8. A Tour Through a NeWS Application

"All craftsmen share a knowledge. They have held
Reality down fluttering to a bench"

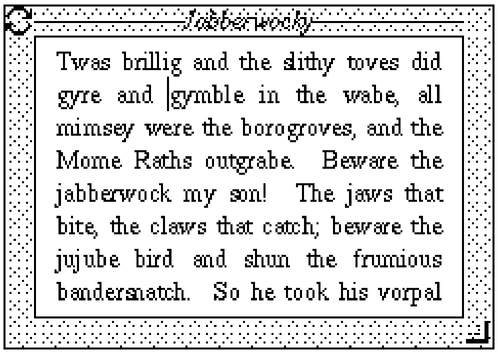
Victoria Sackville-West

This chapter reviews a relatively large NeWS application and explains some of the ways that it uses NeWS to advantage. The application is *ched*, a *ch*eap *ed*itor built as a demonstration of how to build a WYSIWYG editor in NeWS. The source for this application is in the public domain and available from Sun Microsystems. The last section in this chapter explains how to obtain the *ched* program.

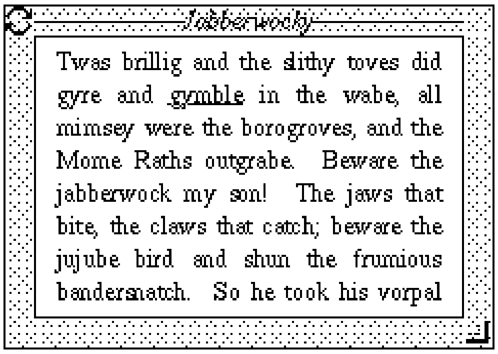
Ched is a fairly simple WYSIWYG (what you see is what you get) editor. It implements automatic line breaking, left- and right- margin justification, and selections using the mouse. In Figure 8.1 is a set of snapshots of *ched* being run, along with a commentary on what is happening at each step.

This chapter begins with a section on the general structure of *ched*. This segment is followed by a series of sections on various key points in the implementation of *ched* that are relevant to its use of NeWS. The points to be covered are:

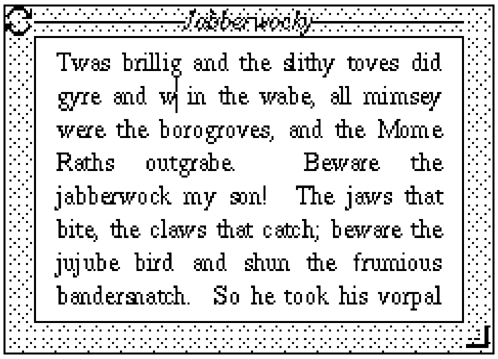
- Fixed point arithmetic.
- How *ched* uses NeWS to display the document.
- Font information.
- Use of *usertokens*.
- Debug initialization.
- Input events with names.



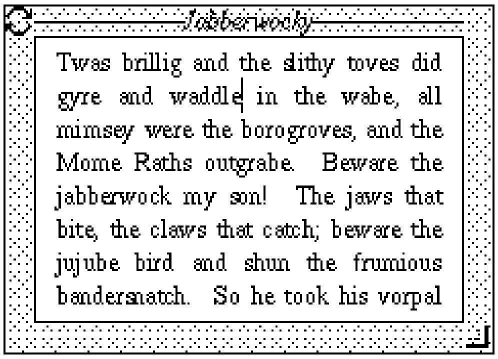
When *ched* is started it creates a window and displays the document in it. It automatically breaks lines between words to put as many words on a line as possible, without extending beyond the right margin. *Ched* enlarges the spaces between words in order to align the right edge of each line.



The current position, or *dot* as it is referred to in some other editors, has a width in addition to a position. If the width is zero, then it is displayed as a vertical line, as in the previous screen. In this screen, an entire word, “gymble” has been selected. This was done by pressing down on the left mouse button between the space and the “g”, and sweeping the mouse right.



After the word “gymble” was selected, a single “w” was typed. This caused “gymble” to be deleted and “w” inserted in its place. Notice that the document has been reformatted so that the right edges remain aligned and there are as many words as possible on each line.



The rest of the word “waddle” is typed and, as it is typed, the document is continuously reformatted to look as it would if it were printed. *Ched* is engaged in a dialogue with NeWS, continuously receiving characters, updating internal data structures, and sending descriptions to NeWS of the reformatted image.

Figure 8.1 An example showing ched in action.

- Responding to damage.

- Selections.

- Typeahead.

Ched uses a number of real-time layout algorithms, which will not be described in detail in this chapter. Interested developers should obtain the *Ched* application in order to get more information from the source code.

8.1 General Structure

Ched follows the Smalltalk model/view/controller paradigm that is used in Smalltalk. There are two key data structures, and three key segments of the program. The first data structure is the *document*. It represents the actual text of the document being edited. It has a number of subsidiary data structures that contain the characters in the document, the formatting information, a set of markers that indicate portions of the document, and a list of the views on the document. There are a set of routines that deal with these structures. These structures and procedures form the *model* portion of *ched*. The next data structure is the *view*. It contains the actual visible representation of the document. It is what maps the document onto the screen, and hence is the part of *ched* most closely connected to NeWS. This, and its subsidiary data structures form the *view* component of the paradigm. These two are glued together by the *controller* that provides the user interface. The Figure 8.2 shows how these parts relate to each other and to NeWS.

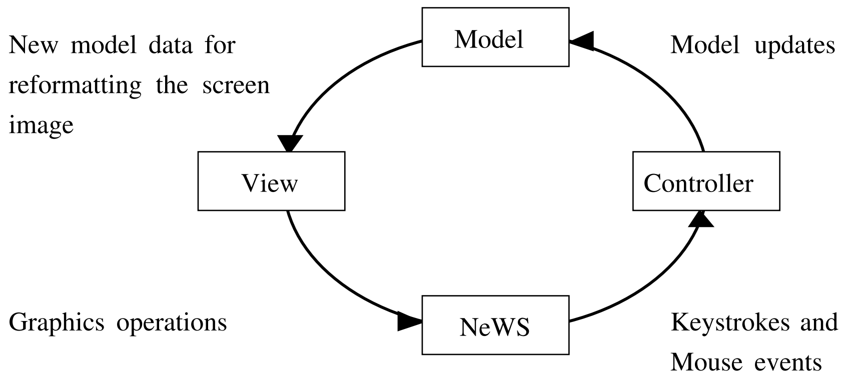


Figure 8.2 The flow of information amongst the various parts of *ched*

A piece of PostScript code that was downloaded by *ched* to NeWS reads keystrokes and mouse events and sends these off to the controller portion of *ched*. The controller decides what to do with the character and updates the model in response. When the model is updated, information is fed back to the view to tell it that something has changed. Periodically, the view uses the updated information from the model to send PostScript graphics operators to NeWS that draw the new visible representation of the altered document.

8.2 Fixed Point Arithmetic.

Internally, *ched* does all of its computations in units of points (approximately 1/72 of an inch). Besides being a generally convenient unit for use in documents, it is also the default unit in PostScript. But points are not sufficiently accurate for general use: a finer precision is needed. *Ched* uses fixed point numbers, with 16 bits of fractional precision, and 16 bits of integer precision, including sign. This fits in very well with most modern computer architectures that have standardized on 32 bit words. Here are the declarations necessary for dealing with these fixed point numbers:

```
typedef long fixed;         /*32 bit fixed point number with
                            16 bits of fraction */
#define fixedi(i) (fixed)((i)*(1<<16))
                            /*convert int to fixed */
#define floorfr(fr) ((fr)>>16)
                            /*convert fixed to int by flooring
                            */
#define floatfr(f) ((double)f/(1<<16))
                            /* convert fixed to float */
#define FIXED_HUGE 0x7FFFFFFF
                            /*the largest possible fixed point
                            value */
```

These definitions provide the basic tools for declaring fixed point numbers and converting between them, integers, and floating point numbers. One fixed point number may be added to another by using standard integer addition, and one can be multiplied or divided by an integer by using the integer operation. Unfortunately, multiplying or dividing one fixed point number by another is more complicated, but *ched* never needs these operations.

A very significant advantage to using this form of fixed point number is that *cps* has special facilities for them. *Cps* understands the `fixed` declaration and can efficiently ship numbers to and from the server in this format. For example, the following declaration appears in ched.cps.

```
cdef ps_frmoveto(fixed x, fixed y) x y moveto
```

Then the call “**`ps_frmoveto(fixedi(1)/2,fixedi(1)/2`**)” generates the PostScript fragment “**`0.5 0.5 moveto`**”. This is especially efficient since the NeWS server internally uses fixed point numbers wherever possible, to avoid floating point.

8.3 Displaying the Document.

In PostScript, as in all other languages, the execution speed of a program is very closely related to how well it is written. Therefore *ched* pays quite a bit of attention to writing clean code to send to NeWS. Developers who write applications that generate PostScript, either for a printer or NeWS, are often seduced by PostScript's programmability. In particular, there is a tendency to download procedure definitions that define a graphics model that is more familiar than the graphics model of the PostScript language. This extensive downloading of code can incur extra overhead in the NeWS server, where operations are more expensive than they are in the application. The PostScript language is, after all, an interpreted language. If, instead of substituting another model, the PostScript language graphics model is used directly for the application, the performance benefits are large. Once again, this rule holds for both NeWS and PostScript printers.

C*hed* uses the basic PostScript operators wherever possible, and it attempts to generate the most efficient code possible. For displaying documents, *ched* actually only generates five PostScript fragments:

x y **moveto**	Sets the current position to *x,y*.
f **setfont**	Sets the current font to *f*.
s **show**	Draws string *s*.
x 0 32 s **widthshow**	Draws string *s* with *x* added to the width of every space. Widthshow is used on lines that are right justified, show is used on those that are not.
w h x y **ERS**	Erase a rectangle starting at *x,y* whose size is *w,h*. This is the only PostScript procedure call ever generated during document display.

With these five operators, *ched* is able to do everything it needs in displaying a document. It is very careful to only invoke **setfont** if the font is really changing. It only invokes **ERS** if the rectangle really needs erasing. And it attempts to draw the longest strings possible. One easy way to sacrifice a lot of performance is to use **show** for each word, and to call **moveto** to reposition at the beginning of each word. When there is a sequence of words with equal spacing between them, it is far more efficient to call **moveto** once, and **widthshow** once. *Ched* takes advantage of the fact that it is using fixed point numbers: it can specify a width to **widthshow** that is a fractional number of pixels. NeWS will use this and distribute the error amongst the spaces between the words such that some of the spaces are one pixel wider than others. Because of this, a line of right justified text with no font or baseline changes is drawn using exactly one **moveto** and exactly one **widthshow**.

8.4 Font Information.

Ched frequently needs to measure the width of strings of text. It could do this by sending the strings over to the server and asking the server to measure them, but that would incur massive message transmission penalties. Instead, it asks the server for a table of widths for each font that it's using, and uses that table locally.

First, we look at the straightforward C language declaration of the font structure used by *ched*. It contains the name of the font (e.g. "*Times-Roman*"), its point size (*line 12*), the height of the bounding box of all the characters, the depth of descenders from the baseline, the number of characters, an "index" for the font, and the actual array of widths. The *index* of the font is a NeWS identifier that refers to the PostScript font. The overall font structure follows.

```
struct font {
 struct font *next;        /* Fonts are in a linked list */
 char *name;               /*Family name */
 char size;                /*size in points */
 char bbheight;            /*Height from highest ascender to
                           lowest descender */
 char descent;             /*distance from lowest descender
                           to baseline */
 unsigned short nchars,    /*The number of characters in the
                           font */
 fontindex;                /*The magic token by which the
                           server knows this font */
 fixed width[256];         /*The array of widths */
};
```

This structure is used by the C function *ft_create* that takes a font name and size and returns an associated font structure. *ft_create* first checks if that font is already known, and if so, returns the known font structure. If the font is unknown, then *ft_create* sends a message to the NeWS procedure (*ps_defstr*) that requests information about the font. Once *ft_create* has got the basic information, it allocates a font structure and fills it in. *Ps_defstr* follows its return results by a stream of widths. This stream is picked up one-by-one and stuffed into the array using *ps_getint*.

```
struct font *
ft_create(name, size)
 char *name;
{
 register struct font *f;
 register index;
```

```
int length,
bbheight,
descent;
for (f = fontroot; f; f = f->next)
  if (f->size == size && f->name[0] == name[0]
            && f->name[1] == name[1]
      && strcmp(name, f->name) == 0)
    return f;
index = ps_next_user_token++;
ps_defstr(name, size, index, &length, &bbheight,
            &descent);
f = (struct font *) malloc(sizeof(struct font) +
  (length >> 1) * sizeof f->width[0] + strlen(name));
f->next = fontroot;
fontroot = f;
f->size = size;
f->fontindex = index;
f->nchars = length >> 1;
f->name = (char *) &f->width[f->nchars];
f->bbheight = bbheight;
f->descent = descent < 0 ? -descent : descent;
strcpy(f->name, name);
for (index = f->nchars; --index >= 0;)
  ps_getint(&f->width[index]);
return f;}
```

Ps_defstr looks up the font and scales it, and passes the scaled font and the font index, to a PostScript function called *DFS*. *DFS* returns the height, descent, and number of characters in the font.

```
cdef ps_defstr(string name, size, index,
       length, bbheight, descent)
    => DEFSTR_TAG (bbheight, descent, length)
       name findfont size scalefont dup index DFS
```

When *DFS* is called, the index is on the top of the stack, and two copies of the font lie below the index. *DFS* first executes **setfileinputtoken** which defines the font as a user defined token, making use of the index. (User defined tokens and how they work will be discussed in the folowing section). It then proceeds to send back the information needed by *ft_create*.

```
/DFS {
 setfileinputtoken
 DEFSTR_TAG tagprint
 begin
   currentdict dup fontheight typedprint
   fontdescent typedprint
   WidthArray dup length typedprint
   aload length 2 div { pop typedprint } repeat
 end
} def
```

ps_getint is the last important function which is used by *ft_create*. It simply receives a fixed point number from the server. It has only a return result, with no body of PostScript code to be sent to NeWS and no tag to wait for in return.

```
cdef ps_getint(fixed x) => (x)
```

8.5 Use of *user tokens*.

In the previous section on fonts, there was a small piece of magic left unexplained that involved the **setfileinputtoken** NeWS primitive. This primitive makes use of the NeWS user defined token facility. Chapter 7 discussed tokens briefly, and how CPS and Compressed PostScript make use of tokens in order to compress data for transmission. NeWS has a mechanism, supported by *cps*, where a client program and the server can cooperatively agree on the definition of a user token. This allows for efficient protocol definition for a specific application. The NeWS reference manual gives more detail about tokens and their definition, but we discuss them briefly here.

There is an array of PostScript objects associated with each input stream. The NeWS protocol and *cps* have facilities that allow references to these objects to be efficiently encoded in the stream that flows from the application to NeWS. These references are represented by indexes into this array. **setfileinputtoken** puts a PostScript object, in this case a font, into this array. *cps* encodes a reference to this object when the parameter type *token* is used:

```
cdef ps_do_usefont(token font) font setfont
```

This is called by the C program passing the same index to *ps_do_usefont* as was passed to **setfileinputtoken**. This is the index that is in the font structure. The global variable *ps_next_user_token*, which is referenced in *ft_create*, simply keeps track of the next available slot in the array.

8.6 Debug Initialization.

Almost all of the interface-code fragments defined in the *ched cps* specification file are quite simple. One, however, is not. It is the piece that initializes the environment in the server. It contains the definitions of procedures that create windows, define fonts, handle input events, and an assortment of other things. These definitions are is kept in a file that is separate from the *cps* specification file. In the specification file, the following lines appear:

```
cdef ps_startup()
#ifdef DEBUG
    (/home/norquay/jag/ched/ched.ps) run
#else
#include "ched.ps"
#endif
```

The initialization file is dynamically loaded from the file *ched.ps* when *ched* is being debugged. Once *ched* is stable and not being debugged, the initialization code is statically compiled into *ched*. This has the advantage when debugging that ched.ps can be altered, and the new version tested, without recompiling *ched*. Once the debugging cycle is finished, there is one less file to be bundled into the release and potentially installed incorrectly.

8.7 Input Events With Names.

The input loop in *ched* is as follows:

```
{ clear awaitevent begin
  ClientCanvas setcanvas
  { Name type /integertype eq
     { 2 tagprint WindowID typedprint Name typedprint }
     { KeyActions Name get cvx exec } ifelse
  } stopped
  end
} loop
```

Whenever an event is received, its **/Name** field is inspected. If it contains an integer, then it is just a simple keystroke, and a message containing the integer is sent to the client. Otherwise, the name is looked up in the **KeyActions** dictionary and whatever value is found will be executed. The **KeyActions** dictionary contains a set of procedures whose names match the names of special events generated by NeWS. The *stopped* and *clear* values in the input loop above are used here as guards and cleanup in the case of missing *KeyActions*.

For instance **/LeftMouseButton** is defined as a procedure that sends a message to the application saying that the left mouse button has gone up or down, depending on the **/Action** field. Besides sending the message to the application, the procedure either enables or disables the catching of **/MouseDragged** events, depending on whether the button is going down or going up. This sequence avoids the overhead of dealing with drag events when no button is pressed. Also, since this enabling happens in the NeWS server, and with the scheduling guarantees given by NeWS, the downstroke and enable happen *atomically*: there is no chance that an up event can happen in the meantime, which might be missed and cause the system to get stuck receiving drag events.

8.8 Responding to Damage.

In the previous examples given in this book, the execution of most graphics operations has been triggered by the invocation of the */PaintClient* method in the window. The typical simple application defines this method as a procedure that draws the entire contents of the window. When the window gets damaged, a **/Damaged** event generated. This event is caught by a handler that invokes the */FixFrame* method. */FixFrame* sets the clipping to the damaged region, repaints the frame, calls */PaintClient*, and finally resets the clipping.

In *ched*, there isn't enough information on the NeWS side for */PaintClient* to repair the display. It needs to send a message to the client asking it to send over the repair code. It would be incorrect to have */PaintClient* send the message since it is called in the midst of the damage repair context established by */FixFrame.* The repair cannot actually be started until the client gets the message and sends the code to handle it. Another reason that it cannot be started yet is that requests in flight from the client to the server have to be executed since they were generated without knowledge that a repair was needed. *ched* does the following:

> It overrides */FixFrame* with a procedure that send a message to the client side. So when damage occurs, nothing else happens immediately. In particular, the record that NeWS keeps of what region has been damaged is untouched. If more damage occurs before the application side gets around to dealing with the first damage request, the two damages will get merged and the application will see them as one.
>
> When the client side of *ched* receives one of these messages, it replies with a message that contains a "start redraw" request, the PostScript code necessary to redraw the window, and an "end redraw" request. Start and end redraw are essentially the same as */FixFrame* broken in half: the part before */PaintClient* and the part following.

8.9 Selections

When many people first see NeWS they are seduced by the fact that they can send code down to the server. For performance, they try to avoid client/server messages by putting as much code as possible in the server. Sometimes this is reasonable, and sometimes it not. The decision needs about client/server distribution of code should be based on whether the programming environment in the server is appropriate.

In the case of *ched*, it has a quite elaborate database describing the document and its formatting properties. This database has to be accessed very rapidly using quite sophisticated algorithms. The PostScript languageis not a very good language for this kind of processing, C is far better. So almost all of *ched* is written in C, with only a thin layer being written using PostScript code.

An example of a situation in *ched* where this tradeoff between C and PostScript is hard to make is in the code that handles selections. From the users point of view, making a selection progresses through three stages: 1) The user positions the mouse at on end of the selection and presses down on the mouse button. The caret appears as a vertical line at the division between two characters nearest the mouse. 2) The user moves the mouse with the button down to extend the selection. As the mouse is being moved, the caret is echoed as a line that underlines the characters between the start of the selection and the position of the mouse. 3) Finally the user lifts his finger off the mouse button, which defines the end of the selection.

The middle phase is difficult because the echoing of the underline that indicates the extent of the selection depends critically on the text that is selected: it is always at the baseline of the text and ends at the boundary between two characters. This involves a fairly expensive calculation that uses the document database. There are two choices: send messages back and forth between the client and the server each time the mouse moves and let the client side deal with the update, or download enough code so that the server side can deal with it. In *ched* the first choice was taken. This choice makes the server side of the application quite small and puts the responsibility for efficient calculation on the client side. Performance of *ched* depends heavily on the performance of message passing between the server. In most environments, the performance is very good, so *ched* works very well. If *ched* is run at some distance from the server, with low speed communication lines or network gateways in the way, it will perform poorly. On the other hand, a bigger investment in server-side code could have been made, but it probably would not perform as well as the C implementation.

8.10 Typeahead.

No computer is ever fast enough. Given an editor that is doing complicated real-time animation of the typesetting of a document, in complicated situations it is probably going to be possible to type faster than the editor can echo, if it is echoing each character as it receives it. This is the well-known typeahead problem. If implemented properly, the model/view/controller paradigm has the advantage that the typeahead problem disappears. The key factor is that the connection between the model and the controller is decoupled from the connection between the model and the view. When the model is updated, the view is not necessarily updated. The controlling loop of *ched* looks like this:

1. While there are user keystrokes in the input queue, process them. These update the model only.

2. Invoke the view to update the screen according to the new state of the model.

3. Wait for the input queue to become non-empty

4. Go back to step **1**.

Input events are being batched together. If the user manages to type faster than the system can echo, then it will just start echoing more keystrokes at once, quickly catching up.

8.11 How to get Ched

Ched can be obtained by sending electronic mail to "*news-archive@sun.com*" (from the ARPAnet) or "*sun!news-archive"* from usenet. The subject field of the message you send should be "*send ched"*. An automated mail handling program will send the source file to you by return mail. The *news-archive* is a collection of generally useful NeWS documents and sample programs. A subject line of "*help*" will return to you a description of the archive and how to get information from it.