WORKING NOTE AC126

Alan Creak
1999 April 13

REGAL AND IMPERIAL

Regal and Imperial are names for different generations of a document formatter on which I have been working from time to time since 1988. In this note they are presented for public view for the first time; it contains a brief description of my reasons for embarking on the development, and a similarly brief (and probably unreliable) history of the work I've done.

I thought I could describe the Regal and Imperial saga as "the things I do for fun while on leave", and certainly that's a good approximation. On looking at my notes, though, I find that, as well as extensive entries for 1988 and 1996 when I was on leave in Reading and York respectively, there's quite a bit of evidence of work done throughout 1989; I think that was mainly to produce a version of Regal which I could use to format some technical reports written during, and as a consequence of, work done at Reading in 1988 (and for our annual family newsletter, which imposed some interesting and significant constraints). In the main, though, the "fun on leave" description is fairly accurate, and I didn't think it justifiable to spend university time on it.

HISTORY.

My first contact with a computer formatting programme was with Runoff on the DEC-10 system (unless it was the Prime) which was used in the Computer Centre somewhere around 1980. Given that all the characters were the same size, it did a surprisingly good job. It used a very simple markup language to indicate margins, indenting, and other such requirements, and worked. A tiny editor on the BBC microcomputer[1] provided similar facilities, and produced similarly good results. I don't remember enough about Runoff to comment further, but a very satisfactory feature of the BBC's editor was its provision of parameters to operators which could either set an absolute value (4) or change the existing value (+3, –2). As an example of the value of this provision, notice that a whole section of text could easily be indented en bloc, retaining the relative formatting structure, by adding a single pair of left-shift and right shift operators in the right places. As I'm fond of indenting as a means of indicating structure (doubtless a consequence of early exposure to Algol), I thought that was good.

The next significant event was my first encounter with MacWrite – and, most notably, with rulers and styles. The idea of storing a set of paragraph attributes as an entity with a name struck me as one of those ideas that's perfectly obvious once someone has told you about it, and which you would have been very pleased to have worked out for yourself. But someone had forgotten to make them cumulative (so you couldn't make a ruler which meant "centre this paragraph within the parameters set by the ruler currently in force", or "move the left boundary 1cm right without changing anything else"), so they wouldn't adapt to make sense of formatting changes as they were required.

Further experience with WYSIWYG formatters convinced me, rather quickly, that a markup system was infinitely preferable. All too often, what I saw wasn't what I got; even when I could manage to achieve the page layout I wanted (usually possible, though not always easily), there were far too many instances of the layout changing when the document was printed. This is one phenomenon which has become worse and worse through the years – as new useless feature piles on new useless feature in Word, any chance of printing what you want seems to diminish. (More recently, with frames and pictures anchored to various things, it has become far more difficult sometimes to achieve the desired layout in the first place.)

The experience did make me think about my requirements, and I worked out a fairly accurate specification of what I wanted. The central point was that I wanted to be relieved of the tedious details of formatting to the extent that they could be worked out automatically by simple and obvious algorithms. I was generally happy with the level of control provided by MacWrite, and similar systems; I don't have much requirement for mathematical typesetting, and the few equations I need can generally be handled with superscripts and subscripts, and I don't use decorative features like illuminated capital letters or wiggly lines of type or other devices invented to make text hard to read. On the other hand, if I use equations I do want the equation numbers (and diagram numbers and table numbers ...) to be assigned systematically, and to adjust themselves if I choose to insert another number, I want to be able to refer to other parts of the document by page number and know it's going to work, I want citations to be handled

automatically and converted into nice reference lists, I want to be able to state that ordinary parentheses and colons and question marks and some other characters should always be spaced from the associated text in certain ways which I like. And so on. All these things are in principle easy to manage, but very tedious to do by hand, so they're just what computers should be doing for me.

Another significant principle is worth mentioning, for it affects my attitude to LaTeX. When I'm preparing a document, I know that I'm preparing a document, and I adjust my language and literary style accordingly. Also, I expect to adjust the presentation of the text accordingly, so I don't mind admitting that I'm dabbling in typesetting, so I don't mind knowing explicitly about line spacing and indenting and skipping spaces and such things. A principle of LaTeX is that you shouldn't have to worry about such details; instead, you should simply be able to identify parts of your text according to their functions – headings at different levels, ordinary text, and so on – and rely on the system to work out the details. I have some sympathy with this notion in principle, but in practice the result, perhaps inevitably, is that LaTeX accepts your specification of structure, then formats the text according to Leslie Lamport's preferences. And his preferences are not mine. In practice, I do the same thing with Regal, and doubtless other people do it with TeX, but I build up *my* preferences as a set of rulers with fairly structural names (title, heading, newitem, miditem, etc.), and use them as required by the document structure.

Cursory inspection of markup systems – which is to say TeX[2] and LaTeX[3] – further convinced me that they were the wrong markup systems; for what I wanted to do they were either unreasonably complicated, or inadequate, respectively. I therefore had to choose between putting up with something not to my liking, or writing my own system. (Another possibility was to search further in the hope of finding a markup system which I could accept, but general opinion seemed to be that it would be a waste of time.) Being more than adequately busy, writing my own system wasn't a serious possibility, so I put up with circumstances. Part of the circumstances was that I had to produce quite a lot of smallish documents fairly quickly (notes, assignments, letters, etc.), and MacWrite – later, Word – would do that well enough, so that's what I did. I often thought of a comment I'd seen attributed to Donald Knuth, I think in an interview reported in some journal : it was to the effect that when he realised that it would be a good idea to write TeX, he took six months off his other activities to study typography and do it. (That's from memory; the details might be wrong, but the tenor is right.) That seemed to be the right way to organise a university, but I didn't belong to that one, so nothing more happened.

- until I went on sabbatical leave in 1988. I still didn't think I could conscientiously take six months off to spend on a formatter which would be unlikely to represent a great leap forward in computing research, but I did have much more control over my time, so started on the formatter as a sort of hobby. I was encouraged by moving from a Macintosh system to a MS-DOS portable machine with no provided formatting tools except yet another markup editor. That's when Regal came into being.

The name, for once, is not primarily an acronym (though I toyed with "Rather Elegantly Generated Attractive Layout"); it is merely that anything so closely concerned with rulers must surely be Regal. Rulers are indeed a major formatting tool in Regal, but it is still primarily a markup language, with rulers used as custom-built markup tools for certain purposes. A few details follow; briefly, paragraph attributes can be defined and stored as rulers which can be invoked at any time. Numeric attribute values can be presented as constants or expressions; attributes defined by expressions can depend on their own previous values and thereby take effect cumulatively.

> *One might ask whether "ruler" is still a sensible name for the construct. After all, it presumably took that name from its appearance as it is displayed on the graphical screen of a word processor. In reply, one might answer that the appearance is immaterial; it is a ruler because it rules the format of the text which follows, and that it is that sense of the word which leads to the name "Regal".*

Despite my taking some care before the event to study the problem and to devise a sound structure for the programme, it turned out to be far more difficult than I'd expected, and the first development dragged on – as I remarked above – into 1989, when I was certainly not on leave, but had in effect committed my reports of my leave activities to Regal form so had to get it into at least a working state before I could print them. After 1989, though, there was a decrease in activity. I still used it occasionally, but for the most part reverted to Word because Regal wasn't really satisfactory and I hadn't time to finish it.

In 1996, I went on leave again. This time, I thought, I'd do the job properly. Drawing on previous experience, I started again, and designed Imperial. The name is chosen for the obvious reason; there is as yet no acronym. (I Must Promptly Effect Regal's Implementation At Last ?) Imperial retains the basic ideas of Regal at the level of formatting paragraphs, but deals with perhaps the greatest source of trouble for Regal – the interaction of page layout and paragraph formatting – in a much more satisfactory way. That is a great improvement, but by the time I'd worked it out there wasn't enough time to write the programme. I guess I reached about half way through by the end of 1996, and have done nothing since.

But it's 1999, and I'm on leave for ever, so I'm having another try. This note is to present a short description of the background, facilities, and structures of Regal, and I hope to enlarge on it in the future development of Imperial.

WHAT ARE THEY ?

Both Regal and Imperial are text formatters, designed to handle a markup language not entirely different in style from LaTeX, but more flexible and not tied to fixed formats. They work from definitions of rulers and page layouts and printer characteristics, and accept marked-up text files for formatting. Markup codes are provided to invoke rulers, and to control other formatting characteristics such as character style and page skips.

So much is common. For the rest, I'll concentrate on describing the facilities of Regal, because that's what I want to get on record and because Imperial is still in a state of flux. To show how Regal is used, a fragment of a marked-up text file, with annotations to explain the functions of the markup notation, is presented in the table on the next page. The text file has been slightly edited to collect some interesting features into a comparatively small compass, but is otherwise almost a real example. The only significant departure from reality is the introduction of empty lines in the interests of aligning the columns of the table; the convention in Regal, as with some other markup systems, is that isolated line ends are ignored, but a pair of adjacent line ends marks the end of a paragraph.

It will be clear that rulers are stacked, with the ruler currently at the top of the stack controlling the formatting. It is not clear from the example (though details given below illustrate the point) that installing a new ruler in general involves computing the new parameters from those in force before – so a ruler which moves the left margin right by 0.5 cm will always do so, until there remains less than 0.5 cm of available space. In fact, therefore, what is stacked is not the ruler identified in the instruction; it is the result of operating on the previous item at the top of the stack with the new ruler. (I have sometimes distinguished the stacked entities by calling them "active rulers".) The use of a ruler stack is perhaps the obvious way to implement the requirements, but I was predisposed towards the notion by having used Burroughs compilers, where setting and popping attributes was the convention, and worked very well. Later experience of implementing a punched-card text editing programme MERGER[4] which followed the same conventions fully confirmed my approval. The ruler attributes control the placing of text lines on the page (margins and line spacing), properties of paragraphs (spaces before and after, first line indenting), layout within the lines (tabulation), typeface, and whether or not text should be formatted.

The markup language deals with local requirements. As well as managing the rulers, its facilities cover local character style (bold, italic), line skips and page skips, file insertion, citations, and variables. All markup symbols are prefaced by the backstroke, \. The examples which appear in the sample text are explained in the comment column.

Text	*Comment*
\rs<paper>\rs<right>seeds:faster.ppr\rp\rs<centre>	Stack ruler (\rs< .. >) "paper" (stays in control throughout); Stack ruler (\rs< .. >) "right"; Text "seeds:faster.ppr" (will be right-justified, because of "right"); Pop (\rp) top ruler (right) from stack; Stack ruler (\rs< .. >) "centre".
MAKING COMMUNICATION FASTER. \rp	This text will be centred. Pop (\rp) top ruler (centre) from stack.
SPEECH. \rs<newitem> Moffat and Jolleff\nc<\{M&J}V. Moffat, N. Jolleff :	Text formatted by "paper". Stack ruler (\rs< .. >) "newitem". Define a citation (\nc< .. >) and an abbreviated name for future use (\{M&J}).
\siSpecial needs of physically handicapped severely speech impaired children when considering a communication aid\ei, in ref. \nc<\{ACA}> page 37.> give the output of	Start italic (\si). End italic (\ei); citations can be nested, and note the use of an abbreviated name (<\{ACA}>).
an "average communication aid" as 1 to 10 words per minute, and compare this with the 100 or so words per minute of normal speech.	
\rr<miditem>	Replace the ruler (\rr< .. >) at the top of the stack (newitem) by "miditem".
Szeto, Allen, and Littrell\nc<A.Y.Z. Szeto, E.J. Allen, M.C. Littrell : "Comparison of speed and accuracy for selected electronic communication devices and input methods", \wiAAC \cb9, 229 (1993).> have compared the speed of different interface devices.	Italics for one word (\wi); bold type for one character (\cb).

THE MARKUP INSTRUCTIONS.

The markup instructions provided are listed in the table below.

Item number			*Meaning*
1	*2*	*3*	
c			Select the following character.
e			End the selected string.
s			Start the selected string.
w			Select the following word.
	b		boldface.
	d		down (subscript).
	i		italic.
	u		up (superscript).
f			Something to do with files.
	i<*filename*>		include.
l			About layout.
	n		new line.
	p		page.
		<*pagecount*>	skip one or more pages.
		e	new page at line end.
		n	new page now.
	s		skip space (vertically).
		e<*howfar*>	skip at line end.
		n<*howfar*>	skip now.
	|		partitions the line.
	*		repeat a character.
		<*repeatcount*>*character*	repeat so many times.
		character	repeat to line end.
n			A note.
	c<*text*>		a citation.
	!<*text*>		define a text variable.
	?<*text*>		use a text variable.
p			About pages.
	n<*pagenumber*>		set the page number.
r			Change the ruler.
	i<*rulername*>		insert the new ruler.
	p		pop the ruler stack.
	r<*rulername*>		replace the current ruler.
	s<*rulername*>		stack the new ruler.
#			Set or read a number.
	!<*string*>		define a numeric variable.
	?<*string*>		use a numeric variable.

An instruction is formed by concatenating items. For example, \lse<1.1> means "insert a vertical gap of 1.1 cm at the end of the current text line".

Words shown in italic style are metalinguistic symbols to be replaced by instances of the object they describe; the rest is literal text, including the brackets < ... >. Brackets are always balanced. < characters are only recognised as brackets when they are used in one of these instructions; > characters are only recognised as brackets if a < character has been recognised as a bracket and not yet matched. If you want to put a > character within a bracketed piece of text, you have to make your own arrangements; in practice, as the material within the brackets is usually fairly precisely defined, this doesn't cause much trouble.

A few of the instructions call for more comment than is implied in the table above or in the earlier example text.

Text styles : The first four entries in column 1 control the type style used with the text elements identified. They are used in conjunction with the four following entries in column 2; thus, \cbX will display the character X in bold type. The markup symbols can be used anywhere, so int\cbense will be displayed as "int**e**nse", and in\siten\eise will be displayed as "in**ten**se". The \s and \e brackets need not be nested; the line

A \sbmixture of \sibold\eb type and italic\ei type

is displayed as

A **mixture of *bold*** *type and italic* type

- or at least it should be. (I did test it, but that was a long time ago.)

A word is identified as the string of characters beginning immediately after the \w? markup symbol and ending at the first space found – so if the symbol is immediately followed by a space, no formatting happens. Another consequence is that the word includes any immediately following punctuation mark, which can be a nuisance, but at least means that a number including commas and a decimal point is regarded as a single word. I chose the convention because it has the desirable property of being clear and simple, but a case can be made out for adding an additional category which is something like "the string of characters beginning immediately after the markup symbol and ending before the first character that is clearly intended to end what is going on".

Inserting files : The instruction \fi<xxx> causes text input to be taken from the file xxx until the end of xxx, at which point the source of input reverts to the file from which the \fi instruction was taken. An inserted file may itself contain \fi instructions.

I had originally intended that this operation should be quite invisible to the formatting routines, which would simply receive the stream of characters from the inserted file in place of the \fi<fff> instruction. Experience demonstrated that, while elegant, this wasn't a good idea in practice. The main reason was that it left the formatting very sensitive to the correctness of the inserted file; in particular, if the pushes and pops of rulers in the inserted file were not precisely balanced, the text sections immediately before and after the original \fi might find themselves governed by different rulers. One could undoubtedly use this property to play clever tricks, and some of them might even be effective in some circumstances, but in general one wishes to know from the beginning how the original file is going to be formatted. Care is therefore taken that the inserted file cannot pop the ruler stack below its level at the \fi<fff> instruction, and also that the stack is cut back to that level if necessary – that is, if the inserted file has left more rulers on the stack. As a corollary of this decision, the \fi<fff> instruction, and the end of the inserted file, are both taken as paragraph breaks. In practice, these constraints have made no difference to the effectiveness of file insertion.

I note that if you really want to do silly things with unbalanced rulers and so on, you can still do it with the variables to be defined shortly. If you take advantage of this facility, you're asking for trouble.

Inserting and replacing rulers : The obvious operations on the ruler stack are provided by \rs<rrr> (stack ruler rrr) and \rp (pop the stack). Two other operations were found to be useful in practice.

Most text has a hierarchic structure : paragraphs come within sections, which in turn come within chapters, and so on. It is common to distinguish the levels by some formatting convention, so it is usual to find a new ruler used at the beginning of each level. Pushing and popping make sense in this context of orderly nesting.

But rulers do not control these abstract levels; they are concerned with the format, not the semantics, of the text, and there are several occasions where a ruler is succeeded by another which depends on it but is in a real sense at the same logical level. This text provides several examples.

Perhaps the most obvious is that the first paragraph after a heading is indented on the left from the boundary of the heading, but the first line is not additionally indented; in subsequent paragraphs under the same heading, the line boundaries of the first paragraph are retained, but an indent for the first line is added. This behaviour could be managed with no special operations, but the result is sometimes – not always – to introduce a level in the ruler stack which does not correspond to the text semantics. Therefore, immediately before the next heading one requires one \rp (pop) operator if the preceding section contained only one paragraph, but two if it contained more than one. This behaviour is surprisingly confusing, and it is easy to make mistakes with such structures when inserting or deleting paragraphs. On recognising that so far as stacking styles is concerned the change from heading to paragraph is significant while the addition of the indented first line isn't, it makes sense to provide the "replace ruler" operator \rr<nnn>, which computes the new paragraph attributes from the previous ruler in just the same way as if the new ruler were to be stacked, but then replaces the old ruler parameters instead of simply stacking the new parameters on top. This reflects the real style stacking in the text rather than imposing a mechanical stacking which does not correspond to the writer's intentions.

Another solution which I considered, but never implemented, would be to provide each ruler with a "next paragraph" attribute, which could be used to specify the ruler to be used for the paragraph following the current one. In principle, judicious choice of next rulers could be used to implement most of the formatting of a document, with intervention needed only to identify steps "down" the stack – new chapters, new main headings, etc. I didn't do it because of the potential confusion of levels mentioned above; how do I know at the end of a (say) chapter how many pops to insert ? Of course, there are answers – some sort of stack marker, with a "pop to marker" operator, for example – but that's a level of complexity for which I wasn't ready.

The reason for introducing the second novelty can be seen from this example. Sometimes it happens that while writing a paragraph one wishes to insert a comment which is to be seen as a parenthesis without interfering with the paragraph formatting.

This might be seen as an odd thing to do – after all, why not use a footnote ? A footnote would not break the text in the way described, so arguably be less intrusive. The answer must be that, in practice, it seems to be a useful way to introduce commentary in some circumstances.

Then after the comment the paragraph should simply continue; it is not a new paragraph, so should not have any special features, such as indenting, in its "first" line. To format such paragraphs, I have provided the "insert ruler" operator, \ri<iii>. This is a true stacking operator, as is appropriate for a new semantic level, differing from the ordinary stacking operator \rs<rrr> only in the action taken when it is popped.

Citations : The \nc<ccc> instruction identifies the text ccc as a citation which is to be listed in full at the end of the document, and linked to the point of appearance of the \nc by one of the customary references links. In practice, only the method based on numerical links numbered in order of appearance (as used in this document) was ever implemented, though one reason for that was perhaps that the method implemented is my preference.

The text within the < and > brackets is stored, with the appropriate reference numbers, until the end of the main text, then formatted as an appendage to the main text. Because of this, all normal formatting machinery works still, so the layout is determined by whatever ruler is in force at the end of the main text. Also, as the brackets are counted and balanced, citations may include further citations, which are handled normally. (That's another good reason for using the order-of-appearance reference numbers.)

Sources with more than one citation are handled by permitting any citation to be labelled. If the first item within the citation text is of the form \{ttt}, the text ttt is stored as the citation's label. Further citations including only this label as the citation text will be given the same reference number.

Variables : Two sorts of variable are provided, one for text and one for numbers. In each case, there are instructions to define the value of a variable and to insert its current value into the text. Both were introduced from need rather than on principle.

The text variables can have values which are any text strings, perhaps including Regal instructions to control what happens. They were originally used to provide a standard format for chapter headings; my standard variable for this purpose is defined as a string including a page skip, ruler invocations for the different lines of the heading, standard text, and references to other text variables (for the name of the chapter) and numeric variables (for the chapter number). No text operators are defined.

Text variables are simply interpreted by taking the Regal input from the variable text instead of the previous source until the text is finished; this automatically provides the recursive properties desired, but does impose certain constraints on the character input system, which I describe further below.

The numeric variables were originally introduced to record page numbers for remote page references. Numeric variables are evaluated by a simple operator-precedence interpreter, which handles standard arithmetic operations, parentheses, and references to other variables, which can include the variable being defined.

Both sorts of variable are undefined at the beginning of a run, with the exception of the page number variable, which is set by Regal as formatting proceeds. The defining instructions are executed when they are encountered in the input stream, and can be used as often as required – so each should be regarded as a redefinition rather than the assignment of an absolute value.

Variable names can be single uppercase or lowercase letters. When I first implemented this part of the system, I thought that would be enough; I was wrong.

Multiple passes : To use a numeric variable for a remote page reference, one must assign its value from the page number variable somewhere on the required page, then use the value where the reference is required. This is (reasonably) straightforward if the reference is required after the object of the reference, but not so straightforward if the positions are reversed.

There are two reasons for the difficulty. The major reason is that, with a forward reference, we don't know the number to insert at the point where it must be inserted. The minor reason is that we don't even know how much space the number will occupy, so we can't do the formatting anyway. It is significant that there are two reasons, because they could in principle interact in an awkward way.

In the first pass through the text, Regal will discover that the required number is undefined when the reference is made; it is therefore obvious that the formatting is unsatisfactory when we come to the end of the run. To deal with the minor problem, it will guess a plausible size for the number so that it can go on with its immediate activities, but this might not be right. Regal will also notice when each variable is defined. If no variable has been defined in the run, the forward reference will never be resolved, so formatting fails, and the process stops. All being well, though, the number has been defined, so Regal can continue. (In practice, it only checks that *some* variable has been defined with a changed value; keeping track of just which is required and which has been defined didn't seem worth while.)

Regal therefore knows when a second pass through the text is needed, and starts again from the beginning if necessary – without resetting the values of the variables. (This sounds like an accident waiting to happen; perhaps you could forget to set the initial value of, say, a chapter number, in which case it would restart from its previous final value and continue to count upwards on the second pass. In fact, that can't happen, because unless it is explicitly set to some value in the first pass it will always remain undefined.) This time the forward reference can be resolved, and correctly formatted. It is now possible, though, that the correct formatting differs from the previous guess sufficiently to change the page numbers later in the document, so the correct reference could be different from that defined in the first pass. In this case, another pass will be needed – so Regal must check whether the value at the end of the pass has been changed, and repeat if necessary.

It is also possible that a chain of forward references could take several passes to resolve completely, so Regal keeps going provided that at least one variable has been newly defined in each pass.

The process stops either when all seems to be well, or when two passes have been made with no improvement – that is, changes in value but no new definitions. This can happen if there are circular definitions – or (this is the awkward interaction) if a formatting change affects a reference in such a way as to undo the change on the next pass. (A reference to a page number might change from 9 to 10, the additional character might then affect the formatting in such a way that the point of reference moves in such a way that the reference in the next pass changes back from 10 to 9. Recall that we can do arithmetic on the values.)

A NOTE ON INPUT STREAM MANAGEMENT.

It might be clear from the descriptions above that input to the formatting procedures can be acquired from several different sources. These are :

Files : Input is initially taken from a primary file, which in the simplest case can be the only source. For any but a very small document, though, it is often convenient to split the source text into several files each corresponding to one part of the material and to use the initial file simply as a coordinating device, including the other files as required. As file inclusion is in principle recursive several files can be active at the same time.

Variables : A text variable is included in the document by simply taking input from the declared variable string instead of from the current character source. A numeric variable is converted into text by a procedure, and the output from the procedure used as input to the formatter. In the Regal implementation, the output is first written to a character buffer, which is then used as the input source. Variables are also recursive, so several variables can be active at the same time. There is nothing in principle to prevent a text variable from including a file. I don't think I've ever tried it, but it should work.

References : As citations are handled by the formatter a string of text for the reference list is constructed. This is appended to the input text after the initial file is ended.

Originally, I underestimated the potential complexity of this stream-management problem, and matters became unsatisfactorily complicated, with many fairly arbitrary variables used to hold various pieces of the system state and rather little in the way of orderly structure. Different variables were introduced to record essentially the same information at different steps in the process, and their values were not always well coordinated; much confusion ensued.

The situation was greatly improved when I changed the input system to use my Stream utilities[5], with which all these input requirements could be handled simply and uniformly.

DEFINING RULERS : AN EXAMPLE.

Rulers to be used in a formatting run are defined in a ruler definition file. Here is an example of such a file; it is sufficient to format the text file example given above. The right-hand column is not part of the file.

% PAPER.RLR.	Comment
% Use "paper" first, then stack others as required.	Comment
#n paper	Define a ruler called "paper".
=lb \c 15	Line boundaries : 15 cm line, centred.
=le i	Single end-of-line symbols ignored.
=lj b	Lines are justified at both ends.
=pi 0	Paragraph indent is 0.
%%%	End of definition.
#n newitem	Define a ruler called "newitem".
=lb @+1 @	Move the line left boundary 1 cm right; leave the right boundary unchanged. (Arithmetic expressions can always be used for numeric values; @ in an expression always means "the current value".)
=pi 0	Paragraph indent is 0.
%%%	End of definition.
#n miditem	Define a ruler called "miditem".
=pi 1.5	Paragraph indent is 1.5 cm.
%%%	End of definition.
#n centre	Define a ruler called "centre".
=lj c	Centre the lines.
=pi 0	No indenting.
%%%	End of definition.
#n right	Define a ruler called "right".
=lj r	Right-justify the lines.
%%%	End of definition.

The example shows that properties of several sorts can be defined for a ruler. It also shows that not all properties need be defined for every ruler. That is because (as explained above) a ruler specifies the *operations* which must be applied to the parameters, not the parameters themselves. The complete set of paragraph properties (the "active ruler") is computed by Regal when the ruler is invoked.

DEFINING THE RULERS : NOTATION.

It will be clear from that example that the language used in constructing the file is not specially self-explanatory. A fairly complete specification of the definitions which can be expressed in the language follows. It must be admitted that there is an element of reverse engineering in some of the details, corresponding to a similar element of vagueness in (or in some cases absence from) the existing documentation. Still, the true ambience of Regal is preserved throughout.

Character position			Parameters	Meaning
1	2	3		
%				Special.
	%		*none*	End of this picture.
	not%		*none*	A comment line
#				Start a new definition.
	d		*none*	A device parameter.
	n		*none*	A ruler's name.
	m		*none*	A page map.
=				Item within a definition.
	c			Things about characters.
		e	string of instances of "cpqrs" – c is a character; p is "~" (at least) or absent; q is spaces before c; r is "~" (at least) or absent; s is spaces after c.	Character environment; defines characters which must have spaces left before and after.
	l			Things about lines.
		b	\c n where n is length of centred line; or n m where n and m are positions of left and right boundaries.	Line boundaries.
		e	i for ignore, or o for observe.	End of input line is significant ?
		j	l for left, or r for right, or c for centre, or b for both.	Justifying.
		s	line spacing required.	Separation.
		t	\N if values are in ens; * n for every n units; list of numbers, each identifying one tabulation stop.	Tabulating.
	p			Things about paragraphs.
		b	how many blank lines to leave before the paragraph text.	Lines before paragraphs.
		c	how many columns on the page.	Columns of text.
		g	width of gap between columns.	Gap between columns.
		i	magnitude (possibly negative) of first line's indenting.	Paragraph indenting.
		m	*none*	Paragraph is monolithic.
	s			Things about spaces.
		f	factor for scaling vertical skips when using nonstandard line spacing.	Factor for \ls.
	u			Things universally effective.
		f	yes or no	Formatting.
		t	name of typeface	Typeface.

The entries are for the most part adequately self-explanatory, but those corresponding to initial character # deserve comment. In the development of Regal, the ruler file came first, then when additional

specifications for page layout and device information were required I added them to the ruler file. In hindsight, that was silly, as the different items of information are only remotely connected, and their treatment in Regal happened in quite different parts of the programme. In practice, it didn't matter much because hardly any information was given in the page and device parts, so I haven't described these any further. (While page and device information was used, it was almost all written into the programme to match my printer, and I never got round to changing it until Imperial was under development; I say a little more about this topic below.)

The two entries (=pc and =pg) which control presentation in columns are more of a curiosity than a reality, though they do represent good intentions. I had intended that a section to be presented in (say) two columns would be introduced by a ruler which was quite likely to contain only the instruction =pc2, and left in the usual way by popping. I spent quite some time on the implementation, and it worked quite well – provided that you were happy to have each paragraph formatted separately into two columns. I wasn't; the effect was bizarre. I never had time (nor inclination) to work out how to get round the strongly paragraph-based structure of Regal to achieve the required effect. The question is sidestepped in Imperial by moving the page format into a separate category, independent of the text formatting; this works much better.

WHAT'S IN THE RULER.

The active ruler is implemented as a C structure, and its declaration appears below.

```
typedef struct                /* Active ruler components.          */
{
char  * arname;               /* The ruler name.                   */
nots  * archarwidth;          /* Current character widths.         */
nots  arennots;               /* Size of an en.                    */
int   arfont;                 /* Number of the font.               */
log   arformat;               /* Formatting.                       */
int   arlinel, arliner;       /* Line boundaries.                  */
log   arsameline;             /* Preserve end-of-line.             */
int   artabs[ artabmax ];     /* Tabulation positions.             */
int   arntabs;                /* How many tabulation stops.        */
int   arlinedy;               /* Line spacing in nots.             */
float arskipfactor;           /* Factor for vertical skips.        */
int   arparbetween;           /* Lines between paragraphs.         */
int   arparindent;            /* Paragraph first line column.      */
int   arcolumns;              /* How many columns in the page.     */
int   arcolumngap;            /* Nots between the columns.         */
int   arjustif;               /* Justification code.               */
log   arsplit;                /* An inserted paragraph.            */
log   armonolithic;           /* Don't split between pages.        */
char  arspecials[ 128 ];      /* Special characters.               */
char  arspbefore[ 128 ];      /* How many spaces before.           */
char  arspafter[ 128 ];       /* How many spaces after.            */
} ACTIVERULER;
```

"**nots**" are "notional dots". (Compare "bits" for "binary digits".) I worried a lot about how to measure positions of elements of the formatted material on the page. While it is sensible to require that all dimensions supplied to the system are in standard units (centimetres), I was using a dot-matrix character printer, in which the real units were either the distances between dots of the matrix or current character sizes, and I had a lot of trouble with rounding errors shifting material either one dot or one character out of place. It was therefore sensible to express some of the measurements in terms which reflected the discreteness of the printed representation, and nots (with sizes defined in the printer definition) were used for this purpose.

Fonts are identified by number; they are defined in the device definition, discussed below. They are always associated with rulers because all my attempts to make my printers change font within a line failed.

I think that many of the other definitions have little but historical significance now, and there's not much point in describing them all.

PAGE LAYOUT.

Regal's facilities for defining page layout are primitive, but useful. That's because I only implemented the bits I really wanted to use; plans for future development were always future plans for development. They are strongly flavoured with the dot matrix printers at my disposal, all of which could print only a few fonts in each of which all characters were of the same width (though the widths were different for the different fonts), and it is quite possible that the many oddities of these devices distracted me from a more orderly approach to the problems.

The Regal view of page layout is based on a model in which text is placed within a frame, which in turn can be positioned on a page. I have unwisely not been careful to preserve the distinction between pages and frames, perhaps because of the pressing problem of working out just where text should appear on a page in which both structures are important, and the result is something of a mess. Here, though, is the structure of the page map for a Regal file. It is clear that the information contained is derived from both frame and page specifications.

```
typedef struct              /* Page maps.                    */
{
float ampagelength;         /* Centimetres down a page.      */
float ampagewidth;          /* Centimetres across a page.    */
float amframetop;           /* Space in page above frame.    */
float amframebottom;        /* Space in page below frame.    */
float amframeleft;          /* Margin at left of frame.      */
float amframeright;         /* Margin at right of frame.     */
float amtexttop;            /* Space in frame above text.    */
float amtextbottom;         /* Space in frame below text.    */
int   amlinesperpage;       /* Total lines in a page.        */
int   ampagelnlength;       /* Total nots in a line.         */
log   amp1header;           /* Put header on first page.     */
char  * amheader;           /* Points to the header.         */
} ACTIVEMAP;
```

A single map is used for the whole file, which is why it contains a flag controlling the appearance or otherwise of the header on the first page. It became clear before very long that this was quite inadequate, and some sort of hierarchic structure was needed in which a standard pattern, corresponding roughly to this page map, could be amended as required for different pages, but this extension was never implemented.

Attributes of the map can be defined in the ruler definition file, after the #m header. The definitions available are tabulated :

Character position		*Parameters*	*Meaning*
1	*2*		
=			Item within a definition.
	h	string – typically a variable.	The page header.
	l	frame length, cm.	Length of the page frame.
	t	position of top line, cm.	The top of text in the frame.
	w	frame width, cm.	Width of the page frame.
	1	yes or no.	Print a header on page 1.

It is interesting that, without specific planning, these definitions do separate the frame layout from the page size imposed by the printer.

One more peculiarity of the interaction between printer size and frame layout in Regal deserves mention, for it is something which I haven't come across anywhere else. (That isn't an assertion that my

implementation is unique – only that my familiarity with other approaches to such problems is limited.) It is the ability to print frames larger than the maximum page size.

This is where the family newsletter comes in. The newsletter is a fairly considerable document, typically around twenty pages in length and distributed to around fifty people. With a production of that size, the amount of paper consumed is considerable, and for environmental and photocopying cost reasons I would like to reduce it as far as is reasonably possible. With a modern printer I can simply use a smaller font; what I had was a dot-matrix printer, with essentially a fixed character size. That in itself is not a serious problem, as I can reduce the size of the original printed document by photocopying so that more fits on a page. The obvious simple method (assuming that one can print on continuous stationery) is to print the text continuously on the printer, then to slice it into segments of suitable length and reduce those as required. To do so with Regal, one simply declares the page frame longer than the notional printer page length; as I never used page skips – because of constant confusion between A4 and quarto pages – that was straightforward.

Unfortunately, it is only half satisfactory, as without further changes the printed width remains the same, so the reduction economises in only one dimension, and blatantly wastes space in unnecessarily large margins. If reduction by a factor of two is feasible, one could try pasting up the reduced version with two columns per page. Page numbers would then become column numbers (odd, but comprehensible) – but in practice the reduction is too extreme, and the resulting text is far from easy to read.

I therefore spent quite a lot of time and ingenuity causing Regal to handle over-sized page frames by printing in such a way that each frame was produced as two (or more, if required) parallel columns (with some overlap for convenience) which could be cut out and stuck together to make a large page ready for reducing. It worked surprisingly well; occasional unevenness in line spacing caused by somewhat sloppy tractor action in the printer led to occasional little jumps in the lines, but they were tolerable, and otherwise all was well.

DEVICE DEFINITION.

Proper device definitions were never used in Regal, but there was some preparation for them. Most of this was in the structure reproduced below; the comment on the second line is almost true.

```
typedef struct          /* Describes the output device.           */
{    /* ***************** NOT YET USED *****************           */
float dvlinecms;        /* Line length, centimetres.              */
float dvlfcms;          /* Line advance, centimetres.             */
int   dvlinenots;       /* Line length, nots.                     */
int   dvlfnots;         /* Line advance, nots.                    */
int   dvdeltaymax;      /* Maximum ( y ) skip, nots.              */
log   dvfixedwidth;     /* All characters are the same width.     */
} DEVICEVALUES;
```

The items included are simply a selection of those which had turned out to be significant in the development of Regal. I don't remember how I chose the selection; certainly, there is much more in Regal that is specific to the printer I most commonly used, such as the control strings to be used for forward skips of various sizes, changing type styles (italic, bold, etc.), and so on.

There is a little more in the font definition, shown below. This was used to switch between different fonts available on the printer, and it works. Though it was not used with the printer, as the character width was fixed for each font, there is provision for the definition of individual character widths. The items `tfstart` and `tfstop` are, respectively, control strings to be sent to the printer to change the font and return the printer to a standard state.

```
typedef struct                      /* Describes the font.        */
{
char tfname[ rulernamelength ];     /* Name of the font.          */
int tfennots;                       /* "Standard" width, nots.    */
nots tfcharwidth[ 128 ];            /* Character width, nots.     */
char tfstart[ tfcodemax ];          /* Codes to start font.       */
```

```
char tfstop[ tfcodemax ];          /* Codes to stop font.         */
} FONT;
```

IMPERIAL AND THE FUTURE.

That is a rough account of the state of Regal in its most developed form. It attained this optimum state around the middle of 1989, while undergoing development for use in formatting some technical reports (including that on the Stream utilities). Unwisely, in my concern to get the reports completed, I departed from my policy up to that time and took some short cuts in the development; it worked for the reports, but Regal was never quite the same again. After 1989 I didn't have time to go back to Regal, undo the hasty additions, and do the job properly, so it remained in its slightly crippled state. The basic formatting machinery was unharmed, and it was used for our family newsletters until December 1993; in 1994, my Zenith portable computer, which was the machine on which Regal had been developed, broke down, and I did nothing with Regal for some years.

As I mentioned earlier, I returned to Regal in 1996, but with the benefit of my previous practical experience, and – most important – the careful notes I'd made in 1988 and 1989. From these sources, I was able to recapture my ideas about Regal and also to identify aspects of the original system which could be significantly improved. After this review, I decided that it would be best to start again rather than to try to rescue Regal, so Imperial was born. Full specifications will appear if I ever get it finished, and in any case this note is really about Regal, but here is a short list of some of the changes. Notice that none of these is about the basic formatting; Regal did that pretty well on the whole, and the developments are mostly concerned with improving other parts of the system and – particularly – with decoupling them as far as possible from the formatting.

- Different parts of the document – main text, headers, footers, pictures, references, etc. – are regarded as separate streams and formatted independently. This solves the significant problems associated with such things as preserving continuity of formatting over page boundaries, when in Regal a complicated transition to a different form to manage the page header proved to be the source of many problems.

- The page map is much more flexible. Each page has a separate map (so that pictures or different numbers of columns or whatever can be laid out properly) in which an arbitrary number of panels can be defined. Each panel is linked to one of the document's streams, and much of the formatting of a particular stream can be done independently of the others. In practice, certain interactions are necessary, and provision is made for these.

- The output is no longer tied to a specific printer. Printer declarations receive proper attention, and in principle any sort of printer or printing technique can be used for the target; the original intention was to use TeX to finish off the process, but other forms of display such as postscript, screen displays, etc. are possibilities.

MORAL.

Now I know :

- why TeX is so complicated – you do want to manage a surprisingly large number of things, and exercising all the management and text input through a single text channel forces the use of complicated codes;

- why Word can't get its formatting right – even slight changes in page layout or pixel size can change the formatting details;

- why Word got worse and worse – the interactions are such that merely addressing symptoms is almost guaranteed to make the problem worse;

- why both accrete more and more bits and pieces as they are made more elaborate – you do need a lot of information.

- but not why Word is so enormous.

REFERENCES.

1 : Acorn Computers Ltd. : *The BBC Microcomputer System Master Series Reference Manual, Part 2* (1986).

2 : D.E. Knuth : *TEX and METAFONT : new directions in typesetting* (Digital Press, 1979).

3 : L. Lamport : *LATEX : a document preparation system* (Addison-Wesley, 1986).

4 : G.A. Creak : *U/MERGER* (Auckland University Computer Centre Usernotes 209, 209A, and 209B, 1976).

5 : G.A. Creak : *The Stream Utilities Manual* (Auckland University Computer Science Department Technical Report #42, 1990).