relative to this reference state, with the exception that dynamic memory is saved in entirety with the empty regions compressed out. The differences are found by logging changes to the state or, in the case of the global variables, by directly comparing them against the initial reference values. State which is static throughout document processing thus never appears in any checkpoint. Certain data structures in TEX are append-only: the string pool and font arrays (complications involving the fonts will be described later). If the difference between these structures and the value at the reference state were saved, these potentially large additions would be saved redundantly in each state checkpoint. Instead, they are saved once for the whole document in separate files, and the current dimensions are recorded in each checkpoint so that the correct data structure size can be recovered. Checkpoint size variation is due primarily to the size of the dynamic memory descriptions. Checkpoints are saved in the **INC** subdirectory where all incremental information is kept. A page checkpoint is written as the file **doc.p.stc** where **doc** is the document root name and **p** is the current page number.

Checkpointing relative to each previous checkpoint would have saved about half the space but would have required loading each page checkpoint; this design requires only one checkpoint to be loaded. INCTEX provides an option that allows checkpoints to be saved every multiple of pages instead of each page, to reduce the storage costs.

TEX's state can be divided as follows:

- an append-only string pool (equivalent to an add-only stack),
- a set of font description arrays which is mainly append-only (fonts can be declared and loaded at any time), An entry describing a character can be changed when a dimension (such as height, width, slant, or spacing parameters) is altered, but this is rare,
- a random access add-only hash table for control sequence names,
- a random access equivalence table (symbol table),
- a dynamic save stack to store nested local symbol definitions in the equivalence table,
- global variables (random access of course), and
- internal dynamic memory for storing macros and typesetting box lists.

A.2 State Compression

A.2.1 Difference List Logging

Changes to the hash table, equivalence table, equivalence level table, and font arrays are logged in four **difference lists**, one for each set of data structures. A small number of routines are involved, and only a small set of software probes needed to be added to log these changes. Each list records the changed entries; the current values are saved retrieved and written when checkpoints are actually made. Each time an entry is modified, it is added to the list. To reduce duplicates, we check against the last 64 list entries for a repeat before adding a change. Logging is disabled during style file initialization when the initial reference state is being established. About 100K (25% of the state) is saved with this technique. Global variables are also described by a difference list which is computed by comparing the current values against a copy made at the initial reference point. An entry is identified by the address of the word which is different.

We make sure an entry has actually changed before adding it to the appropriate list. A pair of macros keep track of entries for each data structure, (*before_eq/after_eq, before_hash/after_hash, before_xeq/after_xeq, before_font/after_font*). These macros or software probes were inserted into the code sections where entries are modified. The *"before"* macro saves the current entry in peril in a temporary variable, the *"after"* macro checks if the value has changed and adds the entry to the proper difference list if so. A significant number of entries which are nonchanges can be eliminated in this manner with very low processing or storage overhead. It was necessary to use two temporary variables in one section which modified two entries in the equivalence table simultaneously.

Only the entry is identified, either by a table index or memory address, in the difference list kept in memory. When a checkpoint is written, the current entry value is also written. Each checkpoint thus records the *current* value of any difference list entries which have been changed from the initial reference value.

The font arrays are a collection of arrays of different element sizes (each at most one word in size). A font is identified by its font number. It is described by the set of elements indexed by its font number in the font arrays. Modifications to the font arrays are separated into changes to current entries, which are recorded in the difference list, and additions to the end of the arrays to describe new fonts, which are saved in a separate file. Font changes are logged as a series of word addresses and word values, although array entries may occupy less than one word. Because values are not recorded until the checkpoint is made, we need not worry that new values are overwritten by old values due to data length overshoots, although restoring the data in the order that was recorded guarantees this. Entries in the other tables are logged as half word integers instead of full words to save storage.

Entries are added to each list by the four routines *note_font*, *note_eq*, *note_hash*, and *note_xeq*). Change lists are kept in doubly linked blocks of list sections. Each block has a count of the number of entries it contains (each block occupies 1K). Each list also has a total list count, and a pair of global pointers to the first and last block in the list. When any of the four routines are called, it checks whether the current block is full and allocates a new one if needed. The routines for the hash, equivalence, and equivalence level tables scan the last 64 entries from the end of the list for a duplicate before adding an entry. This reduces hash and equivalence table list sizes by 15% on several typical documents. Difference list logging is switched off for any list if its size (including overhead) reaches the array size. The array is then written in full.

- The hash table, *hash*[], is modified in the routines *new_font()* and *id_lookup()*.

- The equivalence table, *eqtb*[], is modified in the routines *new_font()*, *eq_define()*, and *eq_word_define()*, *geq_define()*, *geq_word_define()*, and *unsave()*. The routine *new_font()* modifies two entries in the equivalence table simultaneously and requires the use of two temporary variables.

- The equivalence level table, *xeq_level*[], is modified in the routines *eq_word_define()*, *geq_word_define()*, and *unsave()*.

- The font arrays are modified in the routines *prefixed_command()*, and *find_font_dimen()*.

When a checkpoint is loaded, these difference lists are restored, as if the changes during processing of preceding pages had been logged.

The global variable state is saved as a difference list by comparing the current global region against a copy made after initialization. Because remote file access (disk and network overhead) is much higher than the processing required to do this on workstation equipped with a reasonably fast processor, this is actually faster than saving the global variables directly.

A.3 The Save Stack

Symbol definitions in the equivalence table are dynamically redefined and restored using the save stack structure. The array *save_stack* is saved, up to the last occupied entry preceding the limit in global variable *save_ptr*, in each checkpoint.

A.3.1 Append-Only Data Structures

Additions made to the string pool are not described redundantly in every checkpoint. Instead, since additions are cumulative and no changes are ever made, string pool additions are saved in a separate file called **doc.str**, where **doc** is the document root name, after formattting has ended. The correct string pool state is restored simply by recording the current pool limit in each checkpoint and loading any necessary segment fropm the string pool file. The initial portion of the string pool which exists at the initial reference point is not saved because it always exists in memory.

The string pool consists of two arrays: *str_pool*, the character buffer array, and *str_start*, which gives the end of each string. Strings are identified by a number which is used to look up the string location in *str_start*. The next free character buffer location in *str_pool* is given by *pool_ptr*, the next free string number in *str_start* is given by *str_ptr*.

The INCTEX routine *set_skip()* is called when the initial reference state is set. The current string pool is saved in the variables *pool_ptr_lowater* and *str_ptr_lowater*. The current string pool limits are saved in each checkpoint, along with the low water marks for doublechecking.

Modifications to the font arrays are separated into changes to current entries, which are recorded in the difference list, and additions to the end of the arrays for new fonts, which are saved like additions to the string pool in a separate file called **doc.font**, where **doc** is the document root name. Since changes can occur to segments which are added during formatting, any added segment is written at the next checkpoint. Changes made during reformatting are actually written to a temporary file called **doc.font.i** and made permanent when processing is complete.

The correct font state is restored by first loading any font segments which were added to the state for the current page, then any specific font changes from the page checkpoint.

Fonts are described by one primary array called *font_info* which contains the majority of data describing a font. The next free location in *font_info* is given by the global variable *fmem_ptr*. There are also 19 auxiliary arrays (actually 20 but one which is treated differently):

font_size	*font_dsize*	*font_params*	*font_name*	*font_area*
font_bc	*font_ec*	*font_glue*	*hyphen_char*	*skew_char*
char_base	*width_base*	*height_base*	*depth_base*	*italic_base*
lig_kern_base	*kern_base*	*exten_base*	*param_base*	

The last occupied entry in these auxiliary variables is the index in the global variable *font_ptr*. Fonts are described by an internal font number which is used to index into these auxiliary arrays. The initial font region is saved in the variables *fmem_lowater* and *font_lowater* by the routine *set_skip()*. The font region which has been saved is remembered by saving the current font limits in *fmem_last* and *font_last*. The current font limits are saved in each checkpoint, along with the low water marks for doublechecking.

One auxiliary font array, *font_used* is grouped with the global variables for storage purposes because it records which fonts have actually been used in the document. For INCTEX this actually indicates which fonts have been used on the current page. It was grouped with the globals because it might change frequently in INCTEX .

Since font segments are saved at the next checkpoint, the font file format consists of a series of segment descriptions, each of which consists of the indices of the end (*fmem_ptr*) and beginning *fmem_last* of the *font_info* segment which follows, then the indices of the end *font_ptr* and beginning *font_last* of the auxiliary font segments which follow.

The string pool typically occupies 22K, and the font arrays occupy a minimum of 75K for LATEX documents. 25% is eliminated by not saving the initial regions.

A.3.2 Dynamic Memory

The original version of TEX provided a single pool of dynamic memory. The C version written by Pat Monardo which we used as a starting point for INCTEX separated this into two independently allocated pools, dynamic node memory and token memory. Page descriptions are stored in node memory, macro descriptions in the token memory. Node memory is an internally allocated array of words, further separated into two regions, single memory for lists composed of single word elements at one end of the array, and multiword memory for lists composed of variable-sized multiple word elements. Both regions grow towards the middle, allowing them to allocate storage from a shared region of free space within them. Dynamic memory is compressed by saving the used regions and eliminating the unused regions, and in the case of token memory, an initial region created after style initialization does not have to be saved because it never changes.

Single memory is defined to be the region extending from *mem_max* down to the watermark in *hi_mem_min* in the array *MEM*. The unused regions are found by scanning the free list, a linked list of words which begins at the global pointer *avail*. A halfword field in each word gives the next item in the list. A bit in a corresponding bit map of memory is turned on for each empty entry. After creating the bit map, it is scanned and a sequential list of the occupied regions is created. The number of regions is written to the checkpoint, the beginning location of each region, the end of each region, then the contents of the region, repeated for each region in the list.

Multiword memory is saved in a virtually identical manner. It ranges from *mem_min* up to the watermark in the variable *lo_mem_max* in the array *MEM*. The free list which begins at the global pointer *rover* is scanned and bits in a bit map are set. Each free list block contains a *node_size* field which records its size in number of words, and a pointer to the next free block. The bit map is scanned to create a list of occupied regions. The number of regions is saved, followed by the beginning, end, and contents of each region.

Macro definitions are stored in the other region of dynamic memory called token memory. Token memory is organized as two parallel arrays of words, *tok_mem* and *tok_link*. *tok_mem* contains the data portion of each token, *tok_link* contains the pointer to the next token. The occupied region extends from *tok_low* to *tok_end*. The free list begins at the global pointer *tok_head*. The initial region after style file initialization is never changed, although entries from the free list may be used. If the free list is reset so that it points to a single large block outside of this region, then the entire initial region will never be altered, and we never have to save this region. After style file initialization, the procedure *set_skip* is called by INCTEX. This routine determines certain parameters which define the initial reerence state, one of them being the initial token region, which is recorded by saving the current value of *tok_low* in the variable *premac_lo* and the value of *tok_high* in *premac_hi*, and resetting the free list in *tok_head* so it points to the region after *premac_hi*. During checkpointing, the free list is scanned and a bit map of occupied regions of token memory is created. The bit map is scanned and a list of occupied regions is computed (any region within the initial token area is ignored). The number of regions is then written to the checkpoint, then the beginning, end, and contents of each region.

The net result of this compression is the elimination of about 25K (5%) from what would be written simply by saving the regions computed by checking the full extents of each dynamic memory region.

A.4 Page Checkpoint File Format

The page checkpoint file is composed of the following segments:

C	G	A	ST	F	E	H	L	S	M	T

A.4.1 Segment C

This is a preamble with the format:

SF	CC	P

SF: An 8 letter state checkpoint version code. If the checkpoint code is different than expected by the current software, the checkpoint was saved by an incompatible software release and complete reformatting is triggered.

CC: An integer which contains the hex code 8 if full incremental state is enabled, or the code 1 if the globals were not incrementally saved.

P: An integer giving the page produced before this checkpoint.

A.4.2 Segment G

This segment describes the global variables. The format is:

TO	FO	GD

TO: An integer giving the value of the variable *term_offset*.

FO: An integer giving the value of the variable *file_offset*.

GD: The global variable description. This is a difference list (see below) or a 12145 byte segment containing the complete set of variables if the code **CC** in the preamble segment is 1.

The variables *term_offset* and *file_offset* are the current column offsets on the terminal and in the log file. They affect when linebreaks are produced, and are specially saved and restored so the log file continues in the same way from any particular page as before. Unlike all other difference lists, the global variable difference list **GD** is *not* preceded by a list length. It is a sequence of pairs describing words which differ from the initial reference state value, followed by a terminator mark. The format is:

DL	DV	...	DL	DV	mark

DL: A word address which is never *NULL* (0).

DV: A word value.

mark: An integer-length terminator mark with value *NULL* (0), which can always be distinguished from a word address.

A.4.3 Segment A

This segment describes the arrays saved in per-document files, the font descriptions and and the string pool. The current limits of these arrays are recorded in each page checkpoint so that the correct array state can be extracted from the descriptions. The segment has the following format:

FML	**FL**	**PH**	**PL**	**SH**	**SL**	**mark**

FML: A halfword integer of type *ptr* which gives the size of the main font description array *font_info*.

FL: An integer which gives the largest auxiliary font number.

PH: A halfword integer of type *ptr* which gives the size of the character pool (*str_pool*) for strings.

PL: A halfword which gives the character pool size at the initial reference point so it can be checked for consistency.

SH: A halfword integer of type *str* which gives the size of the string index array (*str_start*).

SL: A halfword which gives the string index array size at the initial reference point for consistency checking.

mark: An integer with the value *NULL* (0) marks the end of this section.

A.4.4 Segment ST

This segment describes the save stack section. The format is:

SSL	**SS**	**mark**

SSL: A halfword integer of type *ptr* which gives the save stack size.

SS: The save stack segment: **SSL** elements of type *mword* which are one word long.

A.4.5 Segment F

This is a difference list describing the font arrays with the format:

DC	DN	DL	DV	...	DL	DV	mark

DC: An integer-length format code which is always the value *DiffListCode* (hex 88) for the font description, indicating that a difference list follows.

DN: An integer word giving the number of pairs **(DL,DV)** in the list.

DL: A word address which is never *NULL* (0).

DV: A word value.

mark: An integer-length terminator mark with value *NULL* (0).

A.4.6 Segment E

This segment describes the equivalence table (usually a difference list). The format is:

DC	DN	DL	DV	...	DL	DV	mark

DC: An integer-length format code which is either the value *DiffListCode* (hex 88) if a difference list follows, or *Array_NoDiff* (hex 11) if the table is written as an array.

DN: An integer word giving the number of pairs **(DL,DV)** in the list.

DL: An integer table index.

DV: A one-word table entry of type *mword.*

mark: An integer-length terminator mark with value *NULL* (0).

If the format code **DC** is the value *Array_NoDiff*, then the difference list does not follow the format code. Instead, the entire equivalence table follows the code. This feature is enabled if the difference list description reaches the size of the table.

A.4.7 Segment H

This describes the hash table (usually a difference list). The format is:

DC	DN	DL	DV	...	DL	DV	mark

DC: An integer-length format code which is either the value *DiffListCode* (hex 88) if a difference list follows, or *Array_NoDiff* (hex 11) if the table is written as an array.

DN: An integer word giving the number of pairs **(DL,DV)** in the list.

DL: A halfword table index of length *short.*

DV: A word length table entry of type *hh.*

mark: An integer-length terminator mark with value *NULL* (0).

If the format code **DC** is *Array_NoDiff* (hex 11), then the difference list is replaced by the entire hash table. This feature is enabled if the difference list description reaches the size of the table.

A.4.8 Segment L

This describes the equivalence level table. Its format is:

DC	DN	DL	DV	...	DL	DV	mark

DC: An integer-length format code which is either the value *DiffListCode* (hex 88) if a difference list follows, or *Array_NoDiff* (hex 11) if the table is written as an array.

DN: An integer word giving the number of pairs **(DL,DV)** in the list.

DL: A halfword table index of length *short.*

DV: A one-byte table entry of type *qword.*

mark: An integer-length terminator mark with value *NULL* (0).

If the format code **DC** is *Array_NoDiff* (hex 11), then the difference list is replaced by the entire equivalence level table. This feature is enabled if the difference list description reaches the size of the table.

A.4.9 Segment S

This segment describes the occupied segments of single-word dynamic memory. Its format is:

N	MB	ME	MM	...	MB	ME	MM	mark

N: An integer specifying the number of memory segments.

MB: A halfword integer of type *ptr* giving the beginning index of a memory segment.

ME: A halfword integer of type *ptr* giving the end of the segment.

MM: The segment itself, each word is of type *mword.*

mark: An integer-length terminator mark with value *NULL* (0).

A.4.10 Segment M

This describes the occupied segments of multi-word dynamic memory. The format is:

N	MB	ME	MM	...	MB	ME	MM	mark

N: An integer specifying the number of memory segments.

MB: A halfword integer of type *ptr* giving the beginning index of a memory segment.

ME: A halfword integer of type *ptr* giving the end of the segment.

MM: The segment itself, each word is of type *mword.*

mark: An integer-length terminator mark with value *NULL* (0).

A.4.11 Segment T

This describes the occupied segments of token dynamic memory, which consists of two parallel arrays *tok_mem* and *tok_link.* Its format is:

N	MB	ME	TM	TL	...	MB	ME	TM	TL	mark

N: An integer specifying the number of memory segments.

MB: A halfword integer of type *ptr* giving the beginning index of a memory segment.

ME: A halfword integer of type *ptr* giving the end of the segment.

TM: The segment of *tok_mem*, each halfword is of type *tok.*

TL: The segment of *tok_link*, each halfword is of type *ptr.*

mark: An integer-length terminator mark with value *NULL* (0).

B Detailed Description Of Quiescence Checking

Quiescence detection in INCTEX cannot be based on checking formatting dependencies directly, because they are not available in TEX. Instead it is based on the observation that, if the same state **q** and remaining input α_2 are repeated at some page, then the remaining pages will be identical to the previous pass. In fact, even if the remaining input α_2 is not exactly the same, any pages which are between the current page and the first difference in the input will be the same. Formatting can be halted at the current page and restarted at the page which precedes the next difference. If the input α_2 between each page segment is saved, checking the input is simple.

Comparing states is equivalent to checking quiescence to the degree which the state reflects only the information required for future formatting. However, even in the ideal case, when there are no extraneous state dependencies on the input, a formatter which scans the document in a single sequential pass like TEX is limited by the problem that, at any point in the input, it cannot predict what input it will encounter next and *what entities will be referenced and what information is no longer needed.* Therefore any changes to entities which might be referenced later must be encoded in the state, causing the state to be different. Direct manipulation editors have an inherent advantage because the dependency information for the entire document is available and they are able to recognize which parts of a document are dependent on the entities changed by the user. A changed macro definition which affects chapter headers is one example. INCTEX has no way of knowing if only one header references this macro, or if future headers will also reference it, while a direct manipulation system is theoretically able to scan the dependency information to determine this.

In practice, INCTEX's state comparison algorithm detects quiescence if a page has the same number of lines. TEX's formatting algorithm reads a paragraph at a time and computes linebreaks before deciding whether it can fit on the current page or whether it should be placed on the next page. The pending paragraph is saved in dynamic memory and accumulated onto the next page description. Linebreaking is finished *at page breaks* and future formatting does not depend on the current linebreaking data, which can thus be ignored for state comparison purposes. Theoretically the words (which determine the exact linebreaks in the paragraphs) on a page can be changed, but if the next page starts at the same paragraph as before, the current state should still be the same as before at the current page break. This depends on a clean algorithm design which does not introduce extraneous state dependencies on the previous input history, which in fact, is indeed true of TEX as the success of quiescence detection in INCTEX demonstrates.

One particular design feature of TEX, however, the internal dynamic memory pool, adds a complication to state comparison. Checking binary equivalence fails even though the *meaning* of the state in dynamic memory is the same because a different storage allocation sequence leads to a different *binary state.* The problem is that lists are stored in dynamic memory, and even if the contents are the same, the list items can be stored in different locations and pointers which are part of these items will also be different. The contents of dynamic memory are checked for equivalence by parsing the lists to check their contents.

The C version of TEX which we used divides dynamic memory into two types, one for tokens (macros) and one for "nodes". For ease of implementation, we have chosen to determine equivalence only in cases where tokens (macros) do not change, allowing us to check only if the token dynamic memory pool is exactly the same with a simple binary scan for binary equivalence and to parse only the dynamic node memory. The global variables and regions in arrays which contain references to data in dynamic node memory are identified and checked for content equivalence. Since we check only at new page boundaries, we can also take advantage of knowing that certain groups of pointers do not point to live dynamic data. The remaining portions of the state are checked for binary equivalence.

State comparison is also complicated by the state description compression techniques in INCTEX. Several data structures are described as changes relative to the initial reference state, which results in a shorter list than the complete data structure. These list entries have to be matched and compared to determine state equivalence. Although this is more complex than a direct comparison, a compensating factor is that fewer entries actually have to be compared.

We will now describe TEX's state as a set of separate logical regions which are scanned and compared individually. The reader is referred to the documentation on the original version of TEX, which describes the state organization and implementation [Knu86], The C version by Pat Monardo upon which INCTEX is based follows the original faithfully, except for the separation of dynamic memory into two pools. The formatter state can be divided into the following set of logically separate regions:

- **S** = (**G**, **S**, **E**, **X**, **F**, **P**, **H**, **T**, **N**)
- **G** = global variables
- **S** = Save stack for EQTB
- **E** = EQTB, the equivalence (symbol) table
- **X** = XEQ_LEVEL, auxiliary equivalence level table
- **F** = font information array segments
- **P** = string pool
- **H** = hash table
- **T** = dynamic token memory

- **N** = dynamic node memory

In most cases, the tokens (macros), fonts, strings, hash table, and equivalence levels (**T**, **F**, **P**, **H**, **X**) do not change from one run to another. However, the values of **G**, **S**, **E**, and **N** must be compared for equivalence.

The dynamic memory components (**T**, **N**) greatly complicate state comparison. The dynamic node memory component contains one fixed region N_f which will be described later that contains fixed definitions and a set of pointers to special dynamic memory lists. The global variables **G** can be further divided as a set
$\mathbf{G} = (G_a, G_b, S_n, S_c, H_a, H_b, HL)$ where

- G_b = *extraneous* global variables
- S_n = semantic nest
- S_c = condition stack
- H_a, H_b = word hyphenation pointers
- HL = hyphenation table
- G_a = all other global variables

The reason the state has been separated in this way is because each group, with the exception of G_a and G_b, contains pointers to lists in dynamic node memory (some are actually trees). In order to verify state equivalence, our strategy is to do a binary comparison on G_a, **T**, **F**, **P**, and **X**, (G_b can be ignored), and parse the dynamic node memory and verify the list contents are identical.

B.1 Extraneous Variables

G_b consists of:

- *chk_cid* (file character count)
- *max_var_used* (for statistics)
- *last_page_break* (we are always at a page break)
- *rover* (free list head)
- *avail* (free list head)
- *mem_end, lo_mem_max, hi_mem_min, hi_mem_max* (dynamic memory limits)

Because no paragraph linebreaking is pending, these variables can also be ignored (see section 820 in [Knu86]; all citations to sections are in this reference [Knu86]):

- *just_box* (last previous paragraph line)
- *best_place[]* (for calculating best line breaks)
- *best_pl_line[]*
- *minimum_demerits*
- *minimal_demerits[]*
- *passive* (pointer to a line breaking list data structure)

B.2 Semantic Nest

The top of the semantic list S_n (described section 211) is in the global variable structure *cur_list.* The rest of the stack ranges from *nest[0..nest_ptr−1]* in global memory. Each item contains fields *mode, prev_graf, aux* describing the current mode, and pointers *head, tail,* to the list in dynamic memory which is in the middle of processing. *Show_box()* can be used as a template in tracing the lists. In math mode *aux* can also point to a single record list which can also be checked with *show_box().* The final field, *mode_line,* a source line count, can be ignored.

B.3 Condition Stack

The condition stack S_c (described section 489) stores conditional modes. The global variable *cond_ptr* points to a linked list in dynamic memory with 2 important fields *(if_limit, cur_if),* which have to be checked to have the same values, and 1 extraneous field *if_line* (another source line count). The condition stack is generally very small.

B.4 Hyphenation exception table

The hyphenation exception table HL is actually a table called *hyph_list* of pointers to lists in dynamic memory which store hyphenation positions (section 925). Each *info()* field has to be checked for the same value as before. The table has 307 entries, most of which are usually empty.

B.5 Hyphenation variables

There are two global variables, *ha, hb,* which points to the beginning and end of the character sequence being considered for hyphenation. If there is no pending line breaking, or hyphenation in particular, these variables can be ignored. More study is required to determine this.

B.6 Equivalence Table

The equivalence table *EQTB* consists of six logical regions (section 220), two of which contain pointers to dynamic memory. Regions one and two point to single and multiletter control sequence definitions stored in dynamic token memory. We will check equivalence for cases where token lists (macro definitions) have not changed. Region five contains integer parameters, region six contains dimension parameters. These regions are only checked to have the exact same values as before. Region three point to glue specifications in dynamic node memory. There are about 512 entries, many of which are empty. The glue specifications are just checked to have the same values. Region four contains *par_shape_loc,* the location in the current paragraph outline shape and a subsection with about 256 entries pointing to box definitions which we can check according to *show_box()*; the remaining 1000 entries or so are checked to be exactly identical. There are thus about 750 pointers to dynamic memory items which potentially have to checked, and about 6250 other entries (25K) which should have exactly the same values. However, since we save *EQTB* changes in a list which often has much fewer entries, we actually check the list entries in each region according to the above.

- Entries $glue\_base \leq i < local\_base$ are pointers to glue spec's
- Entry *par_shape_loc* points to a node whose info() field should be the same as as before.
- Entries $box\_base \leq i < cur\_font\_loc$ point to box lists in dynamic memory that are checked according to show_box().

B.7 Save Stack

The save stack allows equivalence definitions to be nested (section 268). The stack is implemented as an array *save_stack[0..save_ptr]*; each item has three fields, *save_type, save_level, save_index.* If the item on the stack is of type *restore_old_value*, then the next item holds the old value in the outer group (scope). We check which region the equivalence entry lies in and perform an equivalence check. If the stack item is of some other type, we check the binary values are equal.

B.8 Font Arrays

The *font_glue* array (section 549) ranges from [0..font_ptr] and points to interword glue specifications in dynamic memory for each font. The other elements of the per-font information arrays, F, should just be checked to be the same.

B.9 Alignment Stack

The global variable *align_ptr* points to the head of a stack in dynamic memory used to save data for row alignments (section 770). The data structure is complex, all lists referenced in this structure should already have been checked for equivalence, but we also need to check that they point to the same sublists. Since aligned structures don't often cross page boundaries, the equivalence check may only handle cases where the alignment stack is empty.

B.10 Pre-Allocated Dynamic Memory

Region *mem_bot* to *lo_mem_stat_max* (0 – 13) holds pre-allocated fixed glue specifications (section 162). The values are always the same. Region *mem_top* to *mem_top-10* holds a special set of pointers.

B.11 Completeness Check

If all used dynamic node memory is not complete accounted for in the current page state or the description from the last run, then the equivalence algorithm fails.

C The Incremental Page Mapping File

The incremental page mapping file is a text file which consists of a preamble that gives version information, a section describing the input files, a section describing where the end of each page occurs relative to the input and output, and a section describing the output files. The page mapping file is saved in the **INC** subdirectory as **doc.inc** where **doc** is the document root name.

The input description section gives each input file, its modification time, so that INCTeX can determine if it has changed, and its parent file and the line where it is included in the parent. Files are identified by an internal program file ID.

The page description section gives the input file currently being read and the current file position at the execution point after an output page has been produced, timing statistics, and the current position in each output file. Since files are read in a well-nested order, the input section and page section provide sufficient information to map any change in a file to a particular page.

Finally, the output description specifies each output file.

An end-of-block mark (**EOB**), which is a line containing the number -2, separates each section.

The format of the page mapping file is as follows:

PRE	**I**	**EOB**	**P**	**EOB**	**O**	**EOB**

C.1 Preamble section PRE

The preamble has the following format:

H	**EOL**	**FL**	>	**FS**	**EOL**	**C**	**EOL**	**T**	**EOL**

H The history code of the last reformatting session. This indicates what type of error occurred if any.

EOL This indicates the end of a text line.

FL The style file is identified on the next line. **FL** is an integer giving the length of the identifier string. This allows INCTeX to determine if a different style file or version from the last session is in use, which will require complete document reformatting.

$>$ A > symbol (with a space on either side) appears next.

FS A string giving the style file identification and version number.

C An 8 letter state checkpoint version code. If the checkpoint was written by an incompatible software release, complete reformatting will be required.

T The time the last reformatting session was started. This is compared against the time in the string pool and font array files to make sure they are not inconsistent which might happen if there were a crash.

C.2 Input section I

The input description section has the following format:

N	EOL	D	...	D

N The number of input files.

EOL End of line.

D Two lines describing each input file (see below).

Each input file is described as following:

FID	L	NN	TS	EOL	PID	BL	EL	EOL

FID The internal input file ID.

L The file name length.

NN The file name string.

TS File modification time stamp.

EOL An end of line.

PID The internal file ID of the parent of this file (it is the same as the file ID if this is the root document file).

BL The character offset in the parent file of the beginning of the line where this file is included (0 if no parent).

EL The character offset of the end of the line.

C.3 Page description section I

The page description section has the following format:

N	EOL	D	...	D

N The number of pages.

EOL End of line.

D Two lines describing each page (see below).

Each page is described in the following format:

P	ID	OFF	FT	BT	ST	LT	QT	EOL

WOFF1	WOFF2	...	WOFF15	EOL

P The page number.

ID The ID of the current input file at the page break.

OFF The character offset in the input file when the page break occurs.

FT The time required to format this page.

BT The time to back up (copy) the input file(s) used on this page.

ST The time to save the checkpoint for this page.

LT The time to load the checkpoint to restart from this page.

QT The time to check quiescence for this page.

EOL End of line.

WOFF1 - WOFF15 The output file offsets for auxiliary output files 1-15 (-1 if the output file is not open).

C.4 Output file description section O

The output file description section has the following format:

N	**EOL**	D	...	D

N The number of output files.

EOL An end of line description.

D A line describing each page (see below).

Each file is described as follows:

ID	**L**	NN	**EOL**

ID The internal output file ID.

L The file name length.

NN The file name string.

EOL End of line.

D The Incremental Trip Test

The TRIP test is an acceptance test for TEX [Knu84b] which exercises the formatter and tests various internal memory limits. It consists of a special document test file and some special fonts which are passed through the formatter. The formatter output [Fuc81, Fuc82] is unparsed using a program tool [KF86] and compared against a copy of the correct output which should be produced. The log file is also compared against a canonical version to ensure that the proper messages are produced and that the correct errors are detected.

The TRIP test was modified into an incremental form by dividing the canonical formatter output file into a file for each output page so that they can be individually compared and by inserting comment lines in the TRIP document test file at the points where various page breaks occur. INCTEX is restarted incrementally at any of these page breaks by changing the comment. The same formatter output should be produced since only a comment has been changed. The reformatted pages are compared against the canonical correct pages. The log file should be identical except for an initial portion of the log which should be missing because it is generated during formatting of the pages which were skipped at the beginning. This is compared against a slightly different canonical log file than the batch version which is produced by INCTEX running in a batch mode option which identifies where each page break occurs and differs only in that the formatter identification message is different, page break messages have been added, and input line numbers are different because some comment lines have been added to the input. The log file is compared to make sure that the proper segment is missing and that all other information is identical except for identification and summary information generated by INCTEX.