Automatic Tables Using SGML, C, and TEX

Robert McGaffey
Oak Ridge National Laboratory
Building 2506 MS 6302
P. O. Box 2008
Oak Ridge, TN 37831-6302 U.S.A.
Phone: 615-574-0618; FAX: 615-574-6983
Internet: rwm@ornl.gov

Abstract

This paper presents yet another method for doing tables with TEX. This process is different in that the source of the tables is not TEX but SGML, and there are no formatting instructions in the input. We present a method whose goal is to produce the highest quality output under the constraints given. We also present a set of problems that result from this method and suggest a solution.

Automatic Tables

Definition. Perhaps the best way to define an automatic table is to describe how it gets created. Suppose you are a TEX expert and I tell you that I want you to typeset a table with a specific number of columns. I also give you the default type of entry for each column. I am not telling you how many rows there are, nor how wide any of the entries might be. Now, it's up to you to design the table without the entries. I will then add the entries myself and expect some good result. Finally, after I add the entries and print the result, I may not modify such things as `\tabskip` glue to make the table be more pleasing to the eye. So, the table is created from its content to the paper without human intervention.

Corollaries. Some obvious conclusions about such tables: since we do not know the length of the headings or the text entries, the decision to wrap headings and/or text must be made automatically. Since we do not know if the headings to a particular column are wider than the other column entries we cannot adopt a simple `\hidewidth` approach for fear that our column headings will overlap each other. Furthermore, we do not know beforehand how long a table will be. Some of ours are 75 pages long which prohibits the direct use of `\halign` due to memory constraints.

Why automation and why TEX. Our problem is that our articles are ultimately to come from a database where a researcher may have selected various parts of many articles for closer scrutiny. In spite of the fact that many want to remove paper from our desks it has not happened yet and will not for some time. Thus our researcher wants several copies of the parts of articles he/she has selected on paper for later study. We do not want anyone at this point to have to gather all of the parts together and typeset them before our hero can have them. We want him/her to be able to say, `PRINT` this and turn around to the laser printer (in a few short minutes) and retrieve the output. Thus our hero need not know any typesetting language at all to get the results. And that's why TEX is being used. It is the most programmable of all typesetters and thus the choice most likely to generate a 'pleasing to the eye' paper for our hero to study.

Goals

We want the following:

1. The inputter should be faced with an easy-to-edit ASCII file.
2. Decimal alignment capability without "extra" columns.
3. No heading to migrate into the heading on either side of it as could happen if `\hidewidth` is used.
4. Sizing of headings and text entries to be determined on the fly depending on the discovered width of both the heading and the column entries under the heading.
5. Both column spanning and row spanning.
6. Tables of many pages to be handled without losing either the inherent benefits of `\halign`'s measuring or exceeding TEX's memory capacity.

7. Headings must be automatically migrated from page to page of a many-page table.

The Algorithm

What I have done. We start with an SGML instance file. For those of you not in the know, this is an ASCII file in which all of the elements of information in a document are labeled with SGML tags. In our case, we use the tag `<CELL>` to indicate a table entry. For example, a numerical field may be indicated by `<CELL>493.7</CELL>`, an equation by `<CELL TYPE="eqn">xy/a</CELL>`, and text by `<CELL TYPE="text">Robert</CELL>`. In the event that a particular cell spans, say three columns and two rows, it must be tagged differently, say, `<CELL TYPE="text" C="3" R="2">Robert</CELL>`. Note that `TYPE`, `C`, and `R` are called attributes of the element `CELL`. A row consists of the start tag `<ROW>`, any number of cells, and the end tag `</ROW>`.

Headings are given special treatment in our system. A main heading consists of `<CH1>`, the heading, and `</CH1>`. Subheadings are included *inside* of the main heading they modify. Let's say we have a table with the main headings: Main 1 and Main 2. And that both of these headings have two subheadings with obvious names. Then the markup of these headings could be `<CH1>`Main 1`<CH2>`Sub 1a`</CH2>` `<CH2>`Sub 1b`</CH2></CH1><CH1>`Main 2`<CH2>`Sub 2a`</CH2><CH2>`Sub 2b`</CH2></CH1>`. Note that the headings do not come out in the order needed by TEX.

And I could go on to describe the rest of the table elements but I hope the above suffices to show how the information is delineated in the tables we use. If not, the Appendix contains a short sample table and the SGML markup we use for that table.

Each `<CELL>` comes equipped with another attribute not yet shown. The `COORD` attribute gives the row and column numbers spanned by the particular entry. Thus if our large `CELL` above was entered in the fourth row and the third column of a table it could be fully described by `<CELL TYPE="text" C="3" R="2" COORD="4-5,3-5">Robert</CELL>`, as it spans rows four and five, and columns three through five. We do not want inputters to have to deal with the determination of the `COORD` attribute so we have a parser/translator program and a `C` program which together generate the `COORD` attributes. The translator expands the table so that all implied (defaulted) attributes are available to the `C` program which then calculates and outputs the `COORD` attributes.

Now, we have the SGML instance file we wanted in the first place. The integrity of the information has been preserved and the coordinates of table cells are now available for insertion into a database so that our hero may access, say, the third and fifth columns of a table while ignoring the other columns.

Now we use our parser/translator to convert from SGML to pseudoTEX. We are now left with many holes in our TEX file. Since we input subheadings as part of our heading elements, our table headings are interleaved; that is, some subheadings may appear in the document before all of the major headings. Our table entries may have white space before and/or after them. And, we have no preamble.

Recalling our goals, we want to take advantage of the `\halign` ability to premeasure columns and yet also handle 75 page tables. Well, the only way to measure the columns is to let TEX do it. Yet we cannot turn TEX loose blindly or we may someday exceed its memory capabilities. Also, we do not ever want to encounter the extra white space in the last column that Knuth refers to on page 247 of *The TEXbook*. What if we let TEX measure each table entry and report the result to another program which can then decide the best width, height, and depth for each column and row in the table? Then, that program could generate a suitable TEX file which would then typeset the table using an algorithm designed to create a result which is pleasing to the eye.

That's the plan.

So, we execute a `C` program to unravel our headings and to remove unwanted white space. Also, this is where we add our extra columns to the aligned decimal columns so that decimals line up. Note this does not break goal number two, as the inputter never sees this TEX file. The result is a TEX file which will run with the proper macros; but, it will not produce a table. There is still no preamble. What it does produce is a file giving the height, width, and depth of every single cell entry in the table. For the headings and text entries, it assumes that the heading/entry will be typeset on one line.

What I hope to do. Here is where I will use a `C` program to mimic TEX's `\halign` process. The natural width of every single column will be determined in two ways: first, by ignoring all headings, and secondly, by assuming headings are not stacked. When headings are ignored, text columns will also be ignored. At the same time,

the natural height and depth of each row will be determined, also, in two ways. During this phase, every spanning entry will be stored away in a list for future processing.

Next, the width of the text columns and the headings will be determined by trying to mathematically choose widths to make the table pleasing to look at. For example, we don't want a column with a two inch heading sitting on a column of numbers whose natural width is one-half inch. I intend to use a set of parameters which can be tweaked with experience until a good job is done. We shall see.

Each entry consists logically of a cell we can create in TEX with the box `\hbox{\tskip\cskip \vbox{\rskip` cell entry `\rskip}\cskip\tskip}`. The `\tskip`s take the place of `\tabskip`s. `\cskip`s are used to surround the columns of a cell when the header is larger than the rest of the column. `\rskip`s are used to handle vertically spanning problems. And `\vtop` or `$\vcenter$` will take the place of `\vbox` when appropriate.

Now, we have to process our spanning entries. If the entry already fits inside the rows and/or columns it spans, then we remove it from the list. If not, we calculate a row factor which is the amount we must expand the spanned rows to make our spanner fit; or, we calculate a column factor; or, both. Then, since spanning entries can overlap, we are going to select the smallest factor we find and increase each of the spanning rows or columns. This is done by increasing the row's `\rskip`s or the column's `\cskip`s. Now, we reprocess the spanning entries, since the factors may be changed by the previous expansion. We then iterate this process until all of the spanning cells fit. Now we can modify our aformentioned pseudoTEX file by inserting `\settabs` and `\+`s in the right places. We only have to make sure that we have correctly specified each box and that we have left gaps to handle vertical spanning.

Of course, we are talking theory here, not reality yet.

The Problems

The biggest problems here involve the fact that a `C` program that doesn't know TEX is stuck in the middle of the process. For example, we allow footnotes in our tables. If the inputter inserts a period in the footnote, then the `C` program may pick that period to be the decimal upon which it must align the column. I know this from experience. I have not solved this problem in theory or practice. For now, I ignore them. I suspect that if I cause `<FTNOTE>` to be translated to something like `\footnoteiii` and also output the footnote to another file so that the `C` program could never see it, I could make it work.

The Proposal

The real cure, however, is for someone with more smarts and time than I have to tackle this problem without the use of the `C` program. After all, TEX always knows more about the text and math and the curly braces, etc., than I could ever teach a `C` program.

Appendix

Sample Table

XYZABC

XYZ			ABC		
X	Y	Z	A	B	C
372.466	493.7	45	124	489	280
372.40	493.7	45	124	489	280
372.	493.7	45	124	489	280
XY/A		832	abc	774	INT
		qrr	aaa	799	

SGML Representation of Table

```
<TABLE NUMBER="1" ID="xyzabc">

<TITLE>XYZABC</TITLE>

<CH1>XYZ<CH2>X</CH2><CH2>Y</CH2><CH2>Z</CH2></CH1>

<CH1>ABC<CH2>A</CH2><CH2>B</CH2><CH2>C</CH2></CH1>

<TBODY>

<ROW1><CELL>372.466</CELL><CELL>493.7</CELL><CELL>45</CELL><CELL>124
    </CELL><CELL>489</CELL><CELL>280</CELL></ROW1>

<ROW1><CELL>372.40</CELL><CELL>493.7</CELL><CELL>45</cell><cell>124
    </cell><cell>489</cell><cell>280</cell></ROW1>

<ROW1><CELL>372.</CELL><CELL>493.7</CELL><CELL>45</CELL><CELL>124
    </CELL><CELL>489</CELL><CELL>280</CELL></ROW1>

<ROW1><cell type="eqn" c="2" r="2">XY/A</CELL><CELL>832
    </CELL><CELL TYPE="TEXT">abc</CELL><CELL>774
    </CELL><CELL TYPE="TEXT">INT</CELL></ROW1>

<ROW1><CELL TYPE="TEXT">qrr</CELL><CELL TYPE="TEXT">aaa
    </CELL><CELL>799</CELL><CELL></CELL></ROW1>

</TBODY>
</TABLE>
```