

# TEX*muse*'s Main Loop

Federico Garcia


August 31, 2005

This document accompanies the program `TEXmuse` in its first submission to CTAN. The program is incomplete, but as it stands it can be called a ‘first stage’. But there is no `dtx` file as with a complete package, that self-extracts and documents at the same time.

All the same, I wanted to provide something in the guise of a documentation. This text contains a description of `TEXmuse`'s `TEX` part (there is an equally substantial `METAFONT` part), that makes clear at least the central ideas of a somewhat unfriendly code.

If you are looking for a guide to the user, the best is to go to the end, where there is a sample with the user's input, as well as to the other samples (Bach's *Inventions*) that are on-line with the package. There are also other documents (a guide to installation and running and a list of shortcomings and wants).

`TEXmuse` has been described in *TUGboat* **24/2**. A second article will appear in the next issue of *TUGboat*, containing pretty much the present document. `TEXmuse`'s webpage is <http://www.fedegarcia.net/TeX/TeXmuse.html>. This first stage was in part possible thanks to a grant from the `TEX` Development Fund. Thanks!

Let's begin with a simple case: . This is the result of the input string `5C4[EC]5GG`. The numbers tell the rhythmic value (5 for the quarter-, 4 for the eighth-note), and the letters tell the pitch. ‘[’ and ‘]’ create the beam of the second and third notes.

Of course, there are several C's—the reference to ‘C’ as the first note would not be sufficient to fix it where it appears in the example. `TEXmuse` was able to fix it as `C5` (`C4` is the so-called ‘middle-C’, and this one is an octave higher

than that) because, in truth, the input started with `\rangefrom{G4}`. This set G4, A4, B4, C5, D5, E5, and F5 as the default for each pitch. The code for this, not devoid of interest, is in lines 165–185, with an auxiliary function defined at 290.

But that is not part of *TEXmuse*'s main loop. The latter is rather about what happens to the input at the different stages.

## Quantization and the ‘matrix’

The first time *TEXmuse* collects rhythmic information: how long each note is. In our first example, the result is the following ‘matrix’:

---

```

0:1.
64:1.
96:1.
128:1.
192:1.
9999:0.

```

---

This simple list registers the following facts: the first note is at position 0; since it is a quarter-note, it has a ‘quantum value’ of  $64q$  (the ‘rhythmic quantum’ is defined as the 256<sup>th</sup>-note, a note with 6 flags, that naturally gets  $1q$ ). Consequently, the next note, whatever it is, will be at 64. In this case it is an eighth-note, with a value of  $32q$ . Therefore, the next note appears at 96. And so on.

All lines, in addition to the quantum position, have a 1. Each of them means that there is a note at the indicated position. The exception is of course the final 9999, that has been added as a marker of the end.

The translation of the input into such a quantized matrix is the first stage of *TEXmuse*'s main loop: the ‘quantization’. Its true significance is actually seen only when there is more than one staff. Look at this other example and its matrix:



```

0:1,1.
64:1,1.
96:0,1.
128:1,1.
192:1,1.
256:0,0.
9999: 0.

```

Here there is an instances of a positions with ‘0’ instead of ‘1’. The left hand (bottom staff, first item after ‘:’ in the matrix lines) does *not* have a note at position 96. And, on the other hand, even though the left hand has a quarter-note (actually a quarter-*rest*, but for the purposes it’s the same) at position 64, the character is not one of a quarter-note (64*q*), but of an eighth-note (32*q*).

The presence of 256 in this matrix, if you are wondering, is due to the inclusion of the barline (as a result of ‘|’ in the input).

Note that `\rangefrom` has no meaning for the construction of the matrix. As far as the matrix is concerned, all notes, and even rests, are the same—there is no difference between C and D. Most other things, like accidentals (flats, sharps, naturals), ties, or any other such additions, also mean nothing for the matrix.

In fact, in lines 280–7, where `TEXmuse` is preparing things to ‘quantize’ the input, it declares

```
\let[\relax\let]\relax
\let|\quant@barline
\letA\quant@note\letB\quant@note\letC\quant@note\letD\quant@note
\letE\quant@note\letF\quant@note\letG\quant@note\letR\quant@note
\let\rangefrom@gobble
\let#\relax\let\n\relax\let\b\relax
```

Quantization is one of the things that happen *always*. `TEXmuse` has a number (and will have more) ways of actually typesetting the musical text: `\music`, for example, makes long musical text be breakable at the end of lines, and appends a final, double bar at the end. `\excerpt`, on the contrary, typesets the music in a single line. (This latter is what I have been using for examples so far). Eventually, a `\musicbox` will allow setting the exact width of a box, and `\musicparbox` will be a combination of `\music` and `\musicbox`.

But, in any case, *all* of these functions carry out the ‘quantization’. It is, in fact, the first part of `TEXmuse`’s main loop, and is done by the command `\@quantization` (defined by lines 411-36).

## The auxiliary files

The second stage is actually also carried out by `\@quantization`, since this allows performing only one loop through the different staves’ input. But it

must be considered a separate stage. It consists in the writing of ‘auxiliary’ files.

One auxiliary file is created for each instrument (i.e., each staff). These, for example, are the files for `\shortl` and `\shortr`, the two staves of the last example:

---

```

                                \next@note\q@n\add@note{c3}\@stem
                                \next@note\q@n\@rest{0}\relax
shortl.tms:                    \next@note\q@n\add@note{c3}\@stem
                                \next@note\q@n\@rest{0}\end@of@block
                                \@bar@line{10}\relax

```

---

```

                                \next@note\q@n\add@note{c5}\@stem
                                \next@note\open@beam\en\add@note{e5}\add@to@beam
shortr.tms:                    \next@note\en\add@note{c5}\add@to@beam\close@beam
                                \next@note\q@n\add@note{g4}\@stem
                                \next@note\q@n\add@note{g4}\@stem\end@of@block
                                \@bar@line{10}\relax

```

---

Although it cannot be visually shown here, one of the most important conventions of *TeXmuse*’s is that each ‘block’ is set to a single line in these files. Both of the files shown have actually only two lines (here set to the left margin of the listing), one starting with `\next@note...`, and the other containing `\@bar@line{10}\relax`.

As you can see, each of the ‘blocks’ (conceive of blocks as equivalent to measures, for the moment) is basically a list of notes, separated from each other by `\next@note`, and closed by `\end@of@block`. Each item in the list—i.e., each note—is composed of several elements:

`\next@note` ‘value’ ‘additional stuff’ ‘note’ ‘rhythmic notation’

The ‘value’ is the *rhythmic* value: eighth-note, sixteenth-note, quarter-note, etc. In the example we have instances of `\q@n` (quarter-note) and `\en` (eighth-note).

The ‘note’ is the note itself—or the rest. So this is always either `\add@note` or `\@rest` (with their arguments). When the ability for several note-heads per note is implemented, this will be repeatable.

The ‘rhythmic notation’ refers to the graphical way a rhythmic value is actually represented. Quarter-notes, for example, are represented by a (bare)

stem: `\@stem`. Lonely eighth-notes (i.e., eighth-notes not beamed together) have both a stem and a flag, and their ‘rhythmic notation’ part would be `\@stem\@flag`. Rests don’t have this element.

‘Additional stuff’ can be appended to a note’s entry. In our very simple example there is only a few instances of this: `\open@beam`, that opens the beam in the second note of the right hand, and `\close@beam`, that closes it in the third. The corresponding `\add@tobeam` commands, that apply to all notes under the beam, is *not* additional stuff, but rather the ‘rhythmic notation’ of those notes.

So, to compare, let’s tweak the example a little bit (with all due respect to Kuhnau, who I think composed the beautiful sonatina from which the example comes). The input for both staves (`\shortrii` and `\shortlii`) appears at the right:



```
\shortrii{\rangefrom{G4}5C4\bec5GG|}
\shortlii{\rangefrom{C3}5CR\trebleclefC+R|}
```

The following are the five notes of the right hand. New are `\@flag` and `\@flat`:

	value	add. stuff	note	rhythm. not.
<code>\next@note</code>	<code>\q@n</code>		<code>\add@note{c5}</code>	<code>\@stem</code>
<code>\next@note</code>	<code>\e@n</code>	<code>\@flat</code>	<code>\add@note{e5}</code>	<code>\@stem\@flag</code>
<code>\next@note</code>	<code>\e@n</code>		<code>\add@note{c5}</code>	<code>\@stem\@flag</code>
<code>\next@note</code>	<code>\q@n</code>		<code>\add@note{g4}</code>	<code>\@stem</code>
<code>\next@note</code>	<code>\q@n</code>		<code>\add@note{g4}</code>	<code>\@stem</code>

And the 4 notes of the left hand (new is the result of `\trebleclef`, `\@clef`):

	value	add. stuff	note	rhythm. not.
<code>\next@note</code>	<code>\q@n</code>		<code>\add@note{c3}</code>	<code>\@stem</code>
<code>\next@note</code>	<code>\q@n</code>		<code>\@rest{0}</code>	
<code>\next@note</code>	<code>\q@n</code>	<code>\@clef0</code>	<code>\add@note{c4}</code>	<code>\@stem</code>
<code>\next@note</code>	<code>\q@n</code>		<code>\@rest{0}</code>	

(The clefs at the beginning, as well as the bracket, are drawn by METAFONT as part of the automatic beginning of the line. Neither is represented by the auxiliary files.)

All the commands in the auxiliary files are private, and the user knows nothing about them (although, of course, the list can be seen and read by him, maybe a good thing for debugging his files). They are the commands that the third stage understands.

## The METAFONT files

Everything is ready now to write the METAFONT file that will draw the characters. (This is done by command `\mf@files`, defined in lines 437–5.)

The first line in the matrix, that corresponds to the first note, is `0: 1,1`. The note is at position 0. Now, how long is it? (The answer will mean, typographically, how much space the note will receive when the line is stretched.) Since the next line in the matrix is at 64, the first note is  $64 - 0 = 64q$  long. So, the character for the first note can be opened: to the METAFONT file is added the line

```
new_char(64);
```

Now, what does this note contain? Because of the first 1 in the character's matrix line, it is known that the first staff (the bottom one, in this case `shortlii`) has a note in this character. So, a line is added that announces it:

```
shortlii;
```

Next, the drawing of the `shortlii`'s note itself. The auxiliary file (remember there is one for each staff) tells the program that the first note is `\q@n\add@note{c3}\@stem`. This string of commands, when executed by `TEXmuse`, result in the following lines in the METAFONT file:

```
add_noteheads(c3);
regular_stem;
```

The first staff is ready. Since there is also a '1' for the second staff (`shortrii`), it will be announced and its note, namely `\q@n\add@note{c4}\@stem`, drawn:

```
shortrii;
  add_noteheads(c5);
  regular_stem;
```

With that the first character is finished. A final `end_of_char` closes it.

One note from each auxiliary file has been 'consumed'. It has in fact *disappeared* from the list: `TEXmuse` deleted it. Now the register that holds

the list of notes (i.e, the one containing ‘\next@note\q@n\add@note{c3}...’) starts from (what was) the *second* item. The next time  $\text{TEXmuse}$  finds a ‘1’ in the matrix that corresponds to that staff, but not before, this (second, now first) note will be executed, and, it too—well, ‘executed’. (By this procedure,  $\text{TEXmuse}$  does not have to ‘count’ the items in the list or in the matrix.)

The second character, with a matrix line of 64: 1,1, lasts only  $32q$ , because the next matrix line begins with 96. But it too has notes for both staves (‘1,1’), which are the simpler \q@n\@rest{0} and the more complex \e@n\@flag\add@note{e5}\@stem\@flag. So the second character in the METAFONT file is written as follows:

```
new_char(32);
shortlii;
  add_rest(0,0);
shortrii;
  add_noteheads(e5);
  add_flat(e5);
  regular_stem;
  add_flag(1);
end_of_char;
```

The third character has a matrix line of 96: 0,1. It will only have a note for the second instrument. The list of notes of the *first* instrument, then, will not be read. Its next item will then *not* be deleted: whatever it is, it remains there until another matrix line actually invokes it. The third character is, then:

```
new_char(32);
shortrii;
  add_noteheads(c5);
  regular_stem;
  add_flag(1);
end_of_char;
```

A quotation of the whole METAFONT file for this example is at the end of this document. The initial declarations and initializations to be found in it are written by \mf@files, in a subroutine called \mf@@headers (not part, actually, of ‘the main loop’).

## The printing of the music

The loop is completed when T<sub>E</sub>X (trusting the METAFONT part of T<sub>E</sub>X<sub>muse</sub> to have correctly drawn the characters) types the characters. This happens at `\@compose` (lines 514–31), and has a very simple form: it’s a loop that selects the new font (`\@musicfont`) and types the characters one by one (`\char\the\char@no`) up to the last one. I can repeat that right here, adding commas between characters:



## Detail: the input

The main loop can be fairly easily explained. Its actual code in `texmuse.tex`, however, is far more obscure. That’s due to a number of complications that occur at every stage. One of them is the fact that the user’s input can contain more items than simply notes and rests.

In fact, each note can be subject to a series of modifiers. I’m not talking here about accidentals (flats, sharps, naturals), which are another subject. Apart from those, notes can be modified, for example, as it comes to ‘register’. As we saw, `\rangefrom` defines a ‘default’ register for all notes—say, `\rangefrom{C4}` defines all C’s in the input to refer to the middle C. But there might be a need for a note in a *different* register—say, C5 instead of C4. This can be achieved, instead of issuing another `\rangefrom`, by typing a + after the note, which to T<sub>E</sub>X<sub>muse</sub> means ‘set this note an octave higher than the current range’. A - means an octave lower, and it is also possible to apply *many* of these modifiers to a single note.

What happens to thus-modified notes in the first stage—the ‘quantization’? For this stage, all notes are the same: they simply produce an entry in the rhythmic ‘matrix’ of the piece. You remember from above that in the first stage all notes are interpreted as `\quant@note`. But in order to provide for modifiers, `\quant@note` is still not an ‘executive’ function (one that, for example, adds the entry to the matrix), but merely a directive. This is its real definition:

```
\def\quant@note#1{\let\@let@token=#1\@quant@note}
```



This reads the next token, and invokes `\@quant@note`. This latter examines the next token:

```
\def\@quant@note{\let\next\@let@token
  \ifx\@let@token-\let\next\quant@note
  \else\ifx\@let@token+\let\next\quant@note
  \else\ifx\@let@token.\let\next\quant@dot
  \else\ifx=\@let@token\let\next\@@quant@note
  \else\def\next{\@@quant@note\@let@token}%
  \fi\fi\fi\fi\next}
```

In the case of a ‘bare’ note (no modifiers), none of the tests carried out by `\@quant@note` will succeed. The token next to the note (that was read by `\quant@note`) has to be returned to the input string—which is done by the ‘default’ `\let\next\@let@token`.

If the next token was actually one of our modifiers, - or +, they should be gobbled (for, in this stage, they are ignored). Now, since there might be *two* (or more) of those modifiers, the whole test has to begin again: `\let\next\quant@note`. This is also the case with another modifier that doesn’t affect quantization, namely = (that puts a ‘tie’ to the note as in



Thus the note is stripped of all modifiers (ignore the dot for the moment). And *then* an executive function—one that *does* something—is called, namely `\@@quant@note`. (This happens also when the note had no modifiers in the first place.) What this executive function does is to add ‘`\@note`’ to the token register in which the quantization tokens are stored:

```
\def\@@quant@note{%
  \global\quant@toks\expandafter{\the\quant@toks\@note}}
```

Later, when these quantization tokens are read, it is `\@note` that adds the relevant entry to the matrix.

A modifier that does have an impact on the first stage, because unlike + and - it has a rhythmic consequence, is the dot. The dot makes a note ‘dotted’: a quarter-note that has a dot after it becomes a ‘dotted-quarter-note’. In music, a dot means that the note lasts half as long again. The ultimate expansion of a dot at this stage is ‘`\quant@@dot\@note\quant@@undot`’. But not before a test to see if the note is, in addition, tied (or, possibly but still not completely implemented, double-dotted):

```

\def\@quant@dot{\let\next\@let@token
  \ifx\@let@token.\let\next\@quant@dot      % Not sure it works
  \else\ifx=\@let@token\let\next\@@quant@dot
  \else\def\next{\@@quant@dot\@let@token}%
  \fi\fi\next}
\def\@@quant@dot{\global\quant@toks\expandafter{%
  \the\quant@toks\quant@@dot\@note\quant@@undot}}

```

Something similar takes place at the second stage: conversion of the input into commands for the auxiliary files. The ultimate expansion of any note at this stage is an `\@add@note` command (which is the one that adds a note to the auxiliary files, as described above). But, again, this doesn't happen immediately.

What a note (letters A, B, . . . , G) in the input does is to set `\@pitch` (to the corresponding letter, a, b, etc.) and `\@octave` (according to the range defined by `\rangefrom`). After that, it launches `\@@octave`. This is the function where tests for modifiers happen:

```

\def\@@octave#1{\let\next\@@octave
  \ifx#1-\advance\@octave-1\else\ifx#1+\advance\@octave1\relax
  \else
    \if@tempswa\else
      \@add@note{\@pitch}{\the\@octave}\@tempswatrue\fi
      \ifx=#1
        \add@tie{\expandafter\@pitch\the\@octave}%
        \let\next\relax
      \else
        \ifx#1.\add@dot{\@pitch}{\the\@octave}%
        \else\let\next#1%
        \fi
      \fi
    \fi
  \fi
  \next}

```

So, if a note is bare, `\next` will be set to whatever the next token is. If the note, on the other hand, has - or +, `\@octave` is changed and `\@@octave` is tried again, to see if there's more modifiers. (It is because of this double try that `\@tempswa` is used.) For all other modifiers, '.' and '=', the note itself

(its pitch and its octave) is set, and only additional commands (`\add@tie` or `\add@dot`) are executed.

So, every note, be it simple as ‘C’ or compound as ‘D++.=’ (a D two octaves higher than set by `\rangefrom`, dotted, and tied), gets translated into a series of commands, that always includes `\@add@note` and can in addition include `\add@tie` or `\add@dot`. These commands are executive: they write in the auxiliary file of an instrument.

Rests, input by ‘R’, are very similar, but they cannot be tied (no test for ‘=’) and their modifiers + and - do not mean change in octave, but vertical shifting. They use, then, functions `\@@rest` and `\@@@rest`, to be finally converted into `\@note` in quantization and `\@rest` in auxiliary files.

But beaming acts differently. [ and ] have no meaning in quantization (and in the first stage they are `\relax`), but the auxiliary files have to reflect them. They, however, cannot have modifiers, so their function is immediately executive. ‘[’ opens a beam by writing (in the auxiliary file) `\open@beam`, and making all following notes part of that beam—which amounts to changing their rhythmic notation. ‘]’ closes the beam by adding `\close@beam` and setting the *rhythmic notation* back to the note’s natural way.

What *is*, though, this ‘natural’ way? It depends on the (rhythmic) kind of note. It is set by the *numbers* in the input. A ‘4’, for example, means ‘eighth-note’, and will set both duration (for quantization purposes) to be  $32q$  and *rhythmic notation* to be ‘a stem, and a flag, please’ (`\@stem\@flag`). A ‘5’ means quarter note, and makes the duration  $64q$  and the *rhythmic notation* only a `\@flag`. These are ‘global declarations’, affecting anything that comes after them until another, overriding one is encountered.

And numbers set also, of course, the *value* of a note.

There is something else about the input. Any *spaces* in the input will not be typed into the actual music: the music is typed mechanically by T<sub>E</sub>X at `\@compose`, and it does not include spaces. But the input—spaces and all—is read, and twice, at `\@quantization` and `\mf@files`. The spaces there *will* create spaces that disturb the final layout (unless the music is the first thing that appears in the line, and then the spaces are ignored because T<sub>E</sub>X is in vertical mode).

So, the spaces have to be stripped off from the input. This is carried out just before starting the `\@quantization`:

```
\def\strip@spaces#1{%
```

```

\@temptoks#1%
\@temptoks\expandafter{\the\@temptoks{\relax} }%
\quant@toks{}%
\@tempwatrue
\loop\if@tempswa
    \expandafter\strip@@spaces\the\@temptoks{\relax} \@nil
\repeat
#1\quant@toks}
\def\strip@@spaces#1 #2#3\@nil{\quant@toks\expandafter{%
    \the\quant@toks#1}%
    \ifx#2\relax\@tempswafalse\fi
    \@temptoks{#2#3}}

```

Command `\strip@spaces` is invoked by `\@quantization`, and its argument is the token register where the input is stored. Two temporary token registers are used, `\@temptoks` and `\quant@toks`, to get portions of the input stream separated by spaces. At the end, because of the way `\stripped@@spaces` is executed, it will find `#2` to be `\relax`, and call the whole operation off. Then, whatever was left in `\quant@toks` will be put back into the original register. (`\quant@toks`, on the other hand, is a register defined for other purposes—quantization—but it can be used here because what it had before, if anything, is of no interest anymore. Thus a token register is saved.)

## Measures, blocks, and automation

A piece of music is usually wider than a sheet of paper: it will have to be broken into lines (which, incidentally, are called ‘systems’). This cannot happen anywhere, but at special points: usually at the end of measures (or ‘bars’). Sometimes, however, for music with very long measures, a line-break can be made within a measure. And sometimes, in modern music, the piece simply has nothing like ‘measures’ (because it’s music without regular meter). So the default correspondence `measure`  $\Leftrightarrow$  `line-breaks` cannot be wired in *TEXmuse*. Rather, *TEXmuse* thinks in terms of ‘blocks’.

Usually, however, there is a meter, and measures do coincide with blocks. This must definitely be the default. But, in addition, since the meter is regular (that’s its essence) and automatic, the typist should not be forced to explicitly say where a measure ends. *TEXmuse* has to be able to figure that out by itself.

Other reason for this automation is that we must avoid the matrix to increase its quantum numbers indefinitely. If it goes by measure, it can healthily reset to 0 every once in a while. And likewise, the auxiliary files should not be overloaded with indefinitely long lines. For all that, some kind of ‘modularity’ is good, and measures provide the best basis for it.

This has a number of consequences on `TEXmuse`’s main loop. Going backwards, the last stage (composition) must typeset blocks separated with a discretionary break, so that when the end of the line is reached `TEX` goes to the next line. Within blocks, however, the characters must be typed one after the other, as single words that won’t be broken.

For the penultimate stage (writing of `METAFONT` files) there is also a consequence of measures. In all probability, a piece will have more than 256 characters, so that more than one font will be needed. But `METAFONT` draws the characters line by line (because it needs to know the stretching of each character before actually shipping them out). So `TEX` has to write the `METAFONT` files block by block, so that the end of each file always coincides with a possible line-break. (In fact, as the system stands today, the end of a `METAFONT` file *is* a line-break. User’s control on this is one of the things to improve.)

The writing of the auxiliary files also goes block by block, because the auxiliary files have to contain special characters for the end of the measures—barlines—that will not be present in the user’s input.

And the compilation of the rhythmic matrix, of course, needs blocks to go back to  $0q$ .

This ‘going by blocks’ is actually responsible for most of the intricacy of `texmuse.tex`. To start develling it, let’s discuss the workings of `\@add@note`. This is the function into which notes in the input are translated by the second stage, and whose main action is to add notes to the auxiliary files of the instruments.

But now we know that this will be done block by block. There is a control sequence, `\current@block`, that holds the tokens that will be written to the auxiliary file when the block is done. Each note, translated as `\@add@note`, appends notes to `\current@block`, until the time comes to write it down to the auxiliary file.

```
\def\@add@note#1#2{%
  \open@block
```

```


\ifnot@written
  \edef\current@block{\current@block\space\string\next@note}%
\fi
\edef\current@block{\current@block
  \@value
  \the\this@note
  \the\@atnote
  \the\@@atnote
  \string\add@note{#1#2}
  \rhythm@not}%
\@atnote{ }%
\@@atnote{ }%
\this@note\@plainnote
\not@writtentrue
\advance\current@quantum\quantum@skip\relax
\block@donefalse}%

```

Toward the middle of this function we see the addition of the note's elements: `value`, `additional stuff`, `note`, and `rhythmic notation`. The `additional stuff` is actually made of several items: `\this@note`, `\@atnote`, and `\@@adnote`. This is because some things apply to everything (to notes, to rests, and to barlines), some only to notes and rests, and some only to notes. This will be more clear when we discuss the functions for barlines and rests.

Only at the end we see `\@add@note` dealing with blocks and quantization. After it has appended the note and prepared things for the next one, it adds the value of `\quantum@skip` to `\current@quantum`. (The latter has been set by the kind of note this is: if it is a half note, for example, `\quantum@skip` is 128, double a quarter-note that we already know is 64).

So, imagine that we have a  $\frac{3}{4}$  piece (three quarter-notes per measure).

Imagine it has the following rhythm: . That is, whole-note, two sixteenths, and one eighth:  $128q, 16q, 16q, 32q$ .

`\current@quantum` is initialized to 0. After the first note, it will be 128; after the second, 144; after the third, 160; and after the fourth, 192. That, `TEXmuse` knows, is the end of the measure, because `\block@period` is 192 (this is calculated by `TEXmuse` from the user's command `\meter34`). Let's see what happens at this point, when `TEXmuse` is going to append yet another note.

The first thing it does is `\open@block`, which is defined as follows:

```

\def\open@block{%
  \ifnum\block@period>\current@quantum
  \else\current@quantum0\relax
  \fi
  \ifnum\current@quantum=0\relax
    \end@block
  \fi}

```

For the first time, it is *not* true that `\block@period>\current@quantum`. So, the actual contents of `\open@beam` will be executed (for the first time). It first resets `\current@quantum` to 0. And next, it will find—curiously enough—that `\current@quantum=0`, and therefore will execute `\end@block`. (This is not the same test because the user can *force* the end of a measure—make `\current@quantum 0`—at any point, and this will have the effect of making true the second test even when the first one is not.)

So, `\end@block`:

```

\def\end@block{%
  \ifblock@done\else
    \immediate\write\instrument@file{%
      \current@block\string\end@of@block}%
    \immediate\write\instrument@file\expandafter{%
      \the\this@note\string\@bar@line{\@barline}}%
    \def\current@block{\@gobble}%
    \this@note\@plainnote
    \let\@barline\barline@default
  \fi
  \block@donetrue}

```

`\block@done` has been set to FALSE by the last `\@add@note`, so the contents of this function is actually executed. It is here that the whole line, with the elements of all the notes in the block, is written to the auxiliary file. Then, another line is written to the same file, with a barline, and any additional stuff that might apply to it: `\this@note`. (Remember that `\this@note` would have also been added to a note. This is how a clef—but not other things that cannot apply to barlines, like a tie—is correctly added either to the next note or to the barline if it comes first.)

So, when the first note of a new block is reached, the previous one is closed by `\end@block`. But not only a note can open a new block: a rest can too, and, less obviously, anything that attaches to a coming note (i.e.,

that implies the existence of a note coming soon). A clef, for example, is *not* such a thing, because it can be put behind a barline, not necessarily behind a note. But an accidental must necessarily refer to a note, as must the opening of a beam. So, `\open@block` (that calls for `\end@block` if necessary) is part of the definition of rests, beam-openings, and accidentals (as well as notes). `\@@@rest`, in fact, behaves very similar to `\@add@note`. As for accidentals, all they do is to execute `\open@block` and then add the respective items to additional stuff:

```
\def\#{\open@block
  \this@note\expandafter{\the\this@note\string\@sharp}}%
\def\b{\open@block
  \this@note\expandafter{\the\this@note\string\@flat}}%
\def\n{\open@block
  \this@note\expandafter{\the\this@note\string\@natural}}%
```

On the other hand, `\@open@beam` can actually be the function that adds the `\next@note` (instead of `\@add@note`).<sup>1</sup> The test `\written@false` helps avoiding the duplication of `\next@note`'s that would be implied in the succession of `\@open@beam` and `\@add@note`. Apart from that, `\@open@beam` is straightforward:

```
\def\@open@beam{%
  \open@block
  \edef\current@block{\current@block\space
    \string\next@note\string\open@beam}%
  \def\rhythm@not{\string\add@tobeam}%
  \not@writtenfalse}%
```

Going block by block has a consequence on the quantization too. The matrix needs to have entries for the ends of the blocks, so that the program later can know how long the last note of the block was. If the matrix were directly to 0, the last note would have a negative quantum value. (And it is not enough to assume `\block@period` as the end of the block, because the user can force an end-of-block at any point.) So, the quantization has to be able to insert this entry automatically.

---

<sup>1</sup>Maybe this is a remnant from another time. I can't see now why this function is not implemented as the accidentals, by simply adding `\@open@beamto additional stuff`. I don't dare modifying it now, though. Tests will be run later.



That is done with a function called `\no@note`—an equivalent of `\@note` (which is what adds the 1’s to the matrix) but able to add 0’s. `\no@note` is defined right next to `\@note`, in lines 137–59. (Neither `\@note` nor `\no@note` are quoted here because most of them is a simple but bulky code for the actual manufacture of the matrix.)

Using the matrix that was built in the quantization, `TEXmuse` counts the characters and blocks that make up the music. From that it then calculates how many fonts will be needed and how many characters (and blocks) go in each font. There is a variable `\FontSplit` that sets the maximum number of characters that can be put in the same METAFONT font. This is not quite 256, because the METAFONT part of `TEXmuse` is very demanding and may exceed METAFONT’s capacity.<sup>2</sup> So, based on this number (180 by default), `TEXmuse` decides on where to ‘split’ the music into fonts. This part of the code, fairly clear, is represented by commands `\count@chars`, `\check@plits`, and `\check@@plits`, scattered about the different stages. They are easy to understand.

## A final sample

Below is the first part of Kuhnau’s sonatina. And after that, the input that creates it.




---

<sup>2</sup>I recently had to start thinking of a modification of the METAFONT program that will relieve this problem considerably. But it is a major change, and will take some time. In part, that is why I am not documenting the METAFONT program: it will soon change in unpredictable ways.

(typeset by `TEXmuse`)

```

\newinstrument\righthand
\newinstrument[\bassclef]\lefthand
\begin{texmuse}
\meter44
\righthand{\rangefrom{G4}
5C 4[EC] 5 GG
5C 4[EC] 5 GG+
4[FEDC] [BCBC]
[DCBA] 5 G R
5C 4[EC] 5 GG
5E 4[G+E] 5C 4[EC]
[DBCA] [BGA\#F-]
[GABC] [DE\#FG+]
5AA+A+A+
4[BCDE] [\#FG+A+B+]
5CC+C+C+
4[D\rangefrom{D5}GBD+] [CBAG]
[\#FEGF] [AGFE]
[EDC-B-] [DC-B-A-]
5G-R6R|}
\lefthand{\rangefrom{C3}
5CR6R
5CR6R
5CR6R
G-R 4[GFED]
5CRRR
5C+RR\#F
GCDD-
G-R6R
4[\#FD+AD+] [\#FD+AD+]
5GRRR
4[AD+C+D+] [AD+C+D+]
5BR6R
5C+RCR
DRD-R
4[G-B-DG] 5G-R|}

```

```
\end{texmuse}  
\noindent\music{righthand,lefthand}\quad(typeset by \TeXmuse)
```

## Conclusion

This explains the basic working of  $\text{\TeX}muse$ . There is quite a bit more to it, but everything should be easily understood, by just looking at the code, after this summary explanation. I would like to claim that with this document and the actual examples of the various functions already coded, anything could be implemented by the industrious user. That is probably true of the  $\text{\TeX}$  part of  $\text{\TeX}muse$ . To add another possibility to the users' input, you have to think in terms of the four stages: figure out what the addition means separately for quantization, auxiliary files, writing of METAFONT files, and composition of the text. In the current `texmuse.tex` there is precedents of virtually everything: things that attach to notes, things that group notes, things that apply to this note and the next, things that apply possibly to barlines, etc.

But, of course, any new thing will have to be implemented in METAFONT too—and for that I have provided no guide. As I have mentioned, there is a major change to the METAFONT system coming soon, so a thorough explanation did not seem worthwhile. In any case, the log files of  $\text{\TeX}muse$ 's development, if anyone feels like reading such disorganized streams of consciousness, reveal what goes on in  $\text{\TeX}muse$ 's METAFONT program. In fact, my experience tells me, it is more important, and challenging, to know well enough METAFONT itself. If you master that,  $\text{\TeX}muse$ 's use of METAFONT should be a piece of cake.