Subsidies[7]	1,180	
Miscellaneous[8]	2,300	10,290
Total expenses		$ 34,710

Budget summary: (revised 4/83)

Balance forward 1/1/83	$ 24,607
Total income	40,350
Total expenses	(34,710)
Estimated balance 12/31/83	$ 30,247

Notes:

These figures, with the exception of the budget summary, are identical to those published in TUGboat Vol. 3, No. 2. The rescheduling of the Stanford meeting from March to July has been noted.

All expense figures include an AMS overhead charge of 18%.

1. Numbers of members and subscriptions are based on 1982 figures for individual memberships and library subscriptions. The estimate for institutional memberships is now obsolete; 27 institutional members have joined as of March 31.
2. Lengthy descriptions of macro packages will be available for purchase separately.
3. Although no formal plans have been made for a second general meeting in 1983, one is assumed for the Fall.
4. Editorial services include programming, reviewing and editing; clerical services include maintaining the data base and mailing list, and other administrative duties.
5. Support is budgeted for attendance at one meeting of ANSI X3J6.
6. Costs for advertising TUG membership in trade publications.
7. Money available to Finance Committee to subsidize travel and membership fees for individuals when appropriate.
8. Postage/express charges, telephone tolls and supplies, plus programmer and clerical services not associated with production of TUGboat.

Respectfully submitted,
Samuel B. Whidden, Treasurer

* * * * * * * * * * * *

Software

* * * * * * * * * * * *

TEXhax SUMMARY

David Fuchs

Here's a distillation of the information that has gone through the TeXhax mailing list over the last few months. Most of it consists of details of installing and maintaining the WEB and TEX systems, and it is not very well organized.

One change to both systems is that all of our WEB programs now discard trailing blanks on all input lines. This aids in portability, since there are a surprising number of systems in which the number of trailing spaces a program gets to see on each line is some random function of the phase of the moon. It also improves the TEX language, since it is harder to have invisible differences between two files lead to differing results. Implementers should check their change files, since we had to alter a system-dependent module in many of the programs to make this change (look for *input_ln*).

Another reason you might get the "Change file entry did not match" error is that most of the discretionary hyphens (\-) that were included in the WEB source code have been removed, since they are no longer necessary with TEX82's hyphenation algorithm. Similarly, a number of comments in the WEB programs that looked like `$x$` now look like `|x|`, since this is the correct way to use WEB. Finally, the ANSI Pascal document specifically rules out the statement `WRITE(I:0)`, so we have changed all our WEB programs to say `WRITE(I:1)` to mean "write out I with no leading or trailing blanks".

For TEX, there are a few other changes to watch for: The module ⟨Globals in the outer block⟩ has been renamed ⟨Global variables⟩, so if any of your change file entries say "⟨Globals ... ⟩" you'll have to change them. The *qi* and *qo* macros have been joined by *hi* and *ho*, which you should add to your change file if this is a module you changed. Finally, the macros *float* and *unfloat* have been added to aid TEX installers on systems on which it is impossible to use native floating point numbers for glue ratios.

There are a few new features in WEB. First, the new control sequence `@"` is similar to `@'`, except that it indicates that a hexadecimal constant follows. Note that TANGLE.WEB actually uses `@"` in one place, so if you are trying to bootstrap a new Tangle from an old one, you'll have to temporarily change `@"8000000` to `@'1000000000` in the new TANGLE.WEB when you process it with the old Tangle (otherwise, the old Tangle will ignore the `@"` and interpret 8000000 as a decimal number, and you'll end up with a TANGLE.PAS that won't run on itself). This adulterated Tangle should be able to run through a clean copy of the new Tangle and produce the same correct Pascal program. Once you reach this state, you can toss out the altered Tangle, and forget that we ever suggested changing a WEB file directly.

Two other new primitives: `@/` forces a line-break in the Pascal output of Tangle, and `@=...@>` means to put the included text into the Pascal output "verbatim". Also, WEAVE now keeps track of which modules are changed, and prints all references to these module numbers with an asterisk. WEBHDR al-

lows you to output only the changed modules, so you can make a short listing of only the modules you changed, by using your change file to put "\let\maybe=\iffalse" in limbo at the beginning of the WEB program. See the latest TOPS-20 TEX change file for an example of this.

The new TANGLE.WEB uses the new @ feature, while the old versions of Tangle will ignore it as an unknown control code. Thus if you are bootstrapping to the new Tangle by using an old Tangle, your first new TANGLE.PAS will not quite agree with the TANGLE.PAS that you get after running the first new TANGLE.PAS on the new TANGLE.WEB. But the difference is only in the debug-breakpoint code, and doesn't alter the meaning of the Pascal program, so this won't cause any special trouble.

A number of people have pointed out deficiencies in the way change files are done. For instance, it is not too convenient to change a single line in the middle of a module. There is also some sentiment for a system in which you list *all* of the lines you are changing, so that Tangle/Weave would detect the case where you get a new version of the program and one of the modules which you have in your change file has been altered (so your change might no longer be correct). I personally favor this approach, but we don't have the manpower to consider implementing this right now.

A note of caution on Tangle: More than one person has tried altering Tangle to produce all lower case output. This is dangerous business, because Tangle's code for collapsing constants looks in the output buffer for the strings "DIV" and "MOD". Thus, you can get incorrect results if you change Tangle such that "div" and "mod" might end up in the buffer.

Tangle's output is now a bit different than it used to be. First of all, line breaks occur at semi-colons or right braces whenever possible. Also, comments are inserted showing where each fragment of code came from. For instance, part of the Pascal output might look like:

```
...{123:} Foo; {456:}
        Bar; {:456} More; {:123} ...
```

This means that module number 123 looked something like this:

Foo;
(Another module)
More;

And module 456, called (Another module), consisted of the single line:

Bar;

Thus, it is now easy to tell where each piece of Pascal code came from. We intend to use this information with the data gathered in the statement counting experiment to find out how much of the total runtime of TEX82 each individual module accounts for.

Note that the new output format implies that each program produced by Tangle is of the form:

```
{xxx:} PROGRAM ZZZ; .... END. {:yyy}
```

The Pascal manual and the ANSI standard are not specific about whether a program can end with a comment, but no one has yet reported this to be a problem.

Here's a problem that a few TEX installers are facing, having to do with evaluation of expressions. Consider:

```
program expres(output);
type sixteenbit = 0..65535;
var s,t : sixteenbit; i,j : integer;
procedure p(k :integer);
begin write(k) end;
begin
    s := 65535; i := s + 10; p(i);
    8em t := 10; i := s + t; p(i);
    8em p(s + t); p(65535 + 10);
end.
```

This program should print out 65545 four times. We have heard reports of a few compilers that try to optimize some of the expression evaluations to be sixteen bit calculations, and produce the wrong result in some of these cases. The new ANSI Pascal standard document specifically says that all expressions have to be computed to full integer precision, so the compilers in question are in the wrong. This does not bode well for sixteen-bit systems with a "LONG INTEGER" type, however, since even if you use your change file to change all integer variables to LONG INTEGER, the compiler might not do the above calculations correctly.

Us folks at Stanford are interested in addressing any bugs you may find in any of our WEB programs. The bug report format we most appreciate is "Module X in program Y is wrong because Z." If you have found that a bug exists in TEX82, but you can't locate the cause, then it would help for us to look at all the data. We need input files as well as the log file. If you say \tracingall, TEX gives its most verbose output. So please turn everything on, in the vicinity of whatever bug you're diagnosing; this makes it much easier to pinpoint the problem.

In the same vein, please send a note if you come across any supposedly non-system-dependent modules in TEX that you find you had to change. They might be appropriate to be added to the index under "system dependent", or we may alter them so they aren't system dependent any more.

A hint for installers/maintainers: One trick I use is to keep a copy of Prof. Knuth's change file for the Sail system. Whenever there is a new TEX, I look to see where his new change file is different from his old one, and I check my change file for TOPS-20 to see if it needs a similar alteration. You can do the same thing by keeping a copy of the TOPS-20 change file, and seeing when it changes. Actually, now that things are pretty settled down, it is probably easier just to check TeX82.BUG to see which modules have been changed, to check whether your system-dependent stuff is impacted. I also save a copy of all the WEB programs, so that when new ones come, I can find all the changes, but this has not been necessary very often.

Advice on making your TEX efficient: It is important that all records are declared such that they will be packed efficiently into memory. Referring to Module 110 in the brown Version 0 listing of TEX, note how the type *memory_word* is made. The hope is that your compiler will use 32 bits of storage for each *memory_word*. Well, one of the variants of *memory_word*, *glue_ratio*, is a *real*. In Pascal/VS, for instance, a *real* is allotted a double-word, which blows it right there. The proper thing to do in that case is to change the definition of *glue_ratio* in module 106 to be a *short_real* (in the change file, of course).

Similarly, VS Pascal insists that if you really wanted a variable declared 0..255 to occupy a byte instead of a word, you have to say

foo: **packed** 0..255

(note that the placement of the reserved word **packed** is non-standard). So, to get TEX down to a reasonable size, you'll have to change the definitions of *quarter_word*, *half_word*, and perhaps even *two_choices* and *four_choices* in module 110. This sort of change might be appropriate in other places too, but because most of memory is taken up by *memory_word*s, it shouldn't be crucial.

After your TEX port has passed the TRIP test, you should turn off run-time debug support. For production TEX it shouldn't be necessary to do bounds checking, uninitialized variable checking, and the like. Of course, if you run into an apparent bug, you'll probably want to turn it back on to help trace the problem as far as possible before reporting it to Stanford (hint, hint).

Advice on porting TEX: First, make sure to consider the modules that are listed in the index under 'Dirty Pascal'. A few of these modules are debugging code that look through the big "mem" array, and are considered dirty because they read from variants that weren't written into. That is, TEX may have done `MEM[0].SC:=0.0`, but a dirty module might `WRITE(MEM[0].INT)`. This has worked OK on all machines we've run into so far, but one can imagine an architecture in which it would cause a problem. Of course, normal users will never run into this code—it's only for TEX installers/debuggers.

The dirty module ⟨Display the value of *glue_set*(*p*)⟩ requires a little special attention. On our system, if *glue_set*(*p*) was erroneously set to a pattern of bits that did not represent a legal floating point value, due to a bug in TEX, then our run-time system would blow up while trying to print out its value. In order to make the code robust in the face of such bugs, so that the person trying to find the origin of the bug would be able to continue the job and use TEX82's internal debugging support to look around for further clues, the module in question was changed so that it first looked at *glue_set*(*p*) as an integer and figured out whether it was a legal floating point number. If not, it simply prints "?.?" instead of *write*(*glue_set*(*p*)). Of course, this is very system-dependent. On other computers, it may be appropriate to remove this test, but it will certainly be true that you'll at least have to change it.

Other than the debugging modules mentioned above, TEX should never read from a different variant than it writes into in any record. Also, TEX should never refer to an uninitialized variable, except for the variable *ready_already*. The details about *ready_already* are pretty well covered in the section of the TEX program titled "**The Main Program.**"

Different systems have different conventions about I/O to the user's terminal. On some systems, INPUT is hardwired to the keyboard, OUTPUT is the screen, and that's it. On others, there might be another built-in file that is hardwired to the screen, and INPUT and OUTPUT might always refer to disk files. Another possibility is that the program can dynamically tell the system which files should be associated with the terminal, and which with the disk. The TEXware programs and TEX itself try to be flexible enough to deal with all these possibilities. Consider TFtoPL, which mentions three files in its program statement in module 2: *tfm_file*, *pl_file* and *output*. Module 2 also mentions that all of the writing to the *output* file goes through the *print* and *print_ln* macros; so if you have a system, say, where output to the terminal must go to file *tty*, then you can change the definitions to:

@d *print*(#)==*write*(*tty*,#)
@d *print_ln*(#)==*write_ln*(*tty*,#)

You'd probably also want to change the program statement not to include the file *output*, and you might have to do a rewrite on *tty*.

The same comments go for PLtoTF. DVItype is a bit different, though. It uses the file *output* for its main output, so you probably don't want this file associated with your terminal. Hence, if *output* is hardwired to the terminal on your system, you will want to change the macros in module 3 to:

```
@d print(#)==write(type_file,#)
@d print_ln(#)==write_ln(type_file,#)
```

You'll also have to include a declaration of *type_file*, and do a *rewrite* in some other modules.

DVItype holds a dialog with the user to get the values of certain parameters. The files *term_in*, *term_out*: *text_file* are declared in the section **Optional Modes of Output** to be used for this purpose. If, say, your system reserves the pre-declared files *input* and *output* for this function, then you can change the declarations to macros:

```
@d term_in==input
@d term_out==output
```

Pretty sneaky! You can do the same thing if the file *tty* is hardwired to your terminal.

There are more headaches due to differing approaches to I/O on different systems. On many systems, reading a single character from a file is a relatively expensive operation. That is, the time spent doing

```
program slow;
var c: char;
begin
while not eof do begin
    while not eoln do read(c);
    readln;
    end;
end.
```

is a major portion of how long TEX itself takes to run. There's not too much we can do about this if your system does *read*(*c*) via a slow procedure call. However, many systems provide some sort of extension so that you can read a whole line of input at once, more efficiently. For instance, on our system, you can say:

```
var line: packed array [1..80] of char;
    howmany: integer;
read(line:howmany);
```

and the variable *howmany* will get the number of characters actually read in. In any case, all of our programs always read a line at a time into a buffer array (usually in a procedure called *input_ln*), so if a facility similar to the one just mentioned exists in your system, you should be able to use it with TEX by changing just a few modules. (Some people may be able to do this sort of thing by calling a procedure in another language.)

Things are even worse for I/O of binary byte data (TFM and DVI files) and word data (FMT files). Not only might it be inefficient, but I/O of binary data is even less standard than character. Even if your compiler accepts things like:

```
var w: file of integer;
    b: file of 0..255;
write(w,456); write(b,123);
```

you are well advised to check out that these things will work as expected. It is best to experiment with a small program to read and write such files before jumping into the TEX system, if there is any doubt as to how these files will work on your system. Once again, for efficiency's sake, you may have to block things up yourself using an array as a buffer.

Two installation points: There have been some questions on how to run the TRIP test file. To get results that are identical to ours, you'll have to compile a special version of TEX that has some compile-time constants set to values that probably don't match the values you'd want to use in a production version of TEX. In particular, you should turn on the **stat** and **debug** switches, and make the following definitions in your change file:

```
@!mem_max=3000; {greatest index in TeX's internal
    mem array, must be strictly less than max_halfword;
    this is the value appropriate to the TRIP test file}
@!error_line=64; {width of context lines
    on terminal error messages}
@!half_error_line=32; {width of first lines
    of contexts in terminal error messages,
    should be between 30 and error_line-15}
@!max_print_line=72; {width of longest
    text lines output, should be at least 60}
@!dvi_buf_size=800; {size of the output buffer,
    must be a multiple of 8}

@d hi_mem_base=2200 {smallest index in the
    single-word area of mem, must be
    substantially larger than mem_base
    and smaller than mem_max}
```

Finally, TEX's *try_break* procedure is still to big for some people's compilers when the **stat** switch is turned on. We suggest using your change file to put the **stat** code into a small procedure statically embedded within *try_break*, so that you won't have to worry about local/global variables.