

After the *Inventions*...

Federico Garcia

Started April. 17th, 2004

Abstract

Last week a grant was confirmed from the T_EX Development Fund for the “First stage of T_EX*muse*,” defined (by me) as a program capable of typesetting Bach’s *Inventions*.

I am starting this new document (the sequel to the sequel of `firstpass`) as a symbolic articulation point in the development of T_EX*muse*. But the first topic treated here is the direct continuation of the last chunk of work on T_EX*muse*, documented in `secondpass`, which is behind only a coupe of weeks ago.

Note the change—definitive, I hope—in the logo.

Contents

| | | |
|----------|--|----------|
| 1 | On beaming... | 2 |
| 2 | The spacing | 3 |
| 3 | The new system | 4 |
| 3.1 | Anatomy of a character | 5 |
| 3.2 | Anatomy of a line of music | 6 |
| 4 | Stage 3 | 6 |
| 5 | Stage 2 | 7 |
| 5.1 | Line-breaking | 8 |
| 6 | Sticking-out elements revisited | 9 |

| | | |
|-----------|---|-----------|
| 7 | T_EX-METAFONT communication | 10 |
| 8 | Blocks and barlines | 11 |
| 9 | Level-5 elements | 11 |
| 10 | Sticking-out again | 12 |
| 10.1 | Ties | 13 |
| 10.2 | Clef changes | 14 |
| 11 | The new two-pass system | 15 |
| 11.1 | Counting characters | 16 |
| 12 | An unexpected glitch—and a deep revision | 17 |
| 13 | More revision | 18 |
| 14 | Pickup-bars and the new function of ‘ ’ | 18 |
| 15 | The New System! | 20 |

1 On beaming. . .

I am abandoning the ultra-specialized nuances of beaming I had been working on in the last period of *T_EXmuse*’s development. But I am here cashing the gains.

Above all, this is not a retreat. I was very close to a complete conquest of the problem of beaming (with very specific rules I had derived from Finale’s behavior) when I discovered that the rules were too tight. The restriction on the angle of beaming are by no means always as strong as I was taking them to be. Even Finale, in previous versions, was discovered to be freer. What this means is that *T_EXmuse* should be flexible about it. And here, then, we have a potential paradigm for *T_EXmuse*’s general flexibility. Different ‘beaming styles’ should be supported by it, just as *L^AT_EX* allows loading packages.

So the first thing to do is to define formally the mechanism by which *T_EXmuse* ‘loads’ the beaming styles—as a first case of flexibility. This involves distilling what is the ‘core’ and the ‘belt’ of the beaming algorithm. And

I am actually very close to such a distinction: the tracing of the beam, coded as METAFONT functions `draw_stem` and `draw_stems`, is *completely* independent from the algorithms that find the different parameters for the beam.

This means two things: the ‘styles’ are coded as different ways of finding these parameters, and the user can *force* them for special effects or needs.

Thus, styles can be loaded, and created, independently from `texmuse.mf`. They basically (re)define the function `prepare_beam`.

For present needs—to get a program that is able to typeset Bach’s *inventions*—I will use a very simple ‘style,’ of results that are not very different from the real edition of the pieces (by Wiener Urtext), and go on. Or rather, for the moment I will use the style that I developed in imitation of (current) Finale’s behavior. Later on I will retouch it, leave it, or whatnot.

2 The spacing

The spacing has so far been surrounded in a cloud of confusion. In particular, the equations were designed with a ‘desired’ width (of the line) in mind; but the coding has been focused on what will be `\musicbox`, which produces a box with the ‘natural’ width of the music. The following is, hopefully, the definitive account.

Let W be the desired width; L the sum of the widths of the notes with no stretching (the ‘natural’ width of the characters, but ‘natural’ here in a different sense from the paragraph above, where it referred to the natural-but-already-musical width); F the sum of the space-factors in the line; f the space factor of the current character; and l its ‘natural’ length. The stretched width s of each character is given by:

$$s = l + f \frac{W - L}{F}.$$

On the other hand, a ‘musically ideal’ width can be defined (which would be the one produced by `\musicbox`). It would work as the minimum space to the right of a note, and it would be the same for all notes shorter or equal to a sixteenth-note. This is, actually, *rigid* space: it will *always* be added to the notes. Stretching takes place *on top* of this rigid space. Thus, a `\musicbox` will give the same space to a sixteenth-note and to a thirtisecond-note. This rigid space, ‘basic’ hereafter, is part of l . In a `\musicbox`, $W = L$, and therefore $s = l$, (which includes the basic space).

For customization, the ‘basic space system’ is modifiable function of the number of quanta. My first trial will be the function

$$\phi(q) = \max(1, 1.618^{\log_2(q/16)}).$$

$\phi(q)$ is a coefficient. Here it gives 1 when the note is a sixteenth-note or shorter. It makes sense to think of it as multiplying the axis of a note-head (before its rotation, i.e., `regular_axis` rather than `horizontal_axis`). Desirable seems to be that the space between two eighth-notes (i.e., $\phi(32) = 1.618^1$) be enough to accommodate a sharp-sign. This is probably an attribute of the font: widths are such that a sharp-sign (plus two framing spaces) is thinner than the 1.618 times the regular axis of a note-head.

In any case, there remains one problem, a fundamental one. Before any calculation of stretching, the whole natural width of the line must be known. There is no two ways around it. And that cannot be known by `TEX`, nor in fact by `METAFONT` *a priori*. Clefs, signatures, accidentals, etc., affect horizontal space in unpredictable ways. But the horizontal elements make it necessary that the stretching is figured out before drawing the characters.

The only solution is a double `METAFONT` pass. In the first pass, nothing is drawn. Most functions are run, but only in order to know the exact final width of the to-be-drawn characters. `METAFONT` writes then to a file some information about each character, notably its width and its `note_axis`. The idea is that the second pass issues only `beginchar`’s, putting notes, accidentals, articulations, and horizontal elements, already knowing *wd* (and, for that matter, *ht* and *dt*).

The mechanism to avoid collisions can make use of the two-pass system too: the first pass uses a variant of the font that contains horizontal rules for everything, and thus can calculate the extremes of the pictures; the second pass uses the final variants, containing no such lines, because now their location is precisely indicated.

This requires profound changes to the programs. In fact, I am making a copy of the current `texmuse.tex`, `txmfont.mf` and `texmuse.mf`, adding the suffix ‘_2’.

3 The new system

The idea is that the main `METAFONT` file produced by `TEX` loads the subsidiary files *twice*. A new boolean test, `preliminary_`, is set to false the

first time, to true the second one. So every function has a `if preliminary_` section and an `else` section.

3.1 Anatomy of a character

Level 1 Note-heads, rests, and line-opening material; barlines.

For the former, the font should communicate: stem-points, note-axis, and width (of the added element if it were totally by itself—i.e., `texmuse` can handle these values to accommodate elements in unusual positions).

For barlines, `note_axis` and `width` are a fixed values (`bar_left_sep` and the width of the barline plus `bar_right_sep`). Its space factor is 0, so `stretched_width` will always be equal to `width`.

Level 2 Stems and flags. In the first pass, these elements must be horizontally continuous. `texmuse` then finds the new `width`. Qualitatively, these are the only elements that can stick to the right. By the way: `note_axis` does not depend (contrary to appearance) on the final stretching at all.

Level 3 Accidentals and articulations, or brackets (elements to the left). In the first pass, these elements must be horizontally continuous. `texmuse` then finds the new `note_axis`. Brackets will not trigger this if they are applied to the first character in the line.

Level 4 Clef and signature changes. These are added to the left of characters, so that they modify only `note_axis`.

Level 5 Dependant on `note_axis`: beams, ties, slurs.

But only level 5 depends on the stretching of the line. Previous levels add elements to a picture that is centered in `(note_axis,0)`, without reference to `stretched_width`. In fact, a search for the latter in `texmuse.mf` gave hits only in functions related to beaming, making of the line (staff-drawing), and position-finding. In the new scheme, position-finding does not depend on the final stretching, and so those references will not be there. And staff-drawing is, as always, the last thing that is done to the characters, just before shipping them out.

3.2 Anatomy of a line of music

Then, the line of music is done like this:

Stage 1 Levels 1–4 of all characters. The character of index 1 (character 0 doesn't exist) is used for the 'opening-line material:' clefs, brackets, and signatures. The `line_opening` function is triggered by the finding of a level-1 element when `index` equals 1 (`new_char` has just stepped `index`). It cannot be triggered by `new_char` itself because for the first line the clef is not yet defined (it is defined in the first command after `new_char`).¹

Stage 2 Calculation of natural length, stretched length, and the stretching for each character.

Stage 3 Level 5 is added.

Stage 4 Staff-lines are added.

Stage 5 Characters are shipped out.

The best way to implement stage 3, it seems, is to have elements of level 5 write lines to an auxiliary file, specifying the character and all that is needed for the tracing of the element. Stage 3 will then read and execute this file.

This is the new interpretation of the 'two-METAFONT-pass system,' and its only problem is that it does not solve the problem of horizontally continuous accidentals, etc. This question is still open.

Some of the elements in Stage 1 also have to be deferred to Stage 3 (and thus to the auxiliary file) because they depend on the *vertical* spacing of staves: barlines and brackets.

4 Stage 3

Well, well, well... METAFONT has no file-writing capabilities. So, how to implement stage 2?

¹Since `\end@block` is only called by notes or beams—so far—it seems that this way of doing things will work even when there is a clef change at the end of the previous line. We'll see...

In principle, it could be done by all deferring level-5 elements. But of course this would potentially create too-long strings of commands—in fact, it’s a whole other program, a file. On the other hand, elements are known to be of level 5 by definition, i.e., `TEX` is also able to single them out. So it is `TEX` that could write the auxiliary file with them. `METAFONT` would calculate widths, and figure out stretched widths, and *then* read the auxiliary file.

`TEX`’s file will have to be a complete list, because it is `METAFONT` that decides on line breaking. `METAFONT` then reads the file selectively, according to the needs for each line.

The staff lines are part of all this. But only staff-style changes are appended to the list. `METAFONT` draws each staff by default, anyway. The ‘default’ is what changes.

However (and I am writing this on May 16, after I wrote a couple of sections below and pretty much figured spacing and line-breaking out), there are drawbacks to the writing of yet another auxiliary file. What is a string is created for each block? `make_line` would execute all of them. They are presumably not many and long at the same time. They can be erased as they are executed.

So, it’s all a matter of writing level-5 elements not to `to_do`, but to `horizontal_elements[block]`. See section 9 below.

5 Stage 2

So, when `METAFONT` is done appending elements of level lesser than 5, it uses its information about the desired width. The basic equation is, for character i ,

$$s_i = \max \left(f_i \frac{D - W}{\sum f} - x_{i+1}, 0 \right),$$

where D is the desired width, W the total natural width, f_i the space factor of character i , and x_i the amount character i has been moved to the right (extra spacing).

f_i is a function of q_i , the number of quanta of character i (u , given by circumstances, is the number of quanta for which f is 1):

$$f_i = \begin{cases} 1.618^{\log_2(q_i/u)} & \text{if } q_i > 0 \\ 0 & \text{if } q_i = 0 \end{cases}$$

When the desired width is not known (a `\musicbox`), it has to be calculated. In those cases, what is known is that $f_i = 1 \Leftrightarrow q_i = u$, and that the space given to such a note is k (by default the width of a sharp sign). If for the moment we ignore x ,

$$\begin{aligned} q_i = u &\Leftrightarrow s_i = k \\ &\Rightarrow \frac{D - W}{\sum f} f_i = k \\ &\Rightarrow D = k \frac{\sum f}{f_i} + W. \end{aligned}$$

But, of course, $f_i = 1$, so

$$D = k \sum f + W = \sum kf + W.$$

Now, we have added spuriously x_{i+1} to each of the characters' s . It has to be removed with a synthetic judgment: a character is spaced too much to the right if $kf_i > x_{i+1}$. Otherwise, it is not. Therefore:

$$D = \sum (\max(0, kf_i - x_{i+1})) + W,$$

always knowing that $x_{z+1} = 0$.

5.1 Line-breaking

So, one way or another, `desired_width` will be known before the composition of the line, and, with it, the final widths of characters.

But if line-breaking will take place in METAFONT (and so it should, since it is METAFONT that has any information at all on widths), a way is needed for it to be able to estimate whether or not another block will fit in the line.

It doesn't seem worth to implement a system that holds everything in memory until stage 1 of the next block is appended: in any case, the last line will have the potential of spoiling it all. In other words, help from the user will always be needed.

METAFONT should then, at the end of each block, calculate the 'ideal width' (just as if `desired_width` were not know), and then compare it to `desired_width`. It thus has a measure of the stretching that would take place if the line ended there. It then finds estimates the width of the next block, as equal to the average of blocks so far (in the line? in the piece?), and

estimates whether it will be better to include it, or rather to ship the line out as it stands.

Formally: at the end of every block, METAFONT finds the ideal width I of the line so far, as well as $S = D - I$. Being n the number of blocks already composed for the present line, $S' = D - (I + I/n) = D - I\frac{1+n}{n}$. If $\|S\| > \|S'\|$, it will include the next block; otherwise, it will ship the current line immediately.

The ideal width I is the same as the ‘desired width’ is when not specified by the user. From the previous section,

$$I = \sum (\max(0, kf_i - x_{i+1})) + W, \quad \text{with } x_{z+1} = 0$$

6 Sticking-out elements revisited

So, the calculation of how much a `note_axis` has to be moved to the right in order to accommodate left-sticking elements in the note does not any more depend on the stretched width of the preceding character. It is rather the other way around: the distance between the two ‘notes’ (and this notion will be sharpened shortly) should amount, in principle, to the space found by the stretching function, as if no sticking-out elements were present. If there *are* such elements, then the stretching has to be decreased to compensate for the extra space that they implied.

For, as it is now, left-sticking elements *always* add to `note_axis`. In principle, the left extreme of the characters should be 0.² For bare notes, this means `note_axis=1/2regular_space`. So, when a note has a `note_axis` greater than this, the difference will be deduced, in principle, from the `stretched_width` of the previous one.

Similarly, right-sticking notes add to `width`, which in bare notes is again equal to `1/2regular_width`. Any increase will be also deduced from the `stretched_width` (in this case of the present note).

Thus, ‘note’ in the expression ‘distance between notes’ gets defined as that part of the element that is `1/2regular_width` to each side of `note_axis`. The distance between two notes will then be the result of stretching, plus `regular_width`.

²Except for a bare note, which is a `framing_space` to the right. Other elements perhaps include a rigid spacing in this fashion too.

This implies two things: the formulas in the previous section have to be updated to include (deduct) `width-1/2regular_width` where necessary. And nomenclature should be changed. This I'll carry out later, but soon.

The updating of the formulas responds to the new definition of the extra spacing x_i as

$$x_i = \chi_{i+1} + \omega_i - r,$$

where χ is `note_axis`, ω is `width`, and r is `regular_width`.

The new formulas, then, are

$$s_i = \max\left(f_i \frac{D - W}{\sum f} - x_i, 0\right),$$

$$D = W + \sum \max(0, kf_i - x_i).$$

The only extra tweak needed now concerns the fact that the sticking-out elements have been affecting `total_natural_length` (W). But many times they are going to be accommodated within the stretched width's, so that they should *not* have affected W . However, there is no way of knowing this before knowing the final stretching parameters, at the end of a block or of a line—and, in any case, W has to be taken into account for the calculation of D at `end_of_block`.

So, `stretch_chars` must compensate for this. Before engaging in the actual calculation, it has to identify what portion of $\sum x$ can be taken care of by stretching, and then add this to $D - W$ (in fact, subtract it from W).

7 **T_EX-METAFONT communication**

Several things make it desirable for T_EX to be able to point at particular index numbers with complete confidence. But if it is METAFONT that decides when to break a line, T_EX does not know when an extra character (the line-opening one) was appended.

A convention will take care of this: every end of block appends a character, no matter what. Now T_EX will be able to type (0-wide) spaces between blocks, and thus line-breaking will be possible.

A consequence of this is that the opening-line character cannot be such. Clefs and signatures will have to be left-sticking elements to the first notes of a line. A way will have to be designed to properly align signatures when they are different for some staves.

8 Blocks and barlines

The one problem with this is that the new character will affect the spacing in between the limiting characters of two blocks. In any case, blocks will be measures, so they will most of the time feature barlines. But a way has to be found of implementing a ‘transparent’ end of block. It seems not to be as easy as making it a negative-width character, for perhaps `TeX` will not backspace. But a hook function can be devised that reduces the width of the neighboring characters.

So, the end-of-block character will be `0q` if it’s a barline. If not, it will be `-1`, marker of a transparent character.

But see Section 12.

Barlines are somehow to be specified by `\newinstrument`. That is only the intra-staff barline. The interline barline, to be traced at the end of the line (when vertical separations are known), is another thing. So there is a `barline` function that takes an argument: the different kind of barlines. The argument is a three-digit number: hundreds indicate anything that comes before the barline (repetition dots: 1, for example); tens indicate the barline itself (regular: 1; double: 2; final: 3; dashed: 4; etc.); units indicate post-barline elements (repetition dots: 1).

More types (up to 10 for each thing) will be user-definable.

`make_line` always occurs at a block-end. It will be in charge of undoing the spacing of the barline that represents that block-end.

But see Section 12.

9 Level-5 elements

It’s been decided that these elements defer not into `to_do`, but to `horizontal_elements []`. They thus need not address in-character levels in any way: they are met during level 1, defer, and the character deals with them no more.

The puzzle to solve is how to make them append `curr_instr`; appropriately. Let’s talk of a `tie`. It can be that the character meets `regular_stem` before `tie`. Now, the latter will defer to `to_do` (level 2), and `curr_instr` will be set to `""`. So `tie` has no way of appending the right instrument name.

On the other hand, horizontal elements, almost by nature, do not belong to any particular character. In fact, a breakthrough of last year was the discovery that the beams do not have to be drawn onto the last character.

They don’t even belong to blocks (and if `horizontal_elements` is broken into blocks, that is to avoid too long strings in memory). An anomaly to this

is, for example, slurs or ties that transcend the current line. The block in which they start must be known to `make_line`, so that it can ‘break’ them.

But the point is that horizontal elements include, within their parameters, the character indices to which they apply. In the future they will even include the *instrument* to which they apply at each moment. Why not, then, implement this now? A beam adds itself to the current `horizontal_elements` list, including instrument and character indices.

In fact, each beam can have its own list. `beam_1`, `beam_2`, etc. At some point I even thought of transferring the task of compiling the list to METAFONT. But so far METAFONT has not been concerned with rhythmic issues (as the number of beams for each character is). So, T_EX keeps compiling the list, but the character is traced only after the line is finished. T_EX issues a `beam` command, that has the list for argument. For the moment, there is only intra-instrument beams, so the position of this command is relevant: it applies to the current instrument. But the procedure is otherwise independent of `new_char` and `end_char`.

So, to restate: a ‘beam list’ has the following structure:

<index>, *<no. of beams>*, *<no. of noteheads>*, *<notehead, notehead, ...>*

For the moment, `beam` will add *<instrument>* to the list. But the instrument of each character can be indicated by a negative number inserted before *<no. of noteheads>*. If there is no such, the instrument is the same as was for the last character.

There will have to be a way for METAFONT to make a note of a horizontal element being *started* in the current block, to provide for those that will be broken with the line.

10 Sticking-out again

But see 13

The section on sticking out elements above is completely wrong, because it doesn’t take into account the fact that there are several staves. Sometimes a sharp sign should *not* increase `note_axis`, because the previous note in the same staff is far away. `avlbl_space`, `ltx`, and `rtx` have to be revived.

So: all notes, in principle, have both `note_axis` and `width` set to `1/2regular_width`. When a sticking-out element is added, increased are `ltx` or `rtx`. They are how much the corresponding extreme *exceeds* either

`note_axis` or `width`, and in principle they are 0. `avlbl_space` is then calculated, and in case it is not enough, *only then*, either `note_axis` or `width` is increased. After that, it seems, the section above applies (about decreasing them again if `stretched_width` gives more space). However, `width` will *always* remain equal to $1/2\text{regular_width}$, so $x_i = \chi_{i+1} - \frac{r}{2}$.

If a note is adjacent to the previous one (in the same staff), `avlbl_space` is always 0. It only increases with any intervening notes.

10.1 Ties

See, however, the new system

The thing about ties is that the second note must also ‘know’ that there is a tie (for the cases when the line is broken through the tie). So this leads to the more general problem of how to get information from a note to the next.

In principle, there are three things that open a note, i.e., that append `\next@note` to the list of elements: `\open@beam`, a note, and a rest. These three things have to be provided with a hook to insert any other things (articulation groups, that can be nested). With a `\staccato` declaration, for example, all notes should get a dot. Something like `\everynote` comes to mind, but, how to deal with things like the following?

```
\staccato EF\accent G\nostaccato EF\noaccent
```

If `\everynote` is a list this seems possible. For example, being the list empty at first, `\staccato` adds `\@staccato`, but also redefines `\nostaccato` to remove element 1 of the list (or replace it by `\relax` if it’s not the last element). This list is going to be called `\note@hook`.

In fact, there is already a hook: `\this@note`. It is already implementing things like sharps.

But, in addition, the hook I was talking about was not a hook for the notes themselves (as written to `.tms` auxiliary files), but for what `TEXmuse` does when it is writing them. The procedure for the tie is the following: `=` adds a tie to the current note, but in addition has to add a tie closure to the next one. So, before writing the current note, `\this@note` has to be appended `\@tie` (or whatever), and *after* that, it has to be replaced by `\@closetie` (or whatever).

In any case, `\this@note` is called for only by real notes. Rests don’t have it (I can’t think now of something ‘normal’ that carries from a note to the next rest). After having written the note into the auxiliary file, it (as

it stands now) resets it back to `{}`. This is where the change must be. It will reset it to the ‘current’ type of note (staccato, etc.), which is stored as `\@plainnote`. Commands like `\nostaccato` are then trivial: they simply remove the item from the ‘current `\this@note`’. But how to do automatic removals (such as the one that takes place with ties)? This can be done with the `\@afternote` hook (an *executive* hook, something that is *done*, not written, after the current note is written out).

The two latter steps, resetting `\this@note` and executing `\@afternote`, are done by rests too.

The whole business is potentially confusing. To bear in mind is when, respect to the inclusion of `\this@note`, is each function executed. For example, the adding of the tie, which is triggered off by `= after` the note, happens before the next note’s `\this@note` happens. So `\add@tie` has to redefine `\this@note`, not `\@plainnote`.

The actual algorithm for the drawing of the tie is interesting. It makes use of a `tie_list` register for each instrument, in which ties are registered as follows:

`\langle index number \rangle, \langle note \rangle, \langle index number \rangle, \langle note \rangle, \dots`

The tie is opening when the `\langle index number \rangle` is positive, closing when negative.

10.2 Clef changes

This is another thorny issue. So far the clef of an instrument has been handled by means of adding `instr_clef[instr_]` to any reference to notes for that instrument. But of course this doesn’t allow for changes (since `make_line` will still make reference to notes before the last clef change). So there will be a *function* that finds the ‘translation’ of a note into the current clef (current for that particular note).

This is how it works: text variable `clef_changes` records information as follows:

`\langle index \rangle, \langle instrument \rangle, \langle old clef \rangle`

Yes: *old* clef. Similarly, clefs are recorded last-to-first. (You’ll see why.)

Every line resets this variable to `unknown`. If there is a change within the line, it will be added to it. Then, any reference to a note will execute the function `_note(note, instr, index)`. In principle it equals `instr_clef` (which is the clef *at the end of the line*), so it works its way back the line to find what old clefs were in force when.

This procedure is to be used only for level-5 elements (since only those can potentially involve notes typeset in different clefs). For the rest, `instr_clef` still works the same way.

11 The new two-pass system

The clef changes, that as solved above work perfectly within blocks, pose a problem when they happen as the first thing of a block. In fact, the new clef should be printed behind the barline, but there is (as yet) no way that `TEXmuse` learns of its presence before the barline is traced. The case of end-of-line barline, moreover, bars the possibility of modifying the barline character after it has been shipped out a first time.

On the other hand, the progress done with the inventions has led me to face the ever-latent problem of running out of characters in a single font. There is also the theme of `TEX`'s handling of line breaks, which are created by `METAFONT` but are unknown to `TEX`.

All of that suggests the need that `TEX` predicts some of what happens in `METAFONT`. There is already a setting in which to implement this: the first scan that `TEX` performs of the user's input, so far only for quantization purposes.

Since it is `TEX` that adds `new_char` commands to the `METAFONT` files, it is clear that it can itself count the characters. Thus it can deduce when to split files, because a given block will overflow `METAFONT`'s 256 limit. But, of course, it can happen that this splitting happens in the middle of a line, and `METAFONT` will be forced to insert a break and over-stretch the characters in it. The issue of splitting files, therefore, is connected, and logically posterior to, the issue of line-breaking.

Thus, a two-pass system is after all needed. The first pass will be done without any precautions, although splitting font files as necessary. But, before printing the music, `TEX` performs a test of the resulting line-breaks. How can it identify unsatisfactory line-breaks?

- A line can be *underfull*. The most common instance will probably be the splitting of `METAFONT` files in the middle of a line, and the case when the last measure falls into a line by itself. The telltale is of course that the average width of each individual character in the line is very large.

- There is another kind of *underfull*: the previous was a ‘METAFONT-underfull.’ But when METAFONT is forced to fit too many characters in the same line (mainly because it didn’t count on an especially wide character in the on-coming block), i.e., when the natural width of the characters in the line is in itself greater than the desired width, T_EX will find an overfull. It may happen that it decides to break the line to avoid the overfull. Of course, the whole layout is disturbed by that.

It is important to avoid that. Otherwise, if something like that happens in line 1, all other lines will be disturbed, and all of T_EX’s research on them will be wasted. The postulate to work on is that METAFONT will never produce a line that is narrower than the desired width. Thus, if the right margin is increased by a certain quantity (which would be a user’s parameter), and the text set to the left, most of the output will be ‘normal.’ Those overfull lines will, in fact, look like just such. This converts these cases into the next class.

- T_EX can easily see regular *overfulls*. It then decides to break it in the next pass. To have in mind is that this is an extreme case: METAFONT will have stretched 0: the characters are *really* lumped together. That means that the next pass will probably produce a satisfying line.

11.1 Counting characters

The information needed is contained in the matrix. Reading it T_EX can now how many characters each block will need, and therefore how many blocks will fit in each font file.

There is a control sequence `\music@font@splits`, that contains the last block of each font file. For example, if the first file is able to accommodate 14 measures, the second file measures 15–28, and the third measure the rest of the piece, `\music@font@splits` will be `{14,28}`.

When T_EX finishes each block, it scans `\music@font@splits`. If the just shipped-out block is the first number in the list, then the current font is finished, another one started, and the number is removed from the list.

12 An unexpected glitch—and a deep revision

After finally making it to the end of the first invention (June 21, first print out of the whole piece), I set out to complete things. First, implement the flats.

And—lo and behold! It so happened that one of the first flats of the piece, on B in the left hand, has a set of specific characteristics whose infrequency had so far disguised a subtle problem with sticking-out elements. This flat came within a beam, and after a leap of a fourth. As a result, the flat itself didn't collide with the previous notehead. It did collide with the stem, but, of course, the stem is not there when METAFONT is checking for collisions. So, the flat did collide.

The solution was at first sight easy: sticking-out elements, if there is no available space from the previous note, have to be moved *at least the previous note's width*.

This was hard to picture at the moment of making the change. But eventually I got it. Now the flat did not collide with the previous stem.

But all sharps after a barline had been moved too! The space to the right of a barline was a rigid space specified by `width:=regular_width`.

To avoid this, I set the barline's width to 0, and proceeded to make it stretch just as `least_spaced` (a sixteenth-note by default). But this meant changing `sf_` (f in Section 3's nomenclature): now $f(0) = 1.618^{\log_2 1} = 1$. The last barline in the line resulted stretched too. And the new version of f didn't give 0 (the needed stretching) in any case.

So, back to $f(0) = 0$. But this is going to apply to the last character *only*. In-line barlines have a $q = u$. In fact, T_EX was writing `new_char(least_spaced)` for barlines. But `make_line` used the old value of 0 to identify barlines (and draw inter-staff barlines).

The final solution is the following: $f(0) = 0$. But in addition $f(m) = f(-m)$. T_EX writes `new_char(-least_spaced)`. And `make_line` identifies elements with $q \leq 0$.

There was another option: make `rtx` of a barline a negative value, so that the available space is (artificially) increased. Careful mode would not even be entered with a sharp after a barline. This seems to be more elegant. But, apart from who knows what side-effects this could have, I think the space

after a barline should not be rigid. Still, this is a worthy option if negative quanta are to be done any special meaning from now on.

13 More revision

It cannot be assumed that all notes are in principle `1/2regular_width` wide: there is barlines. I think the only thing for which I was assuming that, however, was the calculation of x for the spacing.

It's fixed. The equations follow what eventually will appear as the second article on *TUGboat*.

14 Pickup-bars and the new function of ‘|’

After having implemented the flat-key signatures, I went on to typeset more inventions. And soon it became apparent that using `|` for the beaming was confusing. It's better to use `[` and `]`, that give a sense of opening and closing. That was easy to implement.

Now I face the issue of the pickup-bar for the *Invention in D*. I had already thought that now that `|` was available, it could be used for a barlining that overrides `TEXmuse`'s automatic ending of blocks. Well done, it will be good for example for clef changes that apply to the barline—a problem I have not solved yet.

The difficulty of course is that when the block is automatically closed, it's already ‘shipped out,’ both for the quantization and for the writing of METAFONT files. Some kind of ‘marker’ will have to be implemented, i.e., something that tells `TEXmuse` that the user has renounced typing the barline explicitly. Everything should be possible before a new note, or anything that is clearly the contents of the following measure, occurs.

But it is not so easy as catching the first raise in `\current@quant`: beams, for example, are clearly the contents of the next measure, but do not increase `\current@quant`.

Does it seem to have to be arbitrary? In this way, it would be only a beam-opening, a note, or a rest, that executes the ending of the previous block. (In the new model, they would silently trigger off the events that take place when a `|` is read.)

On the other hand, the | could operate even more elegantly: by saying ||, a double bar is produced (but the final double bar should be some other thing, like \|). ‘:||:’ is another combination. And so on. . .

So, there is an automatic behavior that takes place when *material* from the next block is encountered: by default, it is the execution of |, but it could be changed into anything.

The place to do this, it seems, is the matrix. Certain elements, such as clefs, key or time signature changes, commands on line-breaking, and so on, produce—if \current@quant is 0—a negative-quantum entry in the matrix.

There is two kinds of such pre-bar tokens: some affect only their staff (clef changes, for example); others affect the whole ensemble (bars, to begin with, but also no-line-breaks, time- or key-signature changes). So, just as all the notes, rests and opening beams mean \quant@note, pre-bar tokens will mean \quant@bar.

The mechanism uses a new conditional, \ifblock@done. It is made true only by *material* or by a barline command (like |), and false when the block is done. This can be done with a new definition of \end@block, which was what any material executed when it encountered 9998. But now it will execute when it finds 0, and will close *the previous* measure.

Note: \musicbox changed into \music in *texmuse.tex*. And the different parts of it are now different commands.

One problem for the pickup is that later T_EX calculates the space factor of characters from the matrix. So, the last note of the pickup bar will wrongly be assigned a very large space factor. To solve this, it occurred to me to insert a matrix line full of 0’s where the barline takes place. It doesn’t seem to have an effect, for nothing is done when a matrix line is 0 (this is at \compute@char). The adding of a 0’s-matrix line is a general function similar to \@note, but called \no@note, that is executed at each bar.

But then the convention of 9998 as the end of a measure is not needed (and, in fact, is a hindrance). The only place—not connected with quantization, which is already solved with a new \@note—where a comparison with 9998 is made is \count@chars, to count measures. But measures can be counted by matrix line that start with 0. It is true that relevant comparisons are also made not explicitly with 9998 (\block@period>\current@quantum), but it seems that having a 0’s matrix line with the \block@period) value works as well. Also \build@block compared to 9997, but now it compares to \block@period.

Pickup bars, though, leave a 0's matrix line at other than `\block@period`. So it's a matter of ensuring that they will *also* leave one at `\block@period`.

Quantization already prepared for all this, now what to do about the characters. One thing seems clear: each end of measure (currently caught at `\build@block`) triggers off a number of events. The default is that it is a regular barline, and that it can be a line-break point. Anything could be added to it (clefs, etc.). And those two things could be overridden. So, `\end@measure`, by default, has `\@barline#1` and `\@break#1`. The default values of these are both 1 (regular barline and possible break-point). Anything can be added.

Changed has been `\find@sf`: no longer adds space factors together.

15 The New System!

OK, it seems now all this works. The new system is much more natural than the previous one. First of all, the matrix doesn't have 9998's. End of measures are found because the next quantum number is 0.

Then, the auxiliary files have the notes of a block in a line, and the next line is its closing. A usual, default one, would be only `'\@bar@line{10}'`. But I have already tried a clef change. In that case, that line in the auxiliary file will be `'\@clef 6\@bar@line{10}'`.

`\build@block` reads the actual block-line in those files. But when it finds that the next matrix line has 0 quantum, it executes `\finish@block`, which reads those closing-block lines.

The different kinds of breaks are handled as follows: there is the counter `\@@break`, which by default is 1. `\finish@block` has an `\ifcase\@@break` at the end. When `\@@break` is 1, for example, it writes `end_of_block` into the METAFONT file. If it is 2 (forced break), it writes `make_line` (or whatever, I have to check what is exactly needed).³

When `\build@block` finds a 9999, that's the end of the piece. So it will execute the default final-bar code. This is: double bar (`\@bar@line(20)`) and `\@@break{3}`—an 'ideal-width' break (no justification for the last line). This default is different for the different kinds of user commands (`\music`, `\musicbox`, etc.), and in any case user commands will be provided to change them. But they don't have to be too much flexible: the user can always

³The final value is set by the last instrument.

override by appending the necessary staff at the end of the input files, and making that default void.

In addition, `\no@note` (a matrix number appended by barlines) has an argument: `-1` will tell `\count@chars` not to count it as block end. And `\build@block` will still ignore that matrix entry.

Accidentals are ‘material’: once they are met, a new block is started. That consists of closing the previous bar, and setting `\current@quantum` to 0. It’s in a new function called `\open@block`. So far, accidentals, rests, notes, and beam-openings call it.

Ties create a special case: `\close@tie` (or whatever) is added to `\this@note` by `\add@tie` (or whatever). So, when the tie goes over a barline, `\end@block` is encountered before the next note, and *it* gets the `\close@tie`. It is conceivable that this is not going to be the only such case, where information goes from a note to the next note *bypassing any barlines*. For example `\staccato` should apply only to real notes, not to barlines (or to rests, for that matter). Clef or signature changes, on the contrary, do apply to barlines too.

So, `\this@note` is going to be used for things that apply to everything. `\@atnote` only for notes. And `\@@atnote` for notes and rests (like tie closures). For the moment, they will work only once (`\add@note`, `\@@@rest`, etc., reset them to nothing), but eventually three kinds of `\plain@note` will be implemented, so that `\@atnote` is reset to one of them, `\@@atnote` to another, and so on.