Integrating TeX into a Document Imaging System

William M. Richter

Texas Life Insurance Company, 900 Washington Avenue, Waco, TX 76703, USA
hcswmr@texlife.com

Abstract

TeXmerge is an application programming interface for merging variable data into a pre-existing TeX document. This paper introduces the API and discusses its application in a document production and imaging environment.

Introduction

Modern computer hardware and software has made possible the construction of "document-imaging" systems. These systems maintain large repositories of documents in electronic form. In production environments of many large companies and in particular the life insurance industry, a significant percentage of printed documents are produced electronically in an automated fashion, usually by merging variable data into an existing document with some fixed structure. Storing scanned images of these electronically produced print documents wastes time, computing resources, and disk storage space. It is useful to address the problem of document storage along with the related problem of electronically formatting and producing printed documents. Then the choice of document formatter can be made such that the formatting engine used to produce printed documents may be reused to display those same documents in a document imaging environment.

TeX has been used as an important component in building a document production and imaging system at Texas Life Insurance Company. TeX's macro facilities, conditional typesetting, text-based source files, a robust page formatting mechanism, and pre-compiled format files allow it to play a central role in the system. TeXmerge, a C-language API, was developed to allow variable data to be merged, under program control, with static TeX source documents containing special merge tags to produce a final output document. This API is used to prepare policy contracts, produce automated client correspondence, as well as in interactive document preparation and in application-specific document production. Documents produced via the TeXmerge API are filed in the imaging system using a minimalist approach. TeX form files are stored once and separately from all document instances. Variable data along with a pointer to its associated TeX form file is all that comprise a stored document instance. When the document is displayed a "just-in-time" compile technique is used to reconstruct the document's `.dvi` file which is converted to PostScript for display purposes.

TeX has additional attributes that make it an excellent choice as document formatting engine. The ability to convert raster bitmaps to TeX fonts allow complex letter-head/footer macros to be developed and easily used in a fashion that lends itself to effective revision management. Incorporation of a Code 2-of-5 scalable barcode font has enabled printed forms and documents that are returned to the company from external individuals to be recognized by the document imaging scanner and automatically filed in the imaging system.

TeXmerge API

TeXmerge is a C-language API for merging variable data into a pre-existing TeX document (referred to as the "form"). The API is simple, light-weight, and easy to integrate into applications.

The API consists of a small number of functions, here listed in appendix A in the normal order of use. Most functions return an integer result code. Zero implies successful return. The integer result may be passed to the function `TeXmerge_GetErrorString()` to retrieve the corresponding error text string.

Example C Program A straightforward application of the TeXmerge API is illustrated in fig. 1. This program,

1. Creates an associative array with three elements.
2. Sets the elements to have names `THISVAR`, `THATVAR`, and `ANOTHERVAR`, respectively.
3. Opens an output file.
4. Merges the associative array into an existing TeX file called `test_form.tex` to create a temporary file called `temp.tex`.
5. Closes the output file and processes it for viewing with `xdvi`.

```
#include "stdio.h" #include "TeXmerge.h"

int main(int argc, char **argv) { TeXmergeName_t *array; int
count=3; int ret; FILE *fp; char *out_pathname="temp.tex"; char
*form_pathname="test_form.tex";

   array = TeXmerge_AllocNames(count);
   TeXmerge_SetArrayEntry("THISVAR", "some value", &array[0]);
   TeXmerge_SetArrayEntry("THATVAR", "blah, blah", &array[1]);
   TeXmerge_SetArrayEntry("ANOTHERVAR", "la-te-dah", &array[2]);
   ret = TeXmerge_OpenOutput(out_pathname, &fp, 0);
   if (ret != TXM_OK) {
      fprintf(stderr, "TeXmerge_OpenOutput(%s): %s\n", out_pathname,
        TeXmerge_GetErrorString(ret));
      return(1);
   }
   ret = TeXmerge(form_pathname, array, count, 0);
   if (ret != TXM_OK) {
      fprintf(stderr, "TeXmerge(%s): %s\n", form_pathname,
        TeXmerge_GetErrorString(ret));
      return(1);
   }
   TeXmerge_CloseOutput(fp);
   TeXmerge_View(out_pathname, TXM_WAIT);
   return(0);
```

Figure 1: C-language example application of the TeXmerge API

```
\batchmode
\def\THISVAR{some value}
\def\THATVAR{blah, blah}
\def\ANOTHERVAR{la-te-dah}
\input test_form.tex
\bye
```

Figure 2: A TeX file produced by the sample C program in fig. 1

After the call to `TeXmerge_CloseOutput()` the contents of `temp.tex` would appear as in fig. 2.

`test_form.tex` can have any TeX code of your choosing, including invocations of `\THISVAR`, `\THATVAR`, and `\ANOTHERVAR`.

Python Binding A Python[1] binding for the TeXmerge API is also available. A re-implementation of the previous C-code is given in fig 3.

Notes:

1. The Python version is cleaner.
2. Error detection via return values has been replaced with Python's exception mechanism; i.e., instead of methods returning integer result codes, they throw exceptions of the appropriate type which may be caught via the `try/except` construct.

3. The `TeXmergeName_t` arrays used in the C-code example just use simple Python dictionary objects. As a result the nagging `count` integer with tracks the number of array elements is no longer needed.

4. Constants defined in `TeXmerge.h` are accessed as attributes of the Python TeXmerge module.

TeXmerge in Production

Figure 4 depicts data flow between applications using TeXmerge and applications relating to the document imaging system (DIS). Most of the lines in the figure represent, not the flow of documents, but the flow of data necessary to build documents. Data from the policy administration system feeds a number of print-producing applications. Print producers come in four "flavors".

Bulk and Custom Print Producers create large volumes of documents such as annual reports to policy-holders, billings for premium due, and automated client correspondence. Most of these documents are from one to three pages in length and usually consist of three standard parts: a client copy, an agent copy and a file copy. All three parts have

---

[1] http://www.python.org

William M. Richter

```
import TeXmerge import sys

array = {'THISVAR': 'some value',
         'THATVAR': 'blah, blah',
         'ANOTHERVAR': 'la-te-dah'}
out_pathname = 'temp.tex' form_pathname = 'test_form.tex' try: fp
= TeXmerge.openOutput(out_pathname) except IOError, errmsg:
        sys.stderr.write('TeXmerge.openOutput(%s): %s\n' % (out_pathname, errmsg))
TeXmerge.merge(form_pathname, array, fp) TeXmerge.closeOutput(fp)
TeXmerge.view(out_pathname, TeXmerge.TXM_NOWAIT)
```
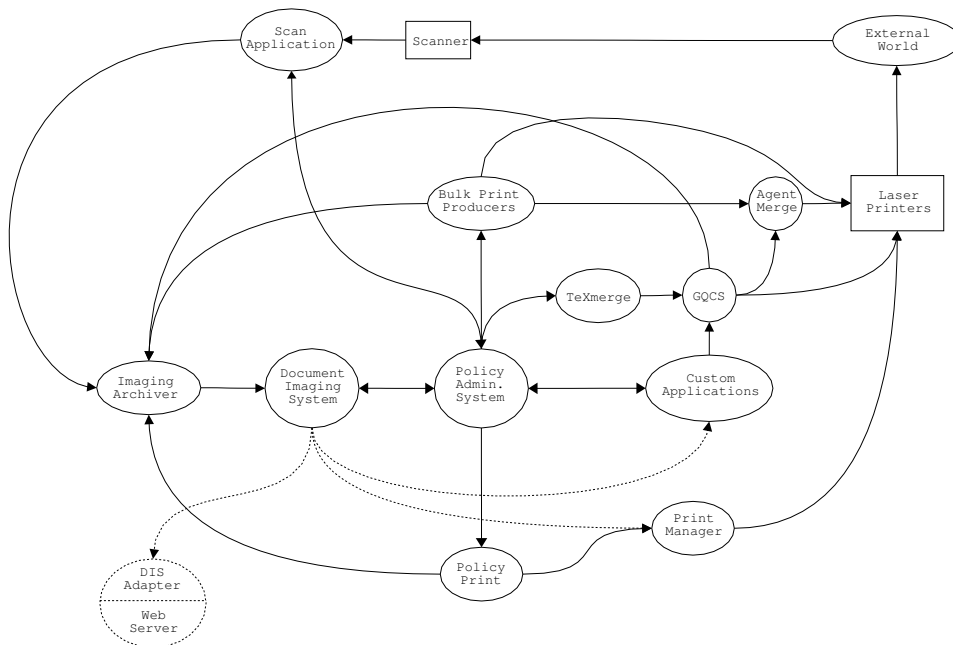
Figure 3: Python example.

Figure 4: Intra-application flow of document data.

content in common, but the agent and file copies may have additional information.

Policy Print is a customized application which produces policy contracts. Contents of policy contract documents vary by product and state in which the policy is issued. Typical contracts are 20 to 30 pages in length when printed duplex and their structure can be complex formatting-wise. A number of pages contain variable tabular information, certain pages must draw paper from an alternate input paper source on the printer, and still other pages must be landscape oriented. Because of the volume of print produced by the policy print ap-

plication[2] its output is routed to a print manager which tracks what documents are to be printed and performs the task of driving print streams to multiple printers. Work is currently under way to deposit a copy of policy contracts into the DIS. Also under construction is a mechanism which will allow the print manager to access the DIS "on-the-fly" as a document is being printed to retrieve an image copy of the original "application for insurance" and insert it into the print stream.[3]

---

[2] As of this writing, Texas Life produces around 2,000 policy contracts per month.

[3] This and other trickery such as selecting alternate input paper sources on the printer are accomplished through especially malicious abuse of the `special` macro.

General Queue Collection System Before discussing the last two print producer types we introduce the General Queueing Collection System (GQCS). Multi-user interactive applications produce multi-part documents that need to be saved for later processing or printing. GQCS removes the details of output handling from print producing programs and becomes a central point for collecting data related to print requests. It collects the tagged data for these documents and stores it in *subqueues*. A single GQCS daemon can serve multiple sub-queues. Usually it serves three for the standard image, agent, and client copies of a document. At night the contents of each sub-queue is extracted in bulk to output files, each file being forwarded to its destination system for final processing. Documents in the image sub-queue are routed the image archiver for storage in the DIS. Documents in the agent sub-queue are routed to the agent merge system which correlates items destined to a given agent into a consolidated print stream that is printed and mailed as a single bundle to the agent. Documents in the client sub-queue are printed and mailed to the external world.

Interactive Print Producers The final two print producer types are interactive in nature. *Custom Applications* are stand-alone systems that integrate TEXmerge to produce printed documents associated with transactions executed against the policy administrative system. Like the bulk print producers, these programs create documents that usually consist of the standard client / agent / file copies.

The TEXmerge Application The remaining print producer is *TEXmerge*. It was the first application developed to use the TEXmerge API, and its purpose is to generate documents interactively from pre-configured "form" letters. A detailed data flow diagram of TEXmerge is shown in fig. 5. TEXmerge uses a configuration file (XMC file) to declare the constituent parts of a document and to which sub-queue of a GQCS daemon each part of the document should be deposited.

For illustrative purposes consider the following files which together make a TEXmerge document.

Example.xmc:

```
Example_client.tex   client
Example_agent.tex    agent
Example_image.tex    image
```

Example_client.tex:

```
% Example_client.tex - the client copy
\stdLetterHead
```

```
\input Example_com.tex
\stdFooter
```

Example_agent.tex:

```
% Example_client.tex - the agent copy
\stdLetterHead
\input Example_com.tex
\vfil \centerline{* AGENT COPY *}\vfil\stdFooter
```

Example_image.tex:

```
% Example_image.tex - the image copy
\stdLetterHead
\input Example_com.tex
\vfil \centerline{* FILE COPY *}\vfil\stdFooter
```

Example_com.tex:

```
% Example_com.tex - part of the document
%                  common to all three parts
\texmergevar THISVAR \texmergevar THATVAR
\texmergevar ANOTHERVAR

This is just a sample document to show how the
multi-part document mechanism operates and how
merge variables work. Here we insert \THISVAR.
and now \THATVAR, and finally \ANOTHERVAR.
```

These four example `.tex` files and the `.xmc` file correspond directly with the entities shown in fig. 5. The environment variable `$TEXMBASEDIR` connects the TEXmerge application to the directory where the `.xmc` and `.tex` files reside. It looks for `.xmc` files in a subdirectory named `xmc` within `$TEXMBASEDIR` and for `.tex` files in a subdirectory named `tex` within `$TEXMBASEDIR`.

At startup TEXmerge scans the XMC directory and assumes the files it finds there are XMC configuration files and lists them in a chooser box as in fig. 6. When a selection is made, the configuration file is read to determine what `.tex` files will be used in the document and to what GQCS sub-queues the constituent parts will be deposited when saved. Then each configured `.tex` file is scanned recursively via the TEXmerge API to determine what merge variables are declared and it displays a frame as in fig. 7. With the frame displayed, TEXmerge's work is done and it's the user's turn to work. The task of entering data into the merge fields may be shortened in many cases. The TEXmerge application has been programmed to recognize the names of many merge variables and it assigns special meaning to them. For example, `ONAME` represents a policy owner's name, `INAME` is the insured's name, `SDATE` is the system date, etc. By typing a policy number into the `POLNUM` field and pressing enter TEXmerge
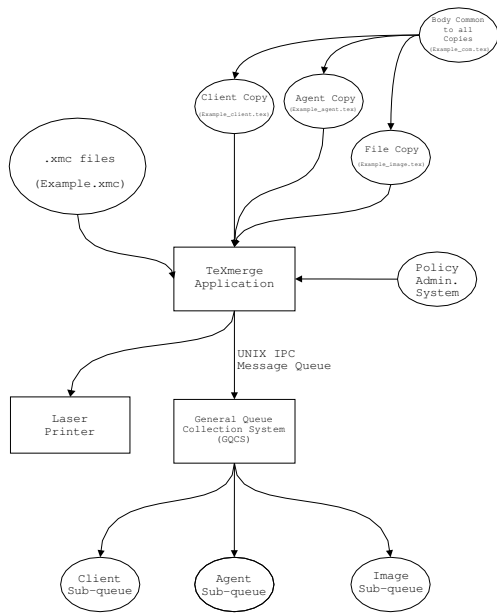
Figure 5: TEXmerge multi-part document configuration.

Figure 6: Top-level window of the TEXmerge application.

will access the policy administration system using the entered value. If it finds data for the matching number then all merge variable names found to have special meaning will be filled with data from the system. Once the merge fields are populated the document can be saved via GQCS, printed, or viewed via xdvi. The printing and viewing operations are accomplished via functions in the TEXmerge API.

Workflow Management Using a 2-of-5 barcode font with TEX has enabled a simple workflow environment to be established. A number of documents are intended to be completed by clients externally and returned to the company for further processing. The TEXmerge API has been extended in the following way: When a document is printed via the TeXmerge_Print() API call the form document is scanned for the sequence \def\WRKBARCODESTR{...}. If this macro is in the document two things happen.

- A transaction identification number will be assigned to the document and stored in the policy administration system.
- The transaction identification number will be encoded as a barcode and printed in the footer of the document. When the document is re-

turned and it is scanned, the transaction identification number will be recognized by the document scanning equipment.

These events enable the document to be automatically archived into the DIS without operator intervention. Future enhancements will also automatically route and image of the document back to the clerk who initiated the transaction for final processing.

Integrating TEX into the Document Imaging System

Finally we focus on the document imaging system in use at Texas Life, and the storage of documents produced via the TEXmerge API. A full description of the DIS is outside the scope of this paper; however a brief overview is appropriate here.

The DIS was originally designed to store two types of documents.

1. Reports normally printed on an old-style IBM line printer. These reports range from one to 10,000 pages or more, each page consisting of a

Figure 7: Merge field input window of a TEXmerge document.

fixed number of rows and columns of printable characters from the ASCII codeset.

2. Scanned images. Usually stored in the CCITT Group-3 fax format.

Over time document types based on Hewlett-Packard's PCL page formatting language (i.e., PCL print streams normally destined to a LaserJet printer), and Adobe's PostScript page description language were added, as well as several standard graphics file formats such as GIF, JPEG, and TIFF.

The screen shots in figures 10 and 11 are of the browser used to view documents stored in the DIS. They show both the index panel used for reviewing the document titles that are stored in the system, and the document panel which actually displays the documents. Folder tabs near the bottom of the window allow quick switching between the index and document displays and eliminate the need for manipulating multiple windows.

From a storage perspective the DIS is split into two components (see fig. 8). Contents of documents are stored in files residing within a UNIX filesystem (*host files*). Document titles are indexed into a hierarchical system having a familiar "document/folder" paradigm and is provided by a Relational Database which associates each title with a host file containing
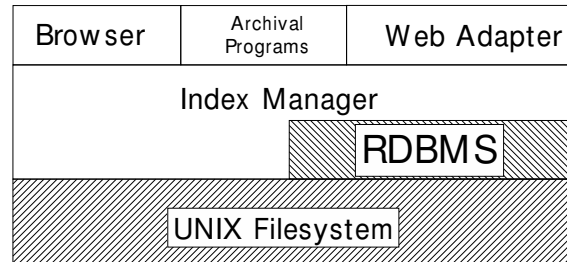

Figure 8: Layer diagram of the document imaging system.

the document's content. Further, the index tracks what folders own what documents.

Host files have a simple internal format. Each host file contains a line-oriented *metadata section*, which stores data about the document. The metadata section contains index information and can be used to rebuild the DIS database index should it become damaged. Format of the host file following the metadata section is document type dependent. ASCII- and PCL-based documents store multiple pages per host file. Scanned documents in CCITT Group-3 fax/JPEG/GIFF/etc. store each page in a separate host file. The storage mechanism for TEXmerge and PostScript documents will be discussed in the next section.

Storage Strategy for TEXmerge Documents
The storage mechanism for archiving TEXmerge-based documents provides several options which result in files sizes from only a few kilobytes (depending on the amount of variable data to be merged into the document) up to the size required to hold a full PostScript version of the document.[4] The different options for storing TEXmerge-based documents are rooted in the idea that this type of document can exist in at least three forms:

1. A simple associative array of tagged data. Special tags in the data specify what .tex file the data should be merged with (TEXFILE) and in what directory the .tex file is to be found (TEXMBASEDIR). This is the canonical form of a TEXmerge document (at least from a storage perspective) and consumes the least amount of disk space.

2. A file of TEX code ready to be compiled. A TEXmerge document in this form would appear as in fig. 2.

---

[4] Most correspondence produced by Texas Life results in PostScript file sizes of 70 kilobytes on average.

3. A PostScript file from `dvips` and is obtained from processing the `.dvi` file which result when compiling the file from item 2 above. From a display/ rendering perspective PostScript is the canonical form for TEXmerge documents since they must be in this form before they can be displayed.

There are benefits and disadvantages to storing TEXmerge documents in each of these forms (a classic *speed vs. space* dilemma), although the first and last forms seem most useful. Operationally, storing the PostScript is simplest and is least demanding in CPU cycles. In the present DIS Ghostscript is used to rasterize PostScript code for display on an X-Windows terminal, and already having the document in PostScript yields the least amount of work to get the image displayed. Depending on complexity of the document, number of fonts used, etc. the size of a PostScript file produced by `dvips` can be considerable, so storing TEXmerge documents in this form does not minimize disk space usage.[5] Option 1 above minimizes disk space usage at the cost of CPU cycles and other complications. The majority of documents will be stored and seldom, if ever, viewed. Therefore, the constraint minimizing disk space is the controlling consideration when choosing a storage strategy.

The Minimalist Storage Strategy Saving disk space costs in other areas. Each time the document is displayed, it must be compiled by TEX and then converted to PostScript via `dvips` which costs in CPU cycles. Both of these operations can be accomplished via the TEXmerge API. The more subtle problem is how to store the `.tex` form files.

TEX-Freeze and Revision Management Saving the tagged data array in the document along with the name of its associated `.tex` form file is not sufficient to enable the document to be reproduced. The form file (and any `.tex` files that it `\input`'s) must somehow also be stored internal to the DIS.[6] A complicating factor is that the original form files themselves will be modified at various times and new forms will be added. So while the form files are living, evolving entities, the documents stored in the DIS must be static and always appear exactly as they did the day they were produced.

The mechanism which allows TEXmerge documents to appear static in the DIS "freezes" the form files when TEXmerge documents are archived into the DIS. The place where frozen form files are stored is called the "TEXfreeze", and it is simply the mount-point for a UNIX filesystem. Pathnames of form files in the TEXfreeze are derived by concatenating the TEXfreeze pathname and the form's original pathname. For example, if we have a form `/x/y/z/tex/form1.tex` and the TEXfreeze directory is `/texfreeze`, then the pathname of the frozen version of `form1.tex` would be `/texfreeze/x/y/z/tex/form1.tex`.

Having a storage mechanism is only half the picture; we still need a method to allow for multiple versions of the same form. We accomplish this by inserting a timestamp into the filename of the form document. So the frozen pathname in the example above would become something like `/texfreeze/x/y/z/tex/form1_20010618.tex`. The most recent version of each form is pointed to by a softlink. The name of the softlink is the forms's un-timestamped name. Continuing the example; in `/texfreeze/x/y/z/tex` we would have a directory entry that looks like:

`form1.tex —> form1_20010618.tex`.

Freezing of form files is the task of the TEXfreeze manager. It detects changes between a form file and its current frozen version by calculating an MD5 cryptographic hash of the two files. If the hash values differ the new version of the form is stored in the TEXfreeze as discussed above and the softlink is updated to point to the new version.[7] The TEXfreeze manager also recursively follows `\input` macros and freezes those files also[8].

Within the document's data there is a tag (TEXMFORMAT) which controls the format TEX should use when processing the input source. Having laid out the scheme for tracking revisions of TEXmerge forms, it should be pointed out that the exact same process is carried out for freezing format files.

With TEXfreeze providing archival and revision management facilities for form files, we turn to the process of archiving tagged data for a TEXmerge document. The document's data is stored in a host file as discussed in the section "Storage Strategy for TEXmerge Documents" with one slight modification. The value of TEXFILE (the document's associated form filename) is adjusted to have the proper timestamp value (obtained from a `readlink(2)` system call on the softlink) inserted into the filename. Also,

---

[5] The current DIS stores over 500,000 TEXmerge documents. (And growing daily.)

[6] Form files used by the TEXmerge application and other TEXmerge-enabled applications are stored in UNIX filesystems and are independent (but related) to the forms stored in the DIS.

[7] The MD5 hashes are also saved in the TEXfreeze for efficiency purposes.

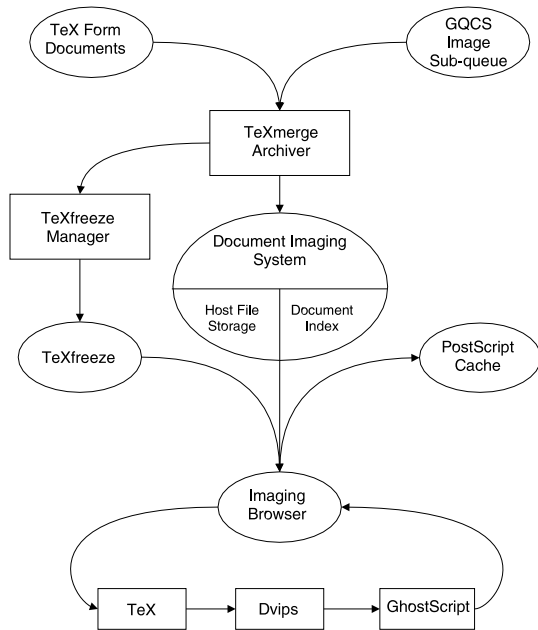[8] Using the environment variable TEXINPUTS to search for non-explicit pathnames.

Figure 9: Storage strategy for TeXmerge documents.

Figure 10: Index panel of imaging browser application.

the TEXMFORMAT tag will be adjusted with its proper timestamp value.

"Just-in-Time" Compile With form files frozen in the TeXfreeze and the document's data archived in the DIS the task of displaying the document is almost trivial. The TeXmerge API is used to convert the tagged data into a `.tex` file which is compiled by TeX. The `.dvi` is passed through `dvips` and the resulting PostScript is rasterized and displayed. One "trick" here is getting TeX to find the form file(s). There is a tag in the document's data, TEXMBASEDIR which originally pointed to the directory in which the document's form resided. That value must now be prepended with the directory of the TeXfreeze. We accomplish this by setting an environment variable, TEXFREEZE, to point to the TeXfreeze directory. Then, just before `exec()`'ing the TeX compiler, the environment variable TEXINPUTS is prepended with `$TEXFREEZE/$TEXMBASEDIR/tex`.

The pipeline sequence of tagged-data to `.tex` to `.dvi` to `.ps` to pixmap is expensive computationally. An easy way to limit this series of operations each time a TeXmerge document is displayed is to insert a cache manager at the beginning of the sequence. The final PostScript output is saved in the cache. Each time before a TeXmerge document is disp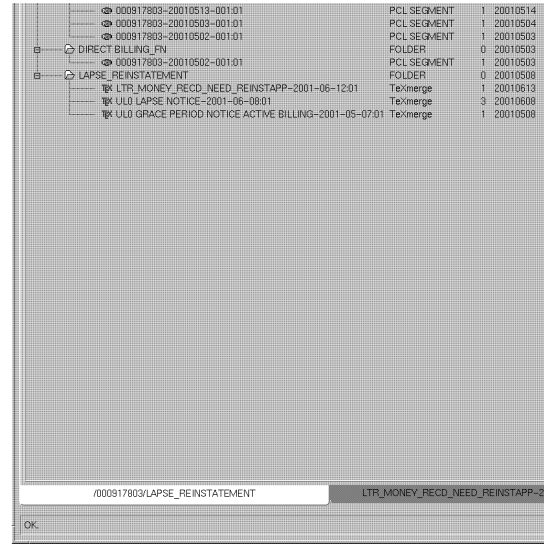layed a check is made to see if the PostScript is already in the cache. If it is found there the whole compile sequence is skipped. Files are dropped from the cache after reaching an age of two weeks to keep the size of the cache directory bounded.

Future Developments

TeXmerge-2 Development is complete for the TeXmerge API and no further work on it is planned. TeXmerge-2, a follow-on API, is being developed. The monolithic `.tex` form files will be replaced with a mechanism which assembles the `.tex` file from a series of small text segments or paragraphs that can be reused in various documents. The simple `\texmergevar` variable declaration will expanded to include other types of variables such as toggle buttons and option boxes to make documents with conditional elements easier to construct. All configuration information, text segments, and variable declarations will be stored in RDBMS tables. A major reason for eliminating the `.tex` form files is resistance from non computer-literate users to embrace the TeX philosophy and learn a new skill. Documents based on the TeXmerge-2 API will not have the space efficiency that TeXmerge documents have since all text for TeXmerge-2 documents will be stored directly in the document's data tags, and
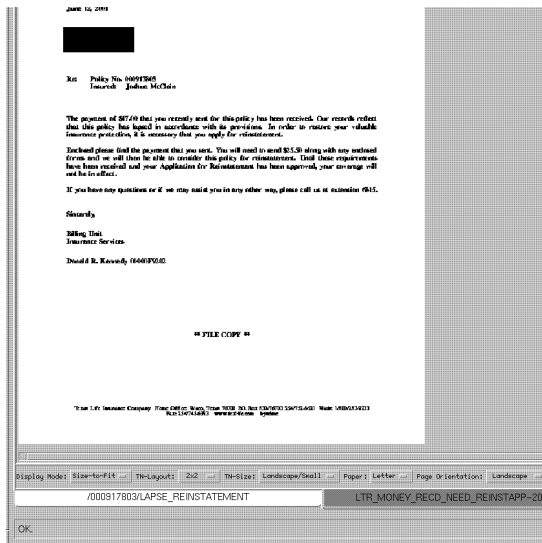
William M. Richter

Figure 11: Document panel of imaging browser application.

preparation of the finished document will be less efficient due to the large number of database accesses that will be required to build the TeX code for the document. It is the opinion of the author that TeXmerge-2 is not the best track that new development efforts should take; it is management and end-users desire.

Web Interface to DIS As shown in fig. 4, work is currently under way to serve documents from the DIS to web browsers. A set of library routines is being developed to render any document type in the DIS to Adobe's Portable Document Format (PDF). This library will be used in the planned DIS web server adapter. Security issues are a major consideration that make this an extensive development effort.

Integrate METAPOST into DIS Other authors have integrated METAPOST into applications. META-POST is anticipated to be useful in designing forms where lots of graphics primitives (lines, curves, etc.) are required. An API similar in style to the TeXmerge API is planned.

Utilize Advanced TeX Formats To date all form documents are based essentially on the plain TeX format. The TeXmerge API allows any TeX format to be used via the TEXMFORMAT environment variable. The LaTeX and ConTeXt formats are anticipated to be useful tools in developing new policy contract forms and for other documents needing a consistent "look-and-feel."

Utilize Other Typesetters Other typesetting engines such as pdfTeX, *xmltex*, and $\mathcal{N_{T}S}$ should be investigated to see what benefits they might bring to the document production and archival environment.

Conclusion

TeX is an excellent engine for integration into a production environment where large volumes of documents need to be produced and archived efficiently. The TeXmerge API allows that integration to occur in many applications and for many purposes. Efficiency is gained by moving data between systems instead of fully formatted documents. Finally, using TeXmerge in the DIS has provided many benefits, the most important being the space savings derived from the method of storing TeXmerge documents, and the ability to interactively produce documents and archive them using the same formatting engine.

Appendix A TEXmerge API

TeXmergeName_t *TeXmerge_AllocNames(int count)

Variable data is passed to TEXmerge as an associative array. Each entry in the array is a name/value pair. This function allocates an array of such name/value pairs containing `count` entries.

TeXmergeName_t *TeXmerge_FreeNames(TeXmergeName_t *array, int count)

This functions frees a name/value array. Obviously this one is not listed in order of use!

int TeXmerge_SetArrayEntry(char *name, char *value, TeXmergeName_t *entry)

This function sets `entry` to the passed `name` and `value` strings. The next two functions are for convenience and their use is not mandatory. In this and the next function, `value` may be passed as NULL to indicate 'no value'.

int TeXmerge_SetName(char *name, char *value, TeXmergeName_t *array, int count)

This function searches `array` for an entry whose name value matches `name`, and then sets the entrie's value to `value`.

char *TeXmerge_GetName(char *name, TeXmergeName_t *array, int count)

This function searches `array` for an entry whose name value matches `name`, and then returns the entries value string. The function returns NULL is `name` is not found in `array`.

char *TeXmerge_GetNames(char *pathname, TeXmergeName_t **array, int *count)

This function searches the `.tex` file specified in `pathname` for occurrences of lines having the format:

`\texmergevar NAME`

Lines of this form enumerate the merge variables that the document will use. The function returns a list of the names in the pointer variable pointed to by `array`. The number of elements in the array is returned in the integer pointed to by `count`. `array` should be freed when no longer needed with a call to `TeXmerge_FreeNames()`.

int TeXmerge_OpenOutput(char *pathname, FILE **outFP, char *preamble)

This functions creates an output file named `pathname`. An opened `FILE` pointer is returned in `outFP`. `preamble` is an optional "snippet" of TEX code that should be written at the beginning of the file. If `preamble` is passed as null, then no preable code is generated.

int TeXmerge(char *pathname, TeXmergeName_t *array, int count, FILE *fp, int options)

This function is the heart of the API. `pathname` is the name of a TEX form file containing invocations of macros whose names are the name values set in the passed `array`. `fp` is the `FILE` pointer returned by `TeXmerge_OpenOutput()`. `options` controls the merge operation. Currently the only option is whether or not to draw a frame around the merged variables (TXM_FRAMEVARS).

int TeXmerge_CloseOutput(FILE *fp)

After all invocations of the above functions are complete, this function should be called to close the output file. `fp` is the `FILE` pointer returned from `TeXmerge_OpenOutput()`.

int TeXmerge_Process(char *pathname, char *dvidrv_name)

Once the output file has been closed, it is ready for backend processing by TEX. This function invokes TEX and then the dvi driver named in `dvidrv_name`. All temporary `.log` and `.dvi` files are removed after use.

int TeXmerge_View(char *pathname, int waitOption)

A convenience function to run TEX and then run `xdvi`. `waitOption` is one of TXM_WAIT or TXM_NOWAIT. If TXM_NOWAIT is passed, then the current process if forked and then `xdvi` is run in a child process.

int TeXmerge_Print(char *pathname, char *lpargs, char *output_pathname)

A convenience function to run TEX and then run `dvilj`. If `lpargs` is non-NULL then it is used as switches for the `lp` command and the resulting `.lj` file will be queued for printing via the `lp` system. The pathname of the resulting `.lj` is returned in the character array pointed to by `output_pathname`.

William M. Richter

```
char *TeXmerge_MakeBarcodeStr(char *composite, TeXmergeName_t *array, int count);
```

Encodes the comma-separated list of data names in `composite` and corresponding values in a character string. It does the encoding by converting the name/value pairs to a Python dictionary object and passing it to the Python barcodeUtil module's encode function.

```
TeXmergeName_t *TeXmerge_DecodeBarcodeStr(char *str, int *count, int *ret);
```

Decode a barcode string and return its contents as an array of `TeXmergeName_t` structures. Caller is responsible for disposing of the array with `TeXmerge_FreeNames()`. If an error occurs, this function returns NULL and returns an error code in the integer pointed to by `ret`. On successful return a pointer to the array is returned, the number of elements in the array is returned in the integer pointed to by `count` and 0 is returned in the integer pointed to by `ret`.

```
char *TeXmerge_MakeTeXBarcodeMacro(char *name, char *composite, TeXmergeName_t *array, int count);
```

Build a snippet of TeX code encoding the values from `array` for names listed in comma-separated list passed in `composite` appropriate for printing with a Code 2-of-5 barcode font. The name of the macro will be the string value pointed to by `name`. The returned value is in a static `char` char array; it must not be freed and will be overwritten on subsequent calls.

```
char *TeXmerge_GetErrorString(int)
```

This functions returns a character string description corresponding to the passed integer value.