



## DUMB VIRTUE -- Virtual terminals on the Mac under UNIX

Kevin Eric Saunders, Project Manager

Cornell's Laboratory of Atomic and Solid State Physics (LASSP), having committed itself to the purchase of 10 Macintoshes to be used as terminals for its VAX-11/750 running under 4.2 BSD UNIX, is currently developing windowing graphics terminal software for the Mac. We are using Stanford University's SUMACCS C-language cross-compiler for the Macintosh; the SUMACCS+UNIX+VAX development system provides an environment distinctly superior to that of the Lisa Workshop's "enhanced" p-system OS. In addition to offering superior utilities, edit-compile-go cycles are reduced to around 2 minutes on the VAX.

The current (.01) version, which has been hacked out of the SUMACCS translation of grow, performs (slow) serial communications with 4.2BSD UNIX, and uses TextEdit calls to store, edit, and display in a single resizable window. The next improvement to be made to the program will be the generalization of the window-management code to handle multiple text windows.

Eventually, the package should support:

- Different window types to support UNIX shell interaction, editing using the vi full screen editor, and graphics;
- The association of Mac windows with UNIX processes; and
- File transfer capabilities so that Macintosh files can be up/downloaded to/from 4.2BSD machines.

Anyone interested in contributing suggestions, helpful hints, ironic jokes, or even code, should contact:

Kevin Saunders (kevin@clarkvax, soon to be kevin@lasspvax)  
513 Clark Hall 607-256-3963  
Cornell University  
Ithaca, New York 14850

7/31/84

dumb virtue: Design and Implementation of a Terminal Emulation Program.  
Kevin Eric Saunders

The Macintosh personal computer combines adequate computing power and exceptional graphics capabilities, which helps make it an excellent choice for terminal emulation applications. "dumb virtue" is a program intended to enhance interaction with mainframe computers by providing several functions in one package: 1) key-driven editing (and simple--asynchronous!--draft printing) using the TextEdit Macintosh ROM routines, 2) tty-like communications in TextEdit windows, so that input and output can be recorded, 3) ANSI X3.64 (DEC VT100) terminal emulation supporting standard Macintosh fonts, 4) support for associating windows with virtual terminals on a BSD UNIX system using John Bruner's uw package.

To provide a functional tty-like interface using TextEdit, which does not handle frequent addition of chunks of text speedily, and to allow the ANSI terminal emulator to handle variable-width fonts, it was necessary to use an approach to processing host output which delayed processing of the output as long as the user's impatience to see output would allow. Thus, the flow of output processing resembles this sketch:

```
Wait for the output timer (settable by the user) to expire;  
Wait until a "reasonable" amount of output has accumulated; then:  
    Demultiplex output according to active window type;  
    For each type:  
        Loop:  
            Route Terminal and Window commands to appropriate routine;  
            Scan output for control characters;  
            On finding a control character, handle data up to that point;  
            Interpret control characters;  
            Handle remaining data  
        Update ANSI emulator window if necessary;
```

The ANSI terminal emulator relies on the additional lag mechanism, which delays drawing the screen until new output has ceased (with some "time-critical" exceptions), to help avoid premature drawing. This improves its performance, because the emulator does not use bit scrolling to insert characters in lines. The use of bit scrolling, which must come immediately before a character insertion, would give a "snappier" appearance when inserting one character, and the emulator has recently been modified to do vertical bit scrolling to give this effect when scrolling. However, repeated bit scrolls (caused, say, by the insertion of a string in the text) cost more than one delayed DrawText() call, so overall performance and appearance can be made "cleaner" by avoiding them.

Implementation was initially done with the SUMACCS cross-compiler running under 4.2BSD UNIX. Significant enhancements in functionality, and a substantial reduction in code size, were realized when the code was ported to Manx Aztec C on the Macintosh. It is unfortunate that the incompatibilities between the compilers are pronounced; Aztec offers better reliability and functionality, but UNIX enhances productivity in managing program text (through global modifications, searches for string matches in text files without leaving the editing environment, and provision of many powerful text-munging utilities).

\*\*\*\*\* Productivity \*\*\*\*\*

dumb virtue 1985 - 1988

```
cd {cometroot}src:dv.1.3src:  
count ~.c | sort -b -d -f 3
```

Total	10356	228153
-------	-------	--------

escape.c	1479	32756
main.c	1362	29444
menu.edit.c	1117	26629
commutil.c	1012	22807
comm.c	875	18484
startup.c	543	12484
windmaint.c	568	12324
menu.file.c	569	11697
util.c	499	9121
menu.top.c	396	8656
windcreat.c	298	8045
scroll.c	331	7530
menu.echo.c	272	7092
menu.comm.c	355	6129
menu.wind.c	187	4315
menu.size.c	167	3801
menu.c	165	3029
uwint.c	97	2223
key.c	63	1000
strpas-c.c	1	587

```

DEFS=-DPAIDFOR -DUW -DALLFINDSEL -DDORES -DTEWINDSWITCH -DLINEPRT -DNEWFIND -DASYNCWRITE -DTRYSERD -DVICONF -DVIATTRIB -DUSEUNDO -DTEKEXIS

#-DPAIDFOR routines compiled in for paying customers!
#-DUW      compile unix window routines
#-DUSEUNDO  compile undo routines
#-DKEYMACROS key macro routines
#-DALLFINDSEL use findsel routine to find selection point
#-DDORES    use resource configuration file
#-DVICONF   use configuration info to set size of emulator
#-DTEWINDSWITCH make a new output window when old one full
#-DLINEPRT  routines for using menu bar as prompt line present
#-DNEWFIND  new find routines, including backward search
#-DASYNCWRITE do asynchronous writes when sending
#-DTEKEISTS include the tek window routines
#-DTRYSERD  try to use the Macintosh RAM serial driver
#-DVIATTRIB  do VT100 attribute processing
#-DVIDUMP   adds dump routine for speed testing

#not used
#-DDOUPD   moving down a line in vi wind causes update
#-DVRT     use vertical retrace for input log
#-DSEGMENT  do UnloadSeg when called for--very RISKY!

OFLAGS=
FLAGS=
#+iinclude.o
#+idefs.o
#LIB=hd:lib/
OLIB=[MPW]Aztec:CObjects:

OBJ=comm.o commutil.o escape.o key.o main.o menu.o menu.comm.o menu.echo.o menu.edit.o menu.file.o menu.size.o menu.top.o menu.wind.o scro
HOLOBJ=comm.o commutil.o escape.o key.o main.o menu.o menu.comm.o menu.echo.o menu.edit.o menu.file.o menu.size.o menu.top.o menu.wind.o s
SEGOBJ=commutil.o key.o main.o menu.o menu.comm.o menu.echo.o menu.edit.o menu.file.o menu.size.o menu.top.o menu.wind.o scroll.o util.o u

.c.o:
echo '---> '$*.c; c $(CFLAGS) $(DEFS) -ps -bm -bs -qq -wd -wn -ww -o $@ $*.c

#Wednesday, December 18, 1991 3:46:48 PM
#modified for Aztec 5.2 ANSI options, removed '+N'
#added -ps (small memory model, 16 bit integers)
#-bm (8 char macsbug symbols embedded in code)
#added -bs (sdb debugger data)
#added -qv (verbose option)
#added -ww (continue compilation after five errors rather than aborting)
#  duplicate -y '$*.c' hd:'$*.c'
#-hi em.dmp

dv: dv.o dv.rc dv.rsrc
  rgen -f dv.rc
  SetFile -a B dv

#$(LIB)sacroot.o

dv.o: $(OBJ)
#  ln -w +M -o dv.o $(LIB)sacroot.o $(HOLOBJ) -lm -lc
#  ln -g -xm -xs 8000 -o dv.o \
#  $(LIB)sacroot.oss $(LIB)crt0.oss \
#  $(HOLOBJ) -lc16 -lm16 a16.lib

seg: $(OBJ)
  ln +M -o dv.o $(SEGOBJ) +00 sacroot.o -lm -lc

#defs.o: commdefs.h commmenudefs.h cursormenudefs.h echomenudefs.h editmenudefs.h filmenudefs.h mac.h menudefs.h videfs.h windmenudefs.h
#  C +hdefs.o superdefs.h

comm.o: comm.c defs.h main.h commdefs.h videfs.h vi.h uw.h
#  C $(FLAGS) $(DEFS) comm.c

commutil.o: commutil.c defs.h main.h comm.h commdefs.h commmenudefs.h
#  C $(FLAGS) $(DEFS) commutil.c

enc: enc.c
#  C -o enc enc.c

escape.o: escape.c defs.h main.h comm.h videfs.h menudefs.h echomenudefs.h uw.h
#  C $(FLAGS) $(DEFS) escape.c

key.o:
#  C $(FLAGS) $(DEFS) key.c

main.o: main.c defs.h comm.h commdefs.h menudefs.h windmenu.h windmenudefs.h echomenudefs.h cursormenudefs.h uw.h startup.h
#  C $(FLAGS) $(DEFS) main.c

```

```
menu.o: menu.c defs.h main.h menudefs.h windmenudefs.h commmenudefs.h echomenudefs.h encopynotice
#   C $(FLAGS) $(DEFS) menu.c

menu.comm.o: menu.comm.c defs.h main.h comm.h mainmenu.h menudefs.h comm.h commdefs.h echomenudefs.h commmenudefs.h startup.h
#   C $(FLAGS) $(DEFS) menu.comm.c

menu.echo.o: menu.echo.c defs.h main.h comm.h mainmenu.h menudefs.h commdefs.h echomenudefs.h windmenudefs.h uw.h
#   C $(FLAGS) $(DEFS) menu.echo.c

menu.edit.o: menu.edit.c defs.h main.h videfs.h vi.h mainmenu.h menudefs.h editmenudefs.h cursormenudefs.h mac.h
#   C $(FLAGS) $(DEFS) menu.edit.c

menu.file.o: menu.file.c defs.h main.h comm.h mainmenu.h menudefs.h filemenudefs.h uw.h
#   C $(FLAGS) $(DEFS) menu.file.c

menu.size.o: menu.size.c defs.h main.h mainmenu.h menudefs.h videfs.h vi.h
#   C $(FLAGS) $(DEFS) menu.size.c

menu.top.o: menu.top.c defs.h main.h mainmenu.h menudefs.h utildefs.h topmenudefs.h
#   C $(FLAGS) $(DEFS) menu.top.c

menu.wind.o: menu.wind.c defs.h main.h comm.h mainmenu.h comm.h menudefs.h echomenudefs.h windmenudefs.h cursormenudefs.h uw.h
#   C $(FLAGS) $(DEFS) menu.wind.c

scroll.o: scroll.c defs.h videfs.h main.h vi.h mac.h
#   C $(FLAGS) $(DEFS) scroll.c

startup.o: startup.c defs.h main.h comm.h commdefs.h menudefs.h filemenudefs.h echomenudefs.h windmenudefs.h commmenudefs.h mainmenu.h
#   C $(FLAGS) $(DEFS) startup.c

util.o: util.c main.h defs.h utildefs.h comm.h
#   C $(FLAGS) $(DEFS) util.c

uwint.o: uwint.c main.h comm.h defs.h mainmenu.h menudefs.h windmenudefs.h windmenu.h uw.h
#   C $(FLAGS) $(DEFS) uwint.c

windcreat.o: windcreat.c defs.h main.h videfs.h vi.h uw.h startup.h
#   C $(FLAGS) $(DEFS) windcreat.c

windmaint.o: windmaint.c main.h uw.h videfs.h mainmenu.h menudefs.h editmenudefs.h echomenudefs.h filemenudefs.h topmenudefs.h cursormenud
#   C $(FLAGS) $(DEFS) windmaint.c
```

```
* input for resource compiler
*
dv
APPLdumv

* menu[0] = NewMenu(appleMenu, "\024");
Type MENU
,65
\14
About dumb virtue
Key->Menu      -Space-/
(-

Type MENU
* menu[FILE] = NewMenu(fileMenu, "File");
,66
File
Get .../G
Save .../F
Save selection ...
Delete ...
(-
Print selection
New page
Cancel Print
Set Header ...
Reset printer
Line spacing ...
Manual feed
LaserPrint
(-
Capture ...
Cancel Capture
Send line      -Return-/^\0D
Send all      -Option-Return-/\0D
Cancel Send
(-
Save Configuration ...
Quit/Q

Type MENU
* menu[EDIT] = NewMenu(editMenu, "Edit");
,67
Edit
Undo/Z
(-
Cut/X
Copy/C
Paste/V
Clear
Select all/
(-
Delete right      -BS-/
Shift left/S
" right      -Option-/S
Key deletes selection/-
Dump cuts

Type MENU
* menu[CURSOR] = NewMenu(cursorMenu, "Cursor");
,68
Cursor
Find '/'
Set '/';
Left/H
Down/J
Up/K
Right/L
Word left/,
" right.
Sentence left/[ 
" right]
Page down/N
" up/M
Match brackets/B
Anchor selection/A
Tag 'uh selection .../U
Yank tag .../Y

Type MENU
* menu[WINDOW] = NewMenu(windowMenu, "Window");
,69
Window
Input/I
Output/O
(-
```

12/10/94 13:18

dv.rc

2

0/0<I!=  
1/1<I  
2/2<I  
3/3<I  
4/4<I  
5/5<I  
6/6<I  
7/7<I  
8/8<I  
9/9<I  
(-  
Display mode/D  
Scratchpad -Tab-/

Type MENU  
\* menu[ECHO] = NewMenu(echoMenu, "Echo");  
,75

Echo  
Record input/R  
Edit locally/E  
Prompt skip/P  
(-  
Send Break  
Option = Ctrl!\12  
Literal output  
Output wrap ...  
BS = DEL  
` = ESC  
CR = CR-LF  
UNIX windows  
Input lag ...  
Output lag ...  
Terminal wrap ...  
XON-XOFF!\12  
VI dump

Type MENU  
\* menu[COMM] = NewMenu(commMenu, "Comm.");  
,71

Comm  
Modem<U  
Modem port  
Printer port<U  
57600-Baud Rate  
19200  
9600  
4800  
2400  
1200!\12  
300  
None -Parity!\12  
Even  
Odd  
1 -Stop Bits  
1.5  
2!\12  
6 -Data Bits  
7!\12  
8

Type MENU  
\* menu[FONT] = NewMenu(fontMenu, "Fonts");  
,72

Fonts

Type MENU  
\* menu[TOP] = NewMenu(topMenu, "FrontWindow");  
,74  
Top  
File segment...  
Scroll  
Show selection size  
Find selection//  
Find .../^  
Go to line .../=  
Word wrap/W  
Reformat  
Free Memory

INCLUDE dv.o  
INCLUDE dv.rsrc

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
/* comm vars */

extern char * sdbuf;      /* the serial driver's buffer */
extern char * readbuf;    /* the read buffer */
extern char * stake;      /* holds a position during interpretation */
extern int rcount;        /* count of read-ready chars -- long ex. SUMACC */
extern char mask;         /* mask for parity bit */

extern char * headerstr;  /* print header */
extern int oldport;       /* which port modem is using */

/* comm modes */
extern char literal;     /* no interpretation of text window output */
extern char switchnl;    /* if TRUE CR=LF && LF=CR */
extern char putskip1;
extern char bstodel;      /* map BS to DEL? */
extern char deesc;        /* map `` to ESC? */
extern char noserd;       /* no ram serial driver available */
extern char escapemode;   /* if TRUE send non-control chars to interpreter */
extern char laserprint;  /* if TRUE use Apple text-streaming routines */

extern char local, logmode, skipprompt;
/* communications modes: local, record input, skip to blanks when sending
 * output (for record input mode)
 */
extern struct Port {
    int in;
    int out;           /* serial port refNums */
    int speed;
    int data;
    int parity;
    int stop;
    int transon;       /* TRUE if async print or send job on */
    int transcont;     /* TRUE if async trans ready to resume */
    int transfin;      /* TRUE if async trans is completed */
    char * printstart; /* ptr to current pos */
    char * printend;   /* ptr to end */
    int transprint;    /* TRUE transmission is for printing */
    int addcr;         /* TRUE add CR after printing */
    int addlf;         /* TRUE add LF after printing */
    Handle transtxhand; /* TE record text or copy locked down for transmission */
    int copied;        /* TRUE if we made a copy of the text */
} modem, printer, * defport;
extern char hostxoff;
```

5/11/88 13:52

commdefs.h

1

```
/* copyright kevin eric saunders 1986 */  
/* serdefs.h */  
/* serial port defs */
```

```
#define SERBUFSIZE      3000  
#define SHUTDOWN        1000  
#define TURNON          200  
#define READBUFSIZE    2048  
#define TIMEOUT         360  
#define NOSKIP          SP  
#define MODEMOFF        1  
#define ETX   '\003'  
  
#define PRHEADMAX     80
```

1/29/88 20:12

commenudefs.h

1

/\* copyright Kevin Eric Saunders 1986--All Rights Reserved \*/

```
/* comm menu defs */
#define MODEMOFF    1
#define MODEPORT    2
#define PRINTERPORT 3
#define BBASE        4
#define B57600      BBASE
#define B19200      BBASE + 1
#define B9600       BBASE + 2
#define B4800       BBASE + 3
#define B2400       BBASE + 4
#define B1200       BBASE + 5
#define B300        BBASE + 6
#define PARITYBASE  11
#define NOPARITY    PARITYBASE
#define EVENPARITY  PARITYBASE + 1
#define ODDPARITY   PARITYBASE + 2
#define STOPBASE    14
#define ONESTOP     STOPBASE
#define ONEHALFSTOP STOPBASE + 1
#define TWOSTOP     STOPBASE + 2
#define DATABASE    17
#define SIXDATA     DATABASE
#define SEVENDATA   DATABASE + 1
#define EIGHTDATA   DATABASE + 2

#define LASTCOMM    EIGHTDATA + 1
```

1/23/88 16:27

cursormenudefs.h

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
/* cursor menu defs */
```

```
#define FINDCURS      1
#define SETCURS        2
#define LEFT           3
#define DOWN          4
#define UP             5
#define RIGHT          6
#define WORDLEFT       7
#define WORDRIGHT      8
#define SENTLEFT       9
#define SENTRIGHT     10
#define PAGEDOWN      11
#define PAGEUP         12
#define MATCHBRACK    13
#define ANCHOR         14
#define TAG            15
#define YANK           16
#define CURSORLAST    17
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#define NUL '\000'
#define SOH '\001'
#define STX '\002'
#define ENTER '\003'
#define EOT '\004'
#define ENQ '\005'
#define ACK '\006'
#define BEL '\007'
#define BS '\010'
#define TAB '\011'
#define LF '\012'
#define VT '\013'
#define FF '\014'
#define CR '\015'
#define SO '\016'
#define SI '\017'
#define DLE '\020'
#define XON '\021'
#define DC1 '\021'
#define DC2 '\022'
#define XOFF '\023'
#define DC3 '\023'
#define DC4 '\024'
#define NAK '\025'
#define SYN '\026'
#define ETB '\027'
#define CAN '\030'
#define EM '\031'
#define SUB '\032'
#define ESC '\033'
#define FS '\034'
#define GS '\035'
#define RS '\036'
#define US '\037'
#define SP '\040'
#define DEL '\177'

/* window types */

#define TEXTWIND 1
#define VIWIND 2
#define TEKWIND 3
#define BUGWIND 4
#define DOWNWIND 5

#define VIWINDNUM 10
#define BUGWINDNUM 11
#define TEKWINDNUM 12
#define MAXWIND 13

/* ROM type constants */

#define ROM64 0x8000

/* TE constants */

#define TEOFFSET 2
#define TEMAXLENGTH 32767

#define HUGE 0xFFFFFFFF /* maximum file length */

#define NOSCROLL 0
#define HSCROLL 1
#define VSCROLL 2
#define VISCROLL 3

#define TRUE (-1)
#define VIS 1
#define FALSE 0
#define INVIS 0

#define windstruct(A) ((struct window *) ((WindowPeek) A)->refCon)
/* makes a window refcon a window structure */

#define swindptr(A) ((struct window *) ((WindowPeek) A)->refCon)
/* makes a window refcon a pointer to a window structure */

```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
```

```
/* defines used by echo menu */
```

```
#define LOGMODE      1
#define LOCAL        2
#define SKIPROM      3
#define SKIPECHO     4
#define BREAK         5
#define SWITCHLOCK   6
#define LITOUT        7
#define WRAPOUT       8
#define BSTODEL      9
#define DECESC       10
#define SWITCHNL     11
#define UWTOGGLE     12
#define SETINLAG     13
#define SETOUTLAG    14
#define SETVIWRAP    15
#define DOFLOW       16

#define VIDUMP      17
```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
/* edit menu items */

#define UNDO      1
#define EDITSKIP   2
#define CUT       3
#define COPY      4
#define PASTE     5
#define CLEAR     6
#define SELECTALL 7
/* skip */
#define DELRIGHT   9
#define SHIFTLEFT  10
#define SHIFTRIGHT 11
#define SELKILL    12
#define DUMPCUT   13
#define EDITLAST   14

/* the undo operations need to follow the real range to avoid conflict */
#define UNDOCUT   40
#define UNDOCOPY   41
#define UNDOPASTE  42

#define CUTINS    50
/* special cut operation for TEKey-like behavior w/ UNDO */

/* operation numbers for tagadjust, which resets tag values on cut/paste */
#define TADD      1
#define TCUT      2
```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
/* File menu items */

#define GET      1
#define SAVE     2
#define SAVESELECT 3
#define FILEDELETE 4
    /* skip */
#define PRINTSEL  6
#define NEWPAGE   7
#define PRINTCANCEL 8
#define PRINTHEADER 9
#define PRINTRESET 10
#define DBLSPACE  11
#define PAGEWAIT  12
#define LASERPRINT 13
    /* skip */
#define CAPTURE  15
#define CAPKILL  16
#define SENDLINE  17
#define SENDALL   18
#define SENDCANCEL 19
    /* skip */
#define SAVECONF 21
#define QUIT     22
#define LASTFILE 23
```

2/14/87 0:22

mac.h

1

```
/* mac globals */

#define TEScrLen (* (short *) 0xab0)
#define TEScrHandle (* (Handle *) 0xab4)
#define KeyRepThresh (* (short *) 0x190)

#define SFSaveDisk (* (short *) 0x214)
#define CurDirStore (* (long *) 0x398)
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
#include "defs.h"

#include "ctype.h"
#include "stddef.h"
#include "stdio.h"
#include "stdlib.h"

#include "toolutils.h"
#include "quickdraw.h"
#include "controls.h"
#include "desk.h"
#include "dialogs.h"
#include "disks.h"
#include "errno.h"
#include "events.h"
#include "fcntl.h"
#include "files.h"
#include "fonts.h"
#include "memory.h"
#include "menus.h"
#include "OSEvents.h"
#include "osutils.h"
#include "packages.h"
// #include "pb.h"
#include "printing.h"
#include "resources.h"
#include "retrace.h"
#include "scrap.h"
#include "SegLoad.h"
#include "serial.h"
#include "stat.h"
#include "string.h"
#include "Errors.h"
#include "textedit.h"
#include "time.h"
#include "types.h"
#include "windows.h"

extern short growRect[], dragRect[];
/* growRect & dragRect are bounds */

extern EventRecord myEvent;
extern WindowPtr whichWindow;
extern CursHandle iBeam, watch;

extern short romtype;      /* ROM type on mac in use */
extern char error[50];     /* global error message buffer */

extern short nullrect[];   /* used for Clip nulling */

extern char vion;
/* global for communications defaults and which window's on top */

struct scrmap {
    int hit;
    int drawn;
};

struct virecord {
    int virows;           /* real number of rows */
    int virowmin;         /* first active row */
    int virowmax;         /* scroll if row to become > max */
    int vicols;           /* number of cols in use */
    int vircols;          /* NOT USED the REAL number of cols in the array */
    int vicolmax;
    int visize;           /* from 0 to maxrow * VICOLS; varies with scroll window size */
    int virsize;

    int lineheight;
    int fontheight;
    int fontdescent;

    /* maps changes to screen by row with column change start position (xpos) */
    /* and last pen location for eraserect.right */

    char insertmode;      /* insert on flag */
    char vitouched;       /* flag indicating redraw needed */
    /* does the screen need to be redrawn when out of input data? */
    char inlastcol;        /* TRUE when x=80 & nlglitch in effect */
    char minorigin;        /* TRUE when home == virowmin */

    int xpos;              /* horizontal cursor location */
    int ypos;              /* vertical cursor location */
    Rect cursrect;         /* vi cursor rectangle */
    char cursor;           /* cursor state, on or off */
};

```

```
char * vitextp;      /* ptr to current vi character */
char * vitextbase;   /* ptr to base of array */
char * vitextend;    /* ptr to end of array */
struct scrmap * scrmap; /* the map of the screen drawing state */
char attrib;         /* screen attribute, normal */

/* for save and restore cursor */
int oldx;
int oldy;
char * oldvip;

/* ANSI interpretation modes */
char foundbrack;   /* TRUE if '[' found after ESC */
char foundmark;    /* TRUE if '?' found, indicating DEC modes */
char foundpound;   /* TRUE if '#' found, indicating DEC double modes */
char foundparen;   /* TRUE if '(' found, indicating undocumented DEC mode */
char foundbang;    /* TRUE if '!' found, indicating my private commands */

int scroffset;     /* offset for scrollbar loc so user can scroll */
int viwrap;        /* pixel at which vi does wrap */
int offset;         /* negative offset for virtual screen scrolling when drawing */

int arg1;
int arg2;
int * argptr;
};

struct window {
    WindowPtr    ptr;
    TEHandle     hTE;
    struct virecord virec;
    char escapemode; /* TRUE in escape sequence */
    ControlHandle hscroll;
    ControlHandle vscroll;
    char    scroll;   /* scroll on or off */
    char    dirty;    /* text window has been touched if TRUE */
    short   type;
    short   fvol;     /* volume ref of open file */
    long    fdir;     /* directory ref of open file */
    int     cstate;   /* REP UIO status of a window */
} extern window[];

extern struct window * inputwind, * outputwind, * bugwind, * viwind, * tekwind;
```

9/21/86 1:29

mainmenu.h

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
```

```
extern MenuHandle menu[];  
extern int oldsize, oldfont;
```

```
extern scrollTE(), resetspeed();
```

12/10/94 13:21

menudefs.h

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */  
/* MENU constants */
```

```
#define APPLE      0  
#define FILE       1  
#define EDIT       2  
#define CURSOR     3  
#define WIND       4  
#define MECHO      5  
#define COMM       6  
#define FONT       7  
#define SIZE       8  
#define TOP        9  
#define NUMMENU    10 /* number of menus */
```

```
/* menu IDs */
```

```
#define appleMenu  65  
#define fileMenu   66  
#define editMenu   67  
#define cursorMenu 68  
#define windMenu   69  
#define echoMenu   75 /* changed from 70 */  
#define commMenu   71  
#define fontMenu   72  
#define sizeMenu   73  
#define topMenu    74
```

2/6/88 6:52

startup.h

1

/\* copyright Kevin Eric Saunders 1986--All Rights Reserved \*/

```
extern int appcount;
extern short deffont;
extern short defsize;
extern short defvifont;
extern short defvisize;
extern int defvirows;
extern int defvocols;
```

9/21/86 1:14

super.h

1

```
#include "main.h"
#include "comm.h"
#include "commdefs.h"
#include "commmenudefs.h"
#include "cursormenudefs.h"
#include "echomenudefs.h"
#include "editmenudefs.h"
#include "filemenudefs.h"
#include "mac.h"
#include "menu.h"
#include "menudefs.h"
#include "startup.h"
#include "util.h"
#include "uw.h"
#include "vi.h"
#include "videfs.h"
#include "windmenu.h"
#include "windmenudefs.h"
```

9/21/86 21:58

superdefs.h

1

/\* super defines file everybody can use with aztec +ifile option to speed up compilation \*/

```
#include "mac.h"
#include "commdefs.h"
#include "commmenudefs.h"
#include "cursormenudefs.h"
#include "echomenudefs.h"
#include "editmenudefs.h"
#include "filemenudefs.h"
#include "menudefs.h"
#include "utildefs.h"
#include "videfs.h"
#include "windmenudefs.h"
```

2/20/88 14:47

topmenudefs.h

1

```
/* item equivalents */
#define FILESEGMENT 1
#define SCROLL 2
#define SELSIZE 3
#define FINDSEL 4
#define FINDPROMPT 5
#define CURSTOLINE 6
#define WORDWRAP 7
#define REFORMAT 8
#define FREE 9
#define TOPLAST 10
```

2/27/88 8:21

utildefs.h

1

/\* copyright kevin eric saunders 1986 \*/

```
#define FINDMAX      1000
#define FORWARD       1
#define BACKWARD      2
```

```

/*
 * uw command bytes
 *
 * Copyright 1985 by John D. Bruner. All rights reserved. Permission to
 * copy this program is given provided that the copy is not sold and that
 * this copyright notice is included.
 *
 *
 * Two types of information are exchanged through the 7-bit serial line:
 * ordinary data and command bytes. Command bytes are preceded by
 * an IAC byte. IAC bytes and literal XON/XOFF characters (those which
 * are not used for flow control) are sent by a CB_FN_CTLCH command.
 * Characters with the eighth bit set (the "meta" bit) are prefixed with
 * a CB_FN_META function.
 *
 * The next most-significant bit in the byte specifies the sender and
 * recipient of the command. If this bit is clear (0), the command byte
 * was sent from the host computer to the Macintosh; if it is set (1)
 * the command byte was sent from the Macintosh to the host computer.
 * This prevents confusion in the event that the host computer
 * (incorrectly) echos a command back to the Macintosh.
 *
 * The remaining six bits are partitioned into two fields. The low-order
 * three bits specify a window number from 1-7 (window 0 is reserved for
 * other uses) or another type of command-dependent parameter. The next
 * three bits specify the operation to be performed by the recipient of
 * the command byte.
 *
 * Note that the choice of command bytes prevents the ASCII XON (021) and
 * XOFF (023) characters from being sent as commands. CB_FN_ISELW commands
 * are only sent by the Macintosh (and thus are tagged with the CB_DIR_MTOH
 * bit). Since XON and XOFF data characters are handled via CB_FN_CTLCH,
 * this allows them to be used for flow control purposes.
 */

#define IAC      0001      /* interpret as command */
#define CB_DIR    0100      /* command direction: */
#define CB_DIR_HTON 0000   /* from host to Mac */
#define CB_DIR_MTOH 0100   /* from Mac to host */
#define CB_FN     0070      /* function code: */
#define CB_FN_NEWW 0000    /* new window */
#define CB_FN_KILLW 0010   /* kill (delete) window */
#define CB_FN_ISELW 0020   /* select window for input */
#define CB_FN_OSELW 0030   /* select window for output */
#define CB_FN_META 0050    /* add meta to next data char */
#define CB_FN_CTLCH 0060   /* low 3 bits specify char */
#define CB_FN_MAINT 0070   /* maintenance functions */
#define CB_WINDOW  0007      /* window number mask */
#define CB_CC     0007      /* control character specifier: */
#define CB_CC_IAC 0001    /* IAC */
#define CB_CC_ON  0002    /* XON */
#define CB_CC_OFF 0003   /* XOFF */
#define CB_MF     0007      /* maintenance functions: */
#define CB_MF_ENTRY 0000   /* beginning execution */
#define CB_MF_EXIT 0007   /* execution terminating */
#define NWINDOW    7        /* maximum number of windows */

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

extern char uwon; /* on/off flag */

```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

/* viwind vars */

extern struct virecord vr;
extern Rect virect;
extern char clearscreen;      /* screen must be completely redrawn */

#ifndef OLD
extern int virows;           /* varies with scroll window size */
extern int virowmin;
extern int virowmax;

extern int vicols;           /* current number of columns */
extern int vircols;          /* the REAL number of columns in the array */
extern int vicolmax;
extern int visize;
    /* from 0 to maxrow * VICOLS; varies with scroll window size */
extern int virsize;          /* the REAL size of the array */

extern int lineheight;
extern int fontheight;
extern int fontdescent;
extern char insertmode;

extern char vitouched;
    /* does the screen need to be redrawn when out of input data? */
    /* used in main.c */

extern Rect cursrect;
extern int vicount; /* count of chars in vi line being put out */
extern int xpos, ypos;

extern char * vitextp, * vitextbase, * vitextend;

extern int scroffset; /* offset for scrollbar loc so user can scroll */

extern int viwrap;      /* pixel at which vi does wrap */

#endif
```

```
/* viwind defs */

#define VIROWS      30      /* the REAL number of rows */
#define VICOLS      80      /* the REAL number of columns */
#define VIROWMAX    29      /* the REAL number of rows -1 */
#define VICOLMAX    79      /* the REAL number of columns -1 */
#define VISIZE     2400     /* the REAL size of the vi buffer(s) */
#define VT100ROWS   24      /* the number of rows to default to */

/* boundaries */
#define VITOP 2
#define VILEFT 2
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#define VIRIGHT 500
```

12/2/87 7:10

windmenu.h

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
```

```
extern int oldin;      /* short ex. SUMACC */
extern int oldout;     /* short ex. SUMACC */
```

```
/* actions taken on user window selection in windmenu */
extern int makefront;
extern int makeinput;
```

```
extern int makeoutput;
```

2/20/88 15:08

windmenudefs.h

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
/* defs for window menus */

#define INPUT      1
#define OUTPUT     2
#define WINDMBASE   4
#define NEXTDISPLAY 15
#define SYSSWITCH   16
#define MODEMOFF    1

#define WS_INPUT    0x01      /* this is an input window */
#define WS_OUTPUT   0x02      /* this is an output window */
#define WS_RECORD   0x04      /* record lines locally, send on CR */
#define WS_LOCAL    0x08      /* local edit mode, no xmit */
#define WS_PROMPT   0x10      /* skip prompt and spaces */
#define WS_UW       0x20      /* window is a uw window */
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "commdef.h"
#include "editmenu.h"
#include "videfs.h"
#include "vi.h"
#include "uw.h"

#define TESWITCHLENGTH 30000

/* the serial port data structure */

struct Port {
    int in;
    int out;           /* serial port refNums */
    int speed;
    int data;
    int parity;
    int stop;
    int transon;      /* TRUE if async print or send job on */
    int transcont;    /* TRUE if async trans ready to resume */
    int transfin;     /* TRUE if async trans is completed */
    char * printstart; /* ptr to current pos */
    char * printend;  /* ptr to end */
    int transprint;   /* TRUE transmission is for printing */
    int addr;          /* TRUE add CR after printing */
    int addlf;          /* TRUE add LF after printing */
    Handle transtxhand; /* TE record text or copy locked down for transmission */
    int copied;        /* TRUE if we made a copy of the text */
} modem, printer, * defport;

extern char hostxoff;

/* serial port read routine variables */

char * sdbuf;          /* the serial driver's buffer */
char * readbuf;         /* the read buffer */
char * stake;           /* holds the last un-interpreted position in the buffer */
                       /* used by all output routines */
char mask = '\177'; /* masks off 8th bit */
long rcount;            /* count of read-ready chars -- long ex. SUMACC */

extern short oldport;  /* which port modem is using */

/* comm modes */
char literal = FALSE;  /* no interpretation of text window output */
char switchnl = FALSE; /* if TRUE CR=LF & LF=CR */
char putskip1 = NOSKIP; /* don't break to write output on this ^char */
char bstodel = FALSE;  /* map BS to DEL? */
char decesc = FALSE;   /* map ` to ESC? */
char escapemode = FALSE; /* if TRUE send non-control chars to interpreter */
char laserprint = FALSE; /* if TRUE use Apple text-streaming routines */

int fcapture = FALSE;  /* file being captured? */
int fdcapture = FALSE; /* fd of file being captured */
int capvol;             /* volume on which being captured */
long ncaptured = 0; /* # of bytes captured */
char saveonlf = FALSE; /* start file capture after next lf */
long captime;          /* time last small file capture chunk arrived */
int capwindtype;       /* saves type of window being captured */
struct window * capturewind;

/* comm modes for Unix Window compatibility--see uw.h */
char uwon = FALSE;     /* accept uw sequences? */
char uwmode = FALSE;    /* if TRUE send next char to interpreter */

extern longtoa();
extern struct window * tekwind;
extern char tewindswitch; /* make a new output window */

char headerstr[PRHEADMAX] = "Page header";

portRead(in, nbytes)
int in;
long nbytes; /* long ex. SUMACC */
{
    int result;
    extern long blockNBytes;

    /* we read what's in our buffer, */
    /* but only what we know is available NOW */
    /* This kluge courtesy of Apple's synchronous-only FSRead routine */

    if (nbytes < READBUFSIZE)
        rcount = nbytes;
}

```

```

else
    rcount = READBUFSIZE;      /* smaller of the two */

FSRead(in, &rcount, readbuf);
#define FILECAPTURE
#ifndef FILECAPTURE
if (fcapture) {
    if (filewrite(fdcapture, &rcount, readbuf)) {
        /* bad write, abort capture */
        errstr("file capture aborted");
        endcapture(outputwind);
    }
    else if (rcount != READBUFSIZE) {
        long fpos;

        errprompt(capturedone);
        rcount -= counttoCR(readbuf + rcount);
        /* save only up to last CR */

        SetFPos(fdcapture, fsFromMark, -fpos);
        /* will be set to EOF in endcapture() */
        ncaptured += rcount;
        endcapture(outputwind);
        return(rcount);
        /* show the prompt! */
    }
}
else
    captime = TickCount();
    ncaptured += rcount;
    return(rcount);
}
#endif
stake = readbuf;
#endif UW
do {
#endif
    switch (outputwind->type) {
        case BUGWIND:
        case TEXTWIND: {
            rcount = textoutput((int) rcount);
            break;
        }
        case VIWIND: {
            rcount = vioutput((int) rcount);
            break;
        }
#endif TEKEXISTS
        case TEKWIND: {
            rcount = tekoutput(outputwind, (int) rcount);
            break;
        }
#endif
        case DOWNWIND: {
            rcount = fileoutput((int) rcount);
            break;
        }
    }
#endif UW
} while (rcount);
#endif
/* do a systemtask to make sure DA's get enough cycles to work ok */
SystemTask();
return(rcount);
}

#endif FILECAPTURE

fileoutput(pcount)
int pcount;
{
    register char maskb;
    register unsigned char bufchar;
    register char * bufp;
    register char * bufend;
    long wcount;

    maskb = mask;
    bufend = stake + pcount;
    /* fix parity bit and convert LF to CR for UNIX */
    for (bufp = stake; bufp < bufend; bufp++) {
        bufchar = *bufp &= maskb;
#endif UW
        if (uwmode) {
            uwmode = FALSE;
            if (uwhandle(bufchar)) {
                stake += 2;
                return(bufend - stake);
            }
        }
    }
}

```

```

        /* return count of chars left */
    }
    else
        continue;
}
#endif SESCTL
if (sessionctl(bufchar)) {
    stake++;
    continue;
}
#endif
ncaptured++;
if (bufchar == LF || bufchar == ESC || bufchar == SOH) {
    if (switchnl) {
        /* remap CRLF so one can 'stty nl' on 4.2 BSD to
         * speed things up
        */
        if (bufchar == LF) {
            *bufp = CR;
            continue;
            /* keep on going */
        }
    }
    if (bufp != stake) {
        /* spit out normal text skipped so far */
        /* and move stake up */
        wcount = bufp - stake;
        filewrite(fdcapture, &wcount, stake);
        stake = bufp;
    }
    /* now we've put out the plain text, we can handle the */
    /* ASCII control codes */
    switch (bufchar) {
#endif UW
        case SOH: {
            /* next byte may be a UW command */
            uwmode = TRUE;
            break;
        }
#endif
        case LF: {
            /* a no-op for TE */
            stake++;
            break;
        }
        case ESC: {
            /* end of file transfer */
            wcount = bufp - stake;
            filewrite(fdcapture, &wcount, stake);
            endcapture(outputwind);
            return(bufend - stake);
            stake++;
            break;
        }
    }
}
/* end if < SPACE; else do nothing but increment bufp in for */
}
/* now we process text not yet written out */
if (bufp > stake) {
    wcount = bufp - stake;
    filewrite(fdcapture, &wcount, stake);
}
capttime = TickCount();
return(0);
}
#endif

char lineadded = FALSE;      /* TRUE if TE has seen a new line */
/* --prevents control jiggling */

textoutput(count)
int count;
{
    register char maskb;
    register unsigned char bufchar;
    register char * bufa;    /* real text ready for TE */
    register char * bufp;    /* reads through, real text copied back to bufa if != */
    register char * bufend;
    if ((outputwind->hTE)->selStart != (*outputwind->hTE)->teLength) {
        /* set cursor at end if not already there */
        tesetsel((long) (*outputwind->hTE)->teLength,
                  (long) (*outputwind->hTE)->teLength, outputwind->hTE);
    }
}
```

```

}

maskb = mask;
bufend = stake + count;
/* fix parity bit and convert LF to CR for UNIX */
for (bufa = bufp = stake; bufp < bufend; bufp++) {
    bufchar = *bufp & maskb;

#ifdef UW
    if (uwmode) {
        struct window * ooutwind;

        uwmode = FALSE;
        ooutwind = outputwind;
        if (uwhandle(bufchar)) {
            puttext(ooutwind, stake, bifa - stake);
            stake = ++bufp;
            return(bufend - stake);
            /* return count of chars left */
        }
        else
            continue;
    }
#endif
    if (literal) {
        /* incorporate ALL characters into TE record */
        *bufa++ = bufchar;
        continue;
    }
#endif SESCCTL
    if (sessionctl(bufchar)) {
        continue;
    }
#endif
    if (bufchar < SP || bufchar == DEL) {
        if (switchnl) {
            /* remap CRLF so one can 'stty nl' on 4.2 BSD to
             * speed things up
            */
            if (bufchar == CR) {
                bufchar = LF;
                lineadded = TRUE;
            }
            else if (bufchar == LF) {
                bufchar = CR;
                lineadded = TRUE;
            }
        }
#endif SLOWTE
        if (bufchar == TAB || (bufchar == putskip1 && !saveonlf) )
            /* don't write buffered text for ordinary chars to
             * make things faster
             * check saveonlf so CR will show up to trigger capture */
            continue;
        if (bufp != stake) {
            /* spit out normal text skipped so far */
            /* and move stake up */
            puttext(outputwind, stake, bufp - stake);
            stake = bufp;
        }
#endif
        /* now we've put out the plain text, we can handle the */
        /* ASCII control codes */
        switch (bufchar) {
#ifndef SESCCTL
            case NUL: {
                /* do nothing */
                break;
            }
#endif UW
            case SOH: {
                /* next byte may be a UW command */
                uwmode = TRUE;
                break;
            }
#endif
#ifndef TEKEXISTS
            case GS: {
                /* go into graphics mode */
                if (tekwind != NULL) {
                    outputwind = tekwind;
                    puttext(outputwind, stake, bifa - stake);
                    stake = ++bufp;
                    return(bufend - stake);
                }
                break;
            }
#endif
            case ESC: {

```

```

        escprep(); /* TODO parameterize */
        break;
    }
    case XOFF: {
        hostxoff = TRUE;
        break;
    }
    case XON: {
        hostxoff = FALSE;
        break;
    }
#endif
    case LF: {
        /* a no-op for TE */
        if (checksave()) {
            puttext(outputwind, stake, bufa - stake);
            stake = ++bufp;
            return(bufend - stake);
        }
        break;
    }
    case CR: {
        *bufa++ = bufchar;
        lineadded = TRUE;
        if (switchnl && checksave()) {
            puttext(outputwind, stake, bufa - stake);
            stake = ++bufp;
            return(bufend - stake);
        }
        break;
    }
    case BS: {
        if (bufa == stake) {
            /* the char to delete is already in TE */
            tagadjust(outputwind, TCUT, (*outputwind->hTE)->selStart, 1);
            TEKEY(BS, outputwind->hTE);
        }
        else {
            --bufa;
            /* just delete the sucker previous */
        }
        break;
    }
    case BEL: {
        beeper();
        break;
    }
    case TAB: {
        *bufa++ = bufchar;
        break;
    }
    default: {
        escapemode = outputwind->escapemode = FALSE;
        /* risk of bad sequence should be avoided */
    }
}
}
else if (escapemode) {
    if (eschandle(bufchar)) {
        escapemode = outputwind->escapemode = FALSE;
    }
    continue;
}
else
    /* move a regular character back to wait for TE */
    *bufa++ = bufchar;
}
/* now we process the text */
puttext(outputwind, stake, bufa - stake);
if (lineadded || literal) {
    DrawControls(outputwind->ptr);
    lineadded = FALSE;
}
#endif
#endif TEWINDSWITCH
if ( !uwon && (*outputwind->hTE)->teLength > TESWITCHLENGTH) {
    /* make a new output window next time around the event loop */
    tewindswitch = TRUE;
}
#endif
return(0);
}

#endif TEKEXISTS

```

```

#define ALPHA 1
#define PLOT 2
#define INC PLOT 3
#define TEKOFFSET 2

int tekmode = ALPHA; /* alpha v. graphic v. graphic in */
int tekpenup = FALSE; /* is the pen up? */
int tekescmode = FALSE;
Point macloc;
int xloc = 0;
int yloc = 0;
int hadlowy = FALSE; /* we've seen an 0x60 tag; if 0x20 comes, it's highx */
int didlowy = FALSE; /* we've seen an 0x60 and an 0x20 tag and have handled them */
int lastlowy = 0;
int plotcount = 0; /* make sure we see no more than 5 bytes before we get a hit */

tekoutput(thewind, count)
struct window * thewind;
int count;
{
    register char maskb;
    register unsigned char bufchar;
    register char * bufp;
    register char * bufend;
    GrafPtr saveport;

    GetPort(&saveport);
    SetPort(thewind->ptr);
    MoveTo(macloc.h, macloc.v);

    maskb = mask;
    bufend = stake + count;
    for (bufp = stake; bufp < bufend; bufp++) {
        bufchar = *bufp &= maskb;
#ifdef UW
        if (uwmode) {
            uwmode = FALSE;
            stake++;
            if (uhandle(bufchar)) {
                macloc.h = qd.thePort->pnLoc.h;
                macloc.v = qd.thePort->pnLoc.v;
                return(bufend - stake);
            }
            continue;
        }
#endif
        if (tekescmode) {
            /* handle tek escape sequence */
            tekescmode = FALSE;
            stake++;
            switch (bufchar) {
                case ENQ: {
                    /* return status and cursor position */
                    break;
                }
                case ETB: {
                    /* make hard copy */
                    break;
                }
                case SUB: {
                    /* start cross hair cursor */
                    break;
                }
                case FF: {
                    /* erase screen, home cursor, set alpha mode */
                    macloc.h = 2;
                    macloc.v = 13;
                    xloc = 0;
                    yloc = 0;
                    EraseRect(&qd.thePort->portRect);
                    tekmode = ALPHA;
                    break;
                }
            }
            continue;
        }
        if (bufchar < SP) {
            if (bufp != stake) {
                /* we process plain data not yet written out */
                /* this code is paralleled below */
                if (tekmode == PLOT) {
                    for ( ; stake < bufp; stake++) {
                        tek14plot((int) *stake);
                    }
                }
                else if (tekmode == ALPHA) {
                    /* shift up to draw things in the place the tek would */
                }
            }
        }
    }
}

```

```

/* Move(0, - qd.thePort->txSize);*/
DrawText(stake, 0, bufp - stake);
/* Move(0, qd.thePort->txSize); */
stake = bufp;
}
else if (tekmode == INC PLOT) {
    int macx, macy;

    for ( ; stake < bufp; stake++) {
        tek14inc(*stake);
    }
    tek14macscale(xloc, yloc, &macx, &macy);
    MoveTo(macx, macy);
}
*/
/* ASCII control codes */
switch (bufchar) {
    case NUL: {
        /* do nothing */
        break;
    }

#ifdef UW
    case SOH: {
        /* next byte is a UW command */
        uwmode = TRUE;
        break;
    }

#endif
    case VT: {
        Move(6, 0);
        break;
    }
    case CR: {
        Move(- qd.thePort->pLoc.h + TEKOFFSET, 0);
        break;
    }
    case LF: {
        Move(0, 11);
        if (checksave())
            return(bufend - stake);
        break;
    }
    case BS: {
        Move(-6, 0);
        break;
    }
    case BEL: {
        beeper();
        break;
    }
    case TAB: {
        Move(6, 0);
        break;
    }
    case SUB:
    case CAN: {
        /* cancel current escape sequence */
        escapemode = outputwind->escapemode = FALSE;
        break;
    }
    case GS: {
        /* go into graphics mode and lift pen */
        tekpenup = TRUE;
        tekmode = PLOT;
        break;
    }
    case US: {
        /* alpha mode */
        tekmode = ALPHA;
        break;
    }
    case RS: {
        /* incremental plotting mode */
        tekmode = INC PLOT;
        break;
    }
    case ESC: {
        tekescprep();
        break;
    }
    case XOFF: {
        hostxoff = TRUE;
        break;
    }
    case XON: {
        hostxoff = FALSE;
        break;
    }
}

```

```

        }
    stake++;
}
else if (escapemode) {
    if (eschandle(bufchar))
        escapemode = outputwind->escapemode = FALSE;
    stake++;
    continue;
}
/* end if < SPACE; else do nothing but increment bufp in for */
}
if (bufp > stake) {
    /* we process plain data not yet written out */
    /* this parallels code above */
    if (tekmode == PLOT) {
        for ( ; stake < bufp; stake++) {
            tek14plot((int)*stake);
        }
    }
    else if (tekmode == ALPHA) {
        /* Move(0, - qd.thePort->txSize); */
        DrawText(stake, 0, bufp - stake);
        /* Move(0, qd.thePort->txSize); */
    }
    else if (tekmode == INC PLOT) {
        int macx, macy;

        for ( ; stake < bufp; stake++) {
            tek14inc(*stake);
        }
        tek14macscale(xloc, yloc, &macx, &macy);
        MoveTo(macx, macy);
    }
}
macloc.h = qd.thePort->pnlLoc.h;
macloc.v = qd.thePort->pnlLoc.v;
SetPort(saveport);
return(0);
}

tekescprep()
{
    tekescmode = TRUE;
}

tek14plot(chunk)
register unsigned int chunk;
{
    register int cmd;

    if (++plotcount > 5)
        /* we've been misled! */
        errprompt("bad plot vector");
    cmd = chunk & 0x0060;
    chunk &= 0x001f;
    /* mask off the command segment */

    /* we've got a byte, interpret it */
    if (cmd == 0x20) {
        if (!hadlowy) {
            /* high y coordinate */
            yloc &= 0xf07f;
            yloc |= chunk << 7;
            return(0);
        }
        else {
            /* high x coordinate always preceded by lowy */
            xloc &= 0xf07f;
            xloc |= chunk << 7;
            return(0);
        }
    }
    else if (cmd == 0x60) {
        /* either low y or an extra byte */
        if (hadlowy) {
            yloc &= 0xff83;
            yloc |= chunk << 2;
            /* do lowy */
            yloc &= 0xffffc;
            yloc |= (lastlowy & 0x0c) >> 2;
            /* y piece of extra */
            xloc &= 0xffffc;
            xloc |= (lastlowy & 0x03);
            /* x piece of extra */
            didlowy = TRUE;
        }
    }
}

```

```

        return(0);
    }
    else {
        lastlowy = chunk;
        hadlowy = TRUE;
        return(0);
    }
}
else if (cmd == 0x40) {
    int macx;
    int macy;

    /* we're finished */
    if (hadlowy & !didlowy) {
        /* there's a lowy waiting */
        yloc &= 0xff83;
        yloc |= lastlowy << 2;
    }
    xloc &= 0xff83;
    xloc |= chunk << 2;

    /* scale the points */
    tek14macscale(xloc, yloc, &macx, &macy);

#if TEKDEBUG
sprintf(error, "%d %d to: %d %d %s", xloc, yloc, macx, macy, tekpenup ? "up" : "");
errprompt(error);
#endif

    if (tekpenup) {
        MoveTo(macx, macy);
        tekpenup = FALSE;
    }
    else
        LineTo(macx, macy);

    hadlowy = FALSE;
    didlowy = FALSE;
    plotcount = 0;
    return(1);
}
}

/* scale a tek 4014 point to a macintosh point */

long yscale = 14L; /* TODO change with window size; == 4096 / ybits */
long ybits = 300L;
long xscale = 8L;
long xoffset = TEKOFFSET;
long yoffset = TEKOFFSET;

tek14macscale(tekx, teky, nxptr, nyptr)
unsigned int tekx;
unsigned int teky;
register unsigned int * nxptr;
register unsigned int * nyptr;
{
    *nxptr = (tekx / xscale) + xoffset;
    *nyptr = - ((long) teky / yscale) + ybits + yoffset;
    /* correct y so it will go into window rather than above it */
}

/* do incremental motion--NOTE: it makes increments big so they have
   some effect on a Mac! */

tek14inc(amt)
unsigned char amt;
{
    if (amt & 0x40)
        amt &= 0x40;
    else
        return;
    if (amt & 0x01) {
        /* right */
        xloc += xscale;
    }
    if (amt & 0x02) {
        /* left */
        xloc -= xscale;
    }
    if (amt & 0x04) {
        /* up */
        yloc += yscale;
    }
    if (amt & 0x08) {
        /* down */
        yloc -= yscale;
    }
}

```

```

        }

}

#endif

#ifndef SESCTL
sessionctl(thechar)
register char thechar;
{
    switch (thechar) {
#endif UW
    case SOH: {
        /* next byte may be a UW command */
        if (uwon) {
            uwmode = TRUE;
            return(TRUE);
        }
    }
#endif GS
    case GS: {
        /* go into graphics mode */
        if (tekwind != NULL) {
            outputwind = tekwind;
            return(TRUE);
        }
    }
    case ESC: {
        escprep(); /* TODO parameterize */
        return(TRUE);
    }
    case XOFF: {
        hostxoff = TRUE;
        return(TRUE);
    }
    case XON: {
        hostxoff = FALSE;
        return(TRUE);
    }
}
if (escapemode) {
    if (eschandle(thechar)) {
        escapemode = outputwind->escapemode = FALSE;
    }
    return(TRUE);
}
#endif

endcapture(thewind)
struct window * thewind;
{
    long fpos;
    extern int readlag;

    thewind->type = capwindtype;
    beeper(5);
    longtoa(error, ncaptured);
    errstr(error);
    errstr(" bytes captured\r");
    /* should display # of bytes */
    GetFPos(fdcapture, &fpos);
    SetEOF(fdcapture, fpos);
    FSClose(fdcapture);
    FlushVolC (StringPtr) NULL, capvol);
    fcapture = FALSE;
    ncaptured = 0;
}

checksave()
{
    if (saveonlf) {
        fcapture = TRUE;
        saveonlf = FALSE;
        capturewind = outputwind;
        capwindtype = outputwind->type;
        outputwind->type = DOWNWIND;
        return(TRUE);
    }
    else
        return(FALSE);
}

counttoCR(charp)
register char * charp;
{
    register int count;

```

11/16/94 12:31

comm.c

11

```
for (count = 0; *--charp != LF && *charp != CR; count++)
;
return(count);
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "comm.h"
#include "commdefs.h"
#include "commnnodefs.h"
#include "editnnodefs.h"
#include "windnnodefs.h"

char * noserwrite = "no serial write\r";
char * serconf = "Serial conf. failed\r";
char * textfull = "Text window full";

char linefeed = LF;
char cr = CR;

char pagewait = FALSE;
int linespacing = 1;

char useTEKey = TRUE;      /* use TEKey delete-selection behavior */
char autoindent = TRUE;    /* do autoindent */

extern char * ramout;

extern char dofflow;
extern char useselbottom;

portinit(portnum, portp)
int portnum;
struct Port * portp;
{
    int config;
    int result;

#define TRYSERD
    noserd = FALSE;
    /* now open it */
    if ((result = RAMSDOpen(portnum == MODEMPORT ? sPortA : sPortB) ) ) {
        /* open failed */
        noserd = TRUE;
        errstr("no RAM driver\r");
        if (result == portInUse) {
            errstr("port in use\r");
            return(-1);
        }
        if (result == portNotCf) {
            errstr("port not configured\r");
            return(-1);
        }
        OpenDriver(portnum == MODEMPORT ? "\P.AOut" : "\P.BOut", &portp->out);
        OpenDriver(portnum == MODEMPORT ? "\P.AIn" : "\P.BIn", &portp->in);
    }
    else {
        /* initialize i/o port only when good driver present */
        /* Both SerHShake & Control(... ) kill the Mac with orig. ROM driver! */
        portp->in = portnum == MODEMPORT ? -6 : -8;
        portp->out = portnum == MODEMPORT ? -7 : -9;
        if (portp == &modem) {
            setshake(portp->out, FALSE, TRUE, TRUE);
        }
        else {
            setshake(portp->out, TRUE, FALSE, FALSE);
        }
    }
}

#else
    noserd = TRUE;
    OpenDriver(portnum == MODEMPORT ? "\P.AOut" : "\P.BOut", &portp->out);
    OpenDriver(portnum == MODEMPORT ? "\P.AIn" : "\P.BIn", &portp->in);
#endif

config = portp->speed | portp->data | portp->stop | portp->parity;
if (SerReset(portp->in, config) || SerReset(portp->out, config)) {
    errstr(serconf);
}
if (portp == &modem) {
    char leaveDTRup;

    /* set up buffers for the modem */
    if (readbuf == NULL) {
        /* allocate memory for the read buffer */
        ResvMem((long) READBUFSIZE);      /* long ex SUMACC */
        if ( !(readbuf = NewPtr((long) READBUFSIZE)) ) {

```

```

        errprompt(ramout);
        return -1;
    }
}

if (sdbuf == NULL) {
    /* allocate the serial driver buffer */
    ResrvMem((long) SERBUFSIZE);
    if ((sdbuf = NewPtr((long) SERBUFSIZE)) != NULL)
        SerSetBuf(portp->in, sdbuf, SERBUFSIZE - 2); /* last param short ex SUMACC */
        /* yup, that "- 2" is supposed to be there! */
}
/* make control call so Mac + will leave its DTR line asserted when you close driver */
leaveDTRup = 0x40; /* mask bit 7 down for DTR */
Control(portp->out, 18, &leaveDTRup);
Control(portp->out, 17, &leaveDTRup);
}

return(0);
}

/* set the handshake options on the serial driver; */
/* SerHShake is also implicated in the hideous looping death of the
   machine when set TF XON XOFF FFFF */

#ifndef TRYSERD
char noxoff[] = "no XON/XOFF\r";
#endif

setshake(out, outhardshake, inshake, outshake)
int out, outhardshake, inshake, outshake;
{
    SerShk serShk;

    serShk.fXOn = (Byte) outshake; /* output XON-XOn/off */
    serShk.fCTS = (Byte) outhardshake; /* CTS on for output--contrary to Inside Mac, FALSE enables CTS */
    serShk.xOn = (char) XON; /* xon char DC1 */
    serShk.xOff = (char) XOFF; /* xoff char DC3 */
    serShk.errs = (Byte) 0; /* input abort settings ALL = 0160 */
    serShk.evts = (Byte) 0; /* status junk */
    serShk.fInX = (Byte) inshake; /* input XON-XOFF on/off */
    serShk.fDTR = (Byte) FALSE; /* input DTR handshake */

    if (Control(out, romtype & ROM64 ? 10 : 14, &serShk))
        errstr(noxoff);
        /* "Advanced" control call, dups SerHShake adding DTR input as 8th byte */
}

#endif

/* put a character out the given serial port, making suitable conversions,
   used primarily for modem output */
/* WARNING emacs users may choke on the ^C input flushing! */

portWrite(portp, thechar)
struct Port * portp;
char thechar;
{
    long count = 1;

    if (!portp->out)
        return(-1);

    if (laserprint && portp == &printer) {
        if (thechar == FF) {
            /* simulate a form feed */
            PrCtlCall(iPrDevCtl, (long) lPrPageClose, (long) 0, (long) 0);
            if (PrError())
                errprompt("Can't close printer page");
            }
            PrCtlCall(iPrDevCtl, (long) lPrPageOpen, (long) 0, (long) 0);
            if (PrError())
                errprompt("Can't open printer page");
            }
            header();
        }
        else if (thechar == CR) {
            PrCtlCall(iPrDevCtl, (long) lPrLFSixth, (long) 0, (long) 0);
            if (PrError())
                errprompt("Can't print CR-LF");
            }
        }
        else if (thechar == LF) {
            /* skip it */
            ;
        }
        else {
            /* send the character */
            PrCtlCall(iPrIOCtl, (long) &thechar, count, (long) 0);
        }
}

```

```

        if (PrError()) {
            errprompt("Can't print character");
        }
    }

    return(0);
}

/* fix char conversions that apply globally to output */
if (portp == &modem && switchnl && thechar == CR)
    thechar = LF;

if (FSWrite(portp->out, &count, &thechar))
    errprompt(noserwrite);
if (portp == &modem && thechar == ETX) {
    /* control-C, flush input */
    long tickcount;

    /* wait for input to finish before flushing */
    Delay((long) 6, &tickcount);
    KillIO(portp->in);
}
}

/* write a string to end of the Scratchpad window--standard error output */

errstr(errs)
char * errs;
{
    int notfront;

    notfront = FALSE;
    if (bugwind == NULL)
        return(-1);
    if (bugwind->ptr != FrontWindow() ) {
        selwind(bugwind->ptr);
        closeclip(bugwind);
        notfront = TRUE;
    }
    tesetsel((long) (*bugwind->hTE)->teLength, (long) (*bugwind->hTE)->teLength, bugwind->hTE);
    puttext(bugwind, errs, strlen(errs));
    if (notfront)
        openclip();
    else
        DrawControls(bugwind->ptr);
}

/* interpret a character, stick it into the input window if
it's on, and write it to serial port if it's on.
CR key gets special handling. */

handlekey(thechar, backspace)
char thechar;
int backspace; /* flag for do backspace */
{
    register char * TEPoS;
    register char * TEbase;
    TEHandle texthand;
    char outchar;
    int count = 0;
    WindowPtr frontwind;

    frontwind = FrontWindow();
    if (((WindowPeek) frontwind)->windowKind) < userKind)
        return;

    /* fix the output character according to current conversion settings */
    outchar = thechar;
    if (bstodel && thechar == BS)
        outchar = DEL;
    if (decesc && thechar == '`') {
        outchar = ESC;
    }

    switch (swindptr(frontwind)->type) {
    /* TOTEST handlekey--was "outputwind" dangerous w/ uw should use front? */
    case BUGWIND:
    case TEXTWIND: {
        texthand = inputwind->hTE;
        if (!logmode && !local) {
            /* just write out char and return */
            if (modem.transon) {
                beeper();
                return();
            }
        }
    }
}

```

```

charout(outchar);
return;
}

if (backspace) {
    /* TEInsert BS doesn't work; to avoid selection eating behavior,
       we simulate TEKey effect */
    int selstart, selend;

    selstart = (*texthand)->selStart;
    selend = (*texthand)->selEnd;
    if ((*texthand)->selStart == 0)
        TEKey(BS, texthand);
    /* fix bad cursor placement */
    else if (useTEKey && (selstart != selend)) {
        /* do Apple backspace-clears behavior */
        editmenu(CUT);
    }
    else if (enoughmem(50)) {
        /* clear the character before the selection & reset selection */
        tesetsel((long) selstart - 1, (long) selstart, texthand);
        TEDelete(texthand);
        --selstart;
        --selend;
        tagadjust(inputwind, TCUT, selstart, 1);
        seekselect(inputwind);
        tesetsel((long) selstart, (long) selend, texthand);
        SetCtlMax(inputwind->vscroll, newvctlmax(inputwind));
    }
}
else {
    /* put text in the input window */
    puttext(inputwind, &thechar, 1);
    if (thechar == CR) {
        if (autoindent && !useselbottom) {
            HLock((*texthand)->hText);
            TEbase = *((*texthand)->hText);
            TEpso = TEbase + (*texthand)->selStart - 1;
            /* place TEpso at CR*/

            while (--TEpos >= TEbase) {
                /* find beginning of current line */
                if (*TEpos == CR)
                    break;
            }
            TEpos++;
            thechar = TAB;
            while (*TEpos++ == thechar)
                count++;
            count++;
            HUnlock((*texthand)->hText);
            while (--count)
                puttext(inputwind, &thechar, 1);

            DrawControls(inputwind->ptr);
        }
    }
}

/* We are in Local edit or Record mode. This stuff must come
 * after puttext: CR has to be there to send! Don't send local
 * with CR.
 */
if (thechar == CR && !local) {
    /* We are in Record mode & have a CR: send the line! */
    if (modem.transon) {
        /* don't send when transon */
        beeper();
        return(FALSE);
    }

    HLock((*texthand)->hText);
    TEbase = *((*texthand)->hText);

    TEpso = TEbase + (*texthand)->selStart - 1;
    /* place TEpso at CR*/

    while (--TEpos >= TEbase) {
        /* find beginning of current line */
        if (*TEpos == CR)
            break;
    }
    TEpos++;
    if (skipprompt) {
        while (*TEpos != CR && *TEpos != SP)
            /* put pos after prompt and spaces */

```

```

        TEpos++;
        while (*TEpos == SP)
            /* eat blanks */
            TEpos++;
    }
    while(*TEpos != CR) {
        charout(*TEpos++);
    }
    charout(CR);
    HUnlock( (*texthand)->hText );
}
break;
}
case VIWIND: {
    if (modem.transon) {
        /* don't send when transon */
        beeper();
        return(FALSE);
    }
    charout(outchar);
    break;
}
#endif TEKEISTS
case TEKWIND: {
    if (modem.transon) {
        /* don't send when transon */
        beeper();
        return(FALSE);
    }
    charout(outchar);
    break;
}
#endif
}

char litinon = TRUE;
char memfull = FALSE;      /* set if MemError after TEInsert */

/* write count chars from a ptr into a textedit window; standard output
   routine, handles scrolling */

puttext(thewind, buf, count)
struct window * thewind;
register char * buf;
int count;
{
    TEHandle texthand;
    extern struct window * makewindow();
    int newmax;
    extern long anchor;

    if (count <= 0)
        return;

    texthand = thewind->hTE;

    /* don't allow addition of text past TE's max size */

    if ( (*texthand)->teLength + count < 0) {
        /* changed to < 0 from > TEMAXLENGTH because failed test ! */
        textfull[12] = (thewind - window) + '0';
        /* stick a number in the prompt */
        errprompt(textfull);
        return(-1);
    }
    if (!memfull) {
        if (enoughmem(count + 1024)) {
            if (count == 1 && useTEKey
                && (thewind->cstate & WS_LOCAL || thewind->cstate & WS_RECORD) ) {
                /* do a CUTINS before inserting the character to simulate TEKey */
                editmenu(CUTINS);
            }
            TEInsert(buf, (long) count, texthand);
            tagadjust(thewind, TADD, (*texthand)->selStart, count);
        }
        else {
            memfull = TRUE;
        }
    }
    else
        beeper();

#endif WINDEBUG
    if (thewind != bugwind) {
        /* if you want to put stuff out to check window functions

```

```

        * do it in here to avoid recursion!
        */
        errstr("in put\r");
    }
#endif
/*thewind->vscroll)->contrlMax = newvctlmax(thewind);

/* if near page--i.e. viewRect--bottom,
 * do scroll if scroll is turned on
 */
if (thewind->scroll) {
    register int diff;

    if (count == 1) {
#ifndef ANCHORHIGH
        if (anchor != -1)
            tesetsel((*texthand)->selStart, (*texthand)->selEnd, texthand);
#endif
        seekselect(inputwind);
    }
    else {
        /* find how far below window bottom 'tis & scroll to it */
        diff = seltop(texthand)
            + (*texthand)->lineHeight
            - (*texthand)->viewRect.bottom;
        if (diff > 0 ) {
            TEScroll(0, -diff, texthand);
            (*thewind->vscroll)->contrlValue += diff;
            /* linecounting--( (diff - 1) / (*texthand)->lineHeight ) + 1;*/
        }
    }
}

/* close the serial port named by the arguments */

portclose(portp)
struct Port * portp;
{
    if (portp->in != 0 && portp == &modem) {
        if (!noserd)
            RAMSDClose(portp->in == -6 ? sPortA : sPortB);
        if (readbuf != NULL)
            DisposPtr(readbuf);
        if (sdbuf != NULL)
            DisposPtr(sdbuf);
        readbuf = sdbuf = NULL;
    }
    portp->in = portp->out = 0;
}

/* WARNING hideous loop bug afflicting sendselect under "certain
circumstances"--associated with SerHShake call, which causes
eventual death through serial driver arrhythmia.
*/
#define PAGESIZE 54

int wrapwidth = 72;           /* wrap at/before this column */
int linecount = 0;
int pagecount = 1;

/* prepare to send text out a serial port using sendline(),
which wraps and adds cr-lf as appropriate, and calls portline() */

porttext(portp, texthand, start, end, print)
struct Port * portp;
TEHandle texthand;
short start, end;
int print;
{
    long length;

    if (!portp->out)
        return(-1);

    if (portp->transon) {
        beeper();
        return(FALSE);
    }

    length = end - start;
}

```

```

if ( (portp->transtxhand = NewHandle(length)) != NULL) {
    /* make a copy of the text so no corruption possible */

    hilock(portp->transtxhand);
    portp->printstart = *(portp->transtxhand);
    portp->printend = portp->printstart + length;
    HLock((**texthand)->hText);
    BlockMove((**texthand)->hText) + start, portp->printstart, length);
    HUnlock((**texthand)->hText);
    portp->copied = TRUE;
}
else {
    /* out of space, lock down the Handle for awhile and run a risk */

    errstr("don't alter print text\r");
    hilock((**texthand)->hText);
    portp->printstart = (**texthand)->hText + start;
    portp->printend = (**texthand)->hText + end;
    portp->transtxhand = (**texthand)->hText;
    portp->copied = FALSE;
}
portp->transprint = print;
shipflag(portp, TRUE);
sendline(portp);
}

/* send selected text or text from caret to CR
   from the input window out the serial port */

sendselect()
{
    register char * TEpso, * TEbase, * TEmax, * selend;
    char * startpos;
    TEHandle texthand;
    TERec * textptr;

    if (!modem.out)
        return(-1);

    if (modem.transon) {
        beeper();
        return(FALSE);
    }

    texthand = inputwind->hTE;
    HLock((Handle) texthand);
    textptr = *texthand;

    if ( textptr->selStart == textptr->selEnd ) {
        long lcount;

        /* if caret, send text to end of line, synchronously & without wrapping */
#ifdef USEASYNCWRITEOFTHIS
        TEbase = *textptr->hText;
        TEmax = TEbase + textptr->telength;
        for (TEpos = startpos = TEbase + textptr->selStart;
             *TEpos++ != CR && TEpos < TEmax; )
            ;
        porttext(&modem, texthand, textptr->selStart, TEpos - TEbase, FALSE);
#else
        HLock(textptr->hText);
        TEbase = *textptr->hText;
        TEmax = TEbase + textptr->telength;
        for (TEpos = startpos = TEbase + textptr->selStart;
             *TEpos != CR && TEpos < TEmax; TEpos++)
            ;
        lcount = TEpos - startpos;
        FSWrite(modem.out, &lcount, startpos);
        lcount = 1;
        FSWrite(modem.out, &lcount, switchnl ? &linefeed : &cr);
        /* handle switchnl mode properly */
        HUnlock(textptr->hText);
        TEpos++;
#endif
        tesetsel( (long) (TEpos - TEbase), (long) (TEpos - TEbase),
                  texthand);
        /* end of line for next line */
        seekselect(inputwind);
    }
    else {
        /* if selection, send selection only and set to caret at selend */
        porttext(&modem, texthand, textptr->selStart, textptr->selEnd, FALSE);
    }
    HUnlock((Handle) texthand);
    return;
}

```

```

#ifndef ASYNCWRITE

/* send a line out the specified port, adding cr & lf */

portline(portp, count, buf, addCR, addLF)
struct Port * portp;
int count;
char * buf;
int addCR, addLF;
{
    long lcount;           /* FSWrite requires a * long */
    static long lagtill;
    extern int writelag;

    lcount = count;
    if (laserprint) {
        PrCtlCall(iPrIOCtl, buf, lcount, (long) 0);
        if (addCR) {
            /* TODO for asyncwrite ? */
            ;
        }
        return(0);
    }

    /* to substitute for output xon/xoff, wait a user-specified while */
    while (lagtill > TickCount() )
    ;
    lagtill = TickCount() + writelag;

    if (count && FSWrite(portp->out, &lcount, buf))
        errprompt(noserwrite);
    if (addCR) {
        lcount = 1;
        FSWrite(portp->out, &lcount,
            (portp == (struct Port *) &modem && switchnl) ? &linefeed : &cr);
        /* handle switchnl mode properly */
    }
    if (addLF) {
        lcount = 1;
        FSWrite(portp->out, &lcount, &linefeed);
    }
}

#endif

/* set up another line to be sent off from a locked text; calculate line
break, ship it off, update pointers for next line */

sendline(portp)
struct Port * portp;
{
    register char * selptr, * selend, * selstart;
    register char thechar;
    int charcount;
    int column;

    if (noserd && dofflow && hostxoff)
        return(FALSE);

    column = 0;
    selptr = selstart = portp->printstart;
    selend = portp->printend;

    for (; selptr < selend; selptr++) {
        /* do nothing unless CR or wrap required */
        if ((thechar = *selptr) == CR) {
            ;
        }
        else if (wrapwidth && column > wrapwidth) {
            /* wrap the text, making sure to leave spaces at the end of this line */
            while ( !((thechar = *selptr) == SP) || (thechar == TAB) || (thechar == '-') || (thechar == '/') ) {
                /* we're on a non-breaking character, go back to white space */
                if (--selptr <= selstart) {
                    /* don't go past last line sent */
                    selptr += wrapwidth;
                    thechar = *selptr;
                    break;
                }
            }
            if (thechar == '-') {
                if (*(selptr + 1) == '-')
                    thechar = *++selptr;
            }
        }
        else if (thechar != '/') {
            /* we're on white space, keep going till we run out */
            while ( ((thechar = *selptr) == SP) || (thechar == TAB) ) {

```

```

        if (selptr < selend)
            selptr++;
        else
            break;
    }
    thechar = *--selptr;
}
else {
    /* keep on looking for a CR or wrap condition */
    if (thechar == TAB && !laserprint)
        column += ((column + 8) & ~7) - column;
    /* TODO should have tabsize var */
    else
        column++;
    continue;
}
if (portp->transprint) {
    if (++linecount > PAGESIZE) {
        pagecount++;
        portWrite(portp, FF);
        linecount = 1; /* count current line on next page! */
        if (pagewait) {
            char trash[1];
            prompt("Insert new page and press CR");
            getline(trash, 1);
        }
    }
    if (!laserprint && *selstart == FF) {
        /* we will detect FF and compensate only if it's at the beginning of a line */
        linecount = 1;
        pagecount++;
    }
}
column = 0;
charcount = selptr - selstart + 1;
if (thechar == CR)
    /* skip the CR, portline handles it */
    --charcount;
portline(portp, charcount, selstart, TRUE, portp->transprint ? TRUE : FALSE);
portp->printstart = selptr + 1;
return(FALSE);
}
if (selptr >= selend) {
    /* line left hanging w/o wrap or CR */
    charcount = selptr - selstart;
    if (portp->transprint) {
        linecount++;
        portline(portp, charcount, selstart, TRUE, TRUE);
    }
    else
        portline(portp, charcount, selstart, FALSE, FALSE);
}
#endif ASYNCWRITE
portp->transfin = TRUE;
#else
/* we are finished sending the text */
HUnlock(portp->transtxhand);
if (portp->copied)
    DisposHandle(portp->transtxhand);
shipflag(portp, FALSE);
return(TRUE);
#endif
}

transcancel(portp)
struct Port * portp;
{
    if (!portp->transon)
        return(0);
    KillIO(portp->out);
    shipflag(portp, FALSE);
    HUnlock(portp->transtxhand);
    if (portp->copied)
        DisposHandle(portp->transtxhand);
    if (portp == &printer)
        portWrite(portp, CAN);
}

/* reset port.transon and little P & S flags at top of screen */

shipflag(portp, on)
struct Port * portp;
int on;

```

```

{
    int fontnum;

    if (on == portp->transon)
        return(0);

    portp->transon = on;
    drawflag(portp);
}

drawflag(portp)
struct Port * portp;
{
    getwport(0);

    if (!portp->transon)
        TextMode(srcXor);
    if (portp == &printer) {
        MoveTo(4, 9);
        DrawChar('P');
    }
    else {
        MoveTo(4, 18);
        DrawChar('S');
    }
    if (!portp->transon)
        TextMode(srcOr);

    retwport();
}

#endif ASYNCWRITE

#define BTRUE 0x0100

ParamBlockRec prPB;
ParamBlockRec modPB;

pascal void
prline()
{
    savea5();
    printer.transcont = TRUE;
    restorea5();
}

pascal void
modline()
{
    savea5();
    modem.transcont = TRUE;
    restorea5();
}

portline(portp, count, buf, addCR, addLF)
struct Port * portp;
int count;
char * buf;
int addCR, addLF;
{
    long lcount; /* FSWrite requires a * long */
    static long lagtill;
    ParamBlockRec * PBptr;

    lcount = count;
    if (laserprint && portp == &printer) {
        if (lcount) {
            PrCtlCall(iPrIOCtl, (long) buf, lcount, (long) 0);
            if (PrError())
                errprompt("Can't print characters");
        }
        portp->addcr = addCR;
        portp->transcont = TRUE;
    }
    else {
        portp->addcr = addCR;
        portp->addlf = addLF;
        /* for execution after endline iocompletion function sets flag */
        if (count) {
            if (portp == &modem) {
                modPB.ioParam.ioCompletion = (ProcPtr) modline;
                PBptr = &modPB;
            }
            else {

```

```

    prPB.ioParam.ioCompletion = (ProcPtr) prline;
    PBptr = &prPB;
}
PBptr->ioParam.ioRefNum = portp->out;
PBptr->ioParam.ioBuffer = buf;
PBptr->ioParam.ioReqCount = lcount;
PBptr->ioParam.ioPosMode = fsAtMark;
PBptr->ioParam.ioPosOffset = 0L;

if (PBWrite(PBptr, BTRUE))
    errprompt(noserwrite);
else
    portp->transcont = FALSE;
if (count < 3 || ((portp->speed == baud57600)))
    /* kludge to make sure transcont gets reset for those small numbers,
       which seem to fail to get an iocompletion call */
    portp->transcont = TRUE;
}
else
    portp->transcont = TRUE;
}

endline(portp)
struct Port * portp;
{
    if (noserd && doflow && hostxoff)
        return(FALSE);

    if (portp == &printer) {
        register count;

        for (count = linespacing; --count > 0;) {
            portWrite(portp, CR);
            portWrite(portp, LF);
            linecount++;
        }
    }
    if (portp->addrCR) {
        portp->addrCR = FALSE;
        portWrite(portp, CR);
    }
    if (portp->addrLF) {
        portp->addrLF = FALSE;
        portWrite(portp, LF);
    }
    if (portp->transfin == TRUE) {
        /* we are finished sending the text */
        portp->transfin = FALSE;
        HUnlock(portp->transtxhand);
        if (portp->copied)
            DisposHandle(portp->transtxhand);
        shipflag(portp, FALSE);
        if (laserprint) {
            PrCtlCall(iPrDevCtl, (long) lPrPageClose, (long) 0, (long) 0);
            if (PrError())
                errprompt("Can't close printer page");
            }
            PrCtlCall(iPrDevCtl, (long) lPrDocClose, (long) 0, (long) 0);
            if (PrError())
                errprompt("Can't close printer document");
            }
        }
        return(FALSE);
    }
    return(TRUE);
}

#endif

savedA5()
{
#asm
OrigA5 EQU $904
    move.l  a5,a3
    move.l  (OrigA5),A5
#endasm
}

restoreA5()
{
#asm
    move.l  a3,a5
#endasm
}

charout(thechar)

```

```
register char thechar;
{
    portWrite(&modem, thechar);
}

/*
strout(thestr)
register char * thestr;
{
    long count;

    count = strlen(thestr);
    if (FSWrite(modem->out, &count, &thestr))
        errprompt(noserwrite);
}
*/

char blankline[] = "                ";

header()
{
    int count;
    int spacecount;
    char pagestr[80];

    count = strlen(headerstr);
    longtoa(pagestr, (long) pagecount);
    spacecount = wrapwidth - count;

    prcrlf(2);
    portline(&printer, count, headerstr, FALSE, FALSE);
    if ((spacecount -= strlen(pagestr)) <= 0)
        count = 1;
    else
        count = spacecount;

    portline(&printer, count, blankline, FALSE, FALSE);
    count = strlen(pagestr);
    portline(&printer, count, pagestr, FALSE, FALSE);
    linecount++;

    prcrlf(3);
}

prcrlf(count)
int count;
{
    while (count--) {
        portWrite(&printer, CR);
        if (!laserprint)
            portWrite(&printer, CR);
        linecount++;
    }
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "comm.h"
#include "videfs.h"
#include "menudefs.h"
#include "echomenudefs.h"
#include "uw.h"

/* contains the escape sequence interpreter--a subset of ansi x3.64--and
the intelligent window driver closely related to it */

/* viwind vars */

struct virecord vr;

/* general variables */

Rect virect;           /* current clip rect for vi */

char termok[] = { '\033', '[', '0', 'n' };
char viblink = TRUE;    /* blink cursor or not? */
long * carettime = (long *) 0x2f4; /* WARNING low memory caret blink constant */

char bitscroll = TRUE; /* use QD ScrollRect rather than redraw on scroll */
int count;              /* general purpose counter */

/* char wasswitchnl = FALSE;    saves old switchnl state ? */

/* comm modes */
extern char switchnl;      /* if TRUE CR=LF && LF=CR */
extern char uwmode;         /* TRUE if SOH rec'd */
extern char escapemode; /* if TRUE send non-control chars to interpreter */

extern RgnHandle updateRgn; /* scrolling update region */

/* Interpret chars in escape sequence;
return TRUE when escape finished or invalid */

eschandle(thechar)
char thechar;
{
    if (outputwind != viwind) {
        /* copy the globals in: minimizes pointer mush in code */
        vicontext(outputwind);
    }
    if (vr.foundpound)
        /* ignore parameter of a DEC double-height/width command */
        return(TRUE);
    if (vr.foundparen) {
        /* ignore parameter of a DEC undocumented command */
        /* 'B' is normal font, '0' is graphics font */
        if (thechar == 'B') {
            /* normal font */
            ;
        }
        else if (thechar == '0') {
            /* graphics font */
            ;
        }
        return(TRUE);
    }

    if (!vr.foundbrack) {
        if (thechar == '[') {
            vr.foundbrack = TRUE;

            /* initialize vars for interpretation */
            vr.arg1 = 0;
            vr.arg2 = 0;
            vr.argptr = &vr.arg1;
            return(FALSE);
        }
        else {
            /* interpret the characters coming after an escape w/o a bracket */
            /* unimplemented vt100 commands
               F   special graphics chars
               G   ASCII chars
               H   home (vt100 uses as tab stop set)
               J   erase end of screen
               K   erase end of line
               Ylc cursor pos
               '   auto print on
               -   auto print off
            */
        }
    }
}

```

```

]  print screen
V  print line
W  print controller on/V off
*/
switch (thechar) {
    case '#': {
        /* this is a DEC double height/width command--ignore the rest */
        /*
            3  current line is top half of double line
            4  current line is bottom half
            both lines must have same contents
            5  current line single width (single height)
            6  current line double width (single height)
            7  hard copy of screen
        */
        vr.foundpound = TRUE;
        return(FALSE);
    }
    case '(': {
        /* this is a DEC undocumented command--ignore the rest */
        /* should detect ')' also */
        vr.foundparen = TRUE;
        return(FALSE);
    }

#endif GO
    case '1': {
        /* graphics processor on */
        break;
    }
    case '2': {
        /* graphics processor off */
        break;
    }
#endif
    case '7': {
        /* save cursor */
        vr.oldx = vr.xpos;
        vr.oldy = vr.ypos;
        vr.oldvip = vr.vitextp;
        break;
    }
    case '8': {
        /* restore cursor */
        vr.xpos = vr.oldx ;
        vr.ypos = vr.oldy;
        vr.vitextp = vr.oldvip;
        updcursor();
        break;
    }
    case '=': {
        /* keypad application mode -- send escape sequences */
        break;
    }
    case '>': {
        /* keypad numeric mode -- send numbers */
        break;
    }
    case '<': {
        /* go into ANSI mode from vt52 mode */
        break;
    }
    case 'D': {
        /* index, from vt52 */
        movedown();
        updcursor();
        break;
    }
    case 'H': {
        /* set tab at current pos */
        break;
    }
    case 'M': {
        /* reverse index, from vt52 */
        /* TODO exact behavior w/ scrolling regions ? */
        if (vr.ypos > vr.virowmin) {
            --vr.ypos;
            vr.vitextp -= vr.vicols;
            updcursor();
        }
        else {
            if (bitscroll) {
                viscrdown();
            }
            else {
                insertlins(vr.virowmin, 1);
            }
        }
    }
}

```

```

        }
        break;
    }
    case 'Z': {
        /* identify yourself! */
        weare();
        break;
    }
}
/* either finished, or not escape sequence */
return(TRUE);
}

if (thechar == '?') {
    if (vr.foundqmark) {
        /* can't be valid sequence */
        return(TRUE);
    }
    else {
        vr.foundqmark = TRUE;
        return(FALSE);
    }
}
else if (thechar == '!') {
    if (vr.foundbang) {
        /* can't be valid sequence */
        return(TRUE);
    }
    else {
        vr.foundbang = TRUE;
        return(FALSE);
    }
}
switch (thechar) {
    /* non-terminal characters cause return FALSE */
    case '0':
    case '1':
    case '2':
    case '3':
    case '4':
    case '5':
    case '6':
    case '7':
    case '8':
    case '9': {
        /* numeric parameter */
        *vr.argptr = (*vr.argptr * 10) + (thechar - '0');
        return(FALSE);
    }
    case ';': {
        /* second arg coming up */
        vr.argptr = &vr.arg2;
        return(FALSE);
    }
    case '@': {
        /* VT200 insert arg1 blanks at cursor with normal attribute */
        break;
    }
    case 'c': {
        weare();
        break;
    }
    case 'g': {
        /* clear tab stop */
        /*
         0   at cursor
         3   all stops
        */
        break;
    }
    case 'h': {
        if (vr.foundqmark) {
            /* set DEC private modes */
            /*
             1   cursor key mode application/cursor
             2   ANSI/VT52 (no ANSI set available)
             3   132/80
             4   scroll smooth/jump
             5   screen reverse/normal
             6   origin relative/absolute
             7   wrap on/off
             8   autorepeat on/off
             9   interlace on/off
            */
            switch(vr.arg1) {
                case 3: {
                    if (vr.virsize >= (vr.virowmax + 1) * 132) {

```

```

        vr.vicols = 132;
        vr.vicolmax = 131;
    }
    break;
}
case 6: {
    vr.minorigin = TRUE;
    home();
    break;
}
case 19: {
/* full screen, no scrolling regions */
    break;
}
}
else {
/* ANSI 3.64 */
switch (vr.arg1) {
    case 4: {
        vr.insertmode = TRUE;
        break;
    }
    case 20: {
/* CR & LF mode--LF = NL, send CR-LF on Return */
        if (!switchnl) {
            docommand(echoMenu, SWITCHNL);
        }
        break;
    }
}
break;
}
case 'i': {
/* autoprint modes */
/* if qmark
   1  print cursor line
   4  auto print off
   5  auto print on
   else
   0  print screen (form feed mode selects terminator FF)
   4  printer controller off
   5  printer controller on
   all but NUL XON XOFF esc[5i & esc[4i are sent to printer
*/
break;
}
case 'l': {
if (vr.foundqmark) {
/* reset DEC private modes */
switch(vr.arg1) {
    case 3: {
        vr.vicols = 80;
        vr.vicolmax = 79;
        break;
    }
    case 6: {
        vr.minorigin = FALSE;
        home();
        vr.vitouched = TRUE;
        break;
    }
    case 19: {
/* use scrolling regions */
        break;
    }
}
}
else {
/* ANSI 3.64 */
switch(vr.arg1) {
    case 4: {
        vr.insertmode = FALSE;
        break;
    }
    case 20: {
/* CR only mode; LF == LF only */
        if (switchnl) {
            docommand(echoMenu, SWITCHNL);
        }
        break;
    }
}
break;
}
}

```

```

#define VIATTRIB
    case 'm': {
        /* set character attributes
         * 0 none
         * 1 bold
         * 4 underscore
         * 5 blink
         * 7 reverse
        if arg1 is 2, then turn off following attr type
        */
        switch (vr.arg1) {
            case 0: {
                /* reset */
                vr.attrib = 0;
                break;
            }
            case 4: {
                /* underscore */
                vr.attrib = (char) underline;
                break;
            }
            case 7: {
                /* standout mode */
                vr.attrib = (char) underline;
                break;
            }
        }
        break;
    }
#endif
    case 'n': {
        /* if foundqmark
         * 15 printer status request
         *      response: 10 ready, 11 not ready, 13 no printer
         * 20 user defined keys
         *      response: 20 unlocked, 21 locked
         * 26 keyboard language
         *      response: 1 North American
        */
        if (vr.arg1 == 5) {
            /* terminal status Response ESC[0n (terminal OK) */
            portline(&modem, 4, termok, FALSE);
        }
        if (vr.arg1 == 6) {
            /* send cursor position */
            ;
        }
        break;
    }
    case 'p': {
        /* soft terminal reset to "power up default states" */
        break;
    }
    case 'q': {
        /* load leds, dec private command */
        /*
         * if foundqmark
         * 0 clear leds
         * 1,2,3,4 light corresponding led
         * else
         * 0 no erase attr
         * 1 not erasable
         * 2 erasable
        */
        break;
    }
    case 'R': {
        /* set columns private command */
        if (!vr.foundbang)
            break;
        if (vr.arg1 > 0 && ((vr.viowmax + 1) * vr.arg1) <= vr.virsize) {
            /* ignore bad arguments */
            vr.vicolmax = vr.vicols = vr.arg1;
            --vr.vicolmax;
            vr.visize = (vr.viowmax + 1) * vr.vicols;
            vr.vitextp = vr.vitextbase + vr.visize;
            blankchars(vr.vitextp, vr.visize);
            /* blank out old material */
            home(); /* home the cursor to be safe */
        }
        break;
    }
    case 'r': {
        /* set scrolling region */
        if (vr.arg1)
            --vr.arg1;
        if (vr.arg2)

```

```

    --vr.arg2;
else
    vr.arg2 = 23;
    /* what vt100 clones will expect ? */
if (vr.arg1 < vr.arg2 && vr.arg2 < vr.virows) {
    /* ignore bad arguments */
    vr.virowmin = vr.arg1;
    vr.virowmax = vr.arg2;
    vr.visize = (vr.virowmax + 1) * vr.vicols;
    home();
    updcursor();
}
break;
}
case 'A': {
/* cursor up */
if (!vr.arg1)
    vr.arg1 = 1;

if (vr.ypos > vr.virowmin && vr.ypos <= vr.virowmax) {
    /* don't go past the top */
    count = vr.ypos - vr.virowmin;
    if (vr.arg1 > count)
        vr.arg1 = count;

    vr.vitextp -= vr.arg1 * vr.vicols;
    /* should be vitextp -= min(arg1, ypos - viowmin) * vircols; */
    vr.ypos -= vr.arg1;
    updcursor();
#endif DUMB
    if (vr.vitouched)
        updviscreen(viwind);
#endif
}
break;
}
case 'B': {
/* cursor down */
if (!vr.arg1)
    vr.arg1 = 1;

if (vr.ypos >= vr.virowmin && vr.ypos < vr.virowmax) {
    /* don't go past bottom */
    count = vr.virowmax - vr.ypos;
    if (vr.arg1 > count)
        vr.arg1 = count;

    vr.ypos += vr.arg1;
    vr.vitextp += vr.arg1 * vr.vicols;
    updcursor();
#endif DUMB
    if (vr.vitouched)
        updviscreen(viwind);
#endif
}
break;
}
case 'C': {
/* cursor forward */
if (!vr.arg1)
    vr.arg1 = 1;
if (vr.arg1 > vr.vicols - vr.xpos) {
    /* don't move past right edge */
    vr.arg1 = vr.vicols - vr.xpos;
}
vr.xpos += vr.arg1;
vr.vitextp += vr.arg1;
updcursor();
#endif DUMB
    if (vr.vitouched)
        updviscreen(viwind);
#endif
}
break;
}
case 'D': {
/* cursor back */
if (!vr.arg1)
    vr.arg1 = 1;
if (vr.arg1 > vr.xpos) {
    /* no backspace wrap */
    vr.arg1 = vr.xpos;
}
vr.xpos -= vr.arg1;
vr.inlastcol = FALSE; /* TODO nlglitch && CUP sequence == ? */
vr.vitextp -= vr.arg1;
updcursor();
#endif DUMB
}

```

```

        if (vr.vitouched)
            updviscreen(viwind);
#endif
        break;
}
case 'f':
case 'H': {
    /* set cursor */
    int tempx, tempy;

    if (!vr.arg1)
        vr.arg1 = 1;
    if (!vr.arg2)
        vr.arg2 = 1;
    tempy = --vr.arg1;
    tempx = --vr.arg2;
    if (vr.minorigin) {
        /* origin is within scrolling region */
        vr.arg1 += vr.virowmin;
        vr.arg2 += vr.virowmin;
    }
    if ( ( !vr.minorigin && tempy >= 0 && tempy <= vr.virows )
        || (vr.minorigin && tempy >= vr.virowmin && tempy < vr.virowmax ) )
        && tempx >= 0 && tempx < vr.vicols) {
        /* we only accept honest propositions! */
        vr.xpos = tempx;
        vr.ypos = tempy;
        vr.vitextp = vr.vitextbase + vr.ypos * vr.vicols + vr.xpos;
        vr.inlastcol = FALSE;
        updcursor();
    }
    break;
}
case 'J': {
    /* if foundqmark & vt200, selective erase */
    clearscreen(vr.arg1);
    break;
}
case 'K': {
    /* if foundqmark & vt200, selective erase */
    eraseline(vr.arg1);
    break;
}
case 'L': {
    /* insert n lines */
    if (!vr.arg1)
        vr.arg1 = 1;
    if (insertlns(vr.ypos, vr.arg1))
        break;
    vr.vitextp -= vr.xpos;
    vr.xpos = 0;
    vr.inlastcol = FALSE;
    break;
}
case 'M': {
    /* delete n lines */
    if (!vr.arg1)
        vr.arg1 = 1;
    if (deleteln(vr.ypos, vr.arg1))
        break;
    vr.vitextp -= vr.xpos;
    vr.xpos = 0;
    vr.inlastcol = FALSE;
    vr.vitouched = TRUE;
    break;
}
case 'P': {
    /* delete n chars */
    vr.inlastcol = FALSE;
    if (!vr.arg1)
        vr.arg1 = 1;
    if ( (count = vr.vicols - vr.xpos - vr.arg1) <= 0)
        break;
    BlockMove(vr.vitextp + vr.arg1, vr.vitextp, (Size) count);
    /* move remaining over to the left */
    BlockMove(vr.vitextp + vr.arg1 + vr.virsize,
              vr.vitextp + vr.virsize, (Size) count);
    /* move attributes */
    blankchars(vr.vitextp + count, vr.arg1);

    /* set the column change number */
    if (vr.xpos < vr.scrmap[vr.ypos].hit)
        vr.scrmap[vr.ypos].hit = vr.xpos;
    vr.vitouched = TRUE;
    break;
}
case 'X': {

```

```

/* vt200 erase char from cursor to arg1 -1 */
break;
}
}
return(TRUE);
}

/* insert a number of lines before another line */

insertlns(linenum, nolines)
int linenum, nolines;
{
    register char * charptr;
    register int insnochars; /* no. of chars to insert */
    int startchar; /* offset to first char */
    int movenochars; /* no. of chars to move down */

    if (linenum > vr.virowmax)
        /* wholly bad argument--ignore it */
        return(-1);
    if (linenum + nolines > vr.virowmax)
        /* poor argument--rephrase it! */
        nolines = vr.virowmax - linenum;

    if (nolines <= 0)
        return(-1);

    startchar = linenum * vr.vicols;
    charptr = vr.vitextbase + startchar;
    insnochars = nolines * vr.vicols;
    movenochars = vr.visize - startchar - insnochars;

    BlockMove(charptr, charptr + insnochars, (Size) movenochars);
    BlockMove(charptr + vr.virsize, charptr + insnochars + vr.virsize, (Size) movenochars);
    /* move attributes */

    /* blank out new lines */
    blankchars(charptr, insnochars);
    setmap(linenum, vr.virowmax);
}

/* special case scroll down with bit scrolling */

viscrdown()
{
    register char * charptr;
    register int insnochars; /* no. of chars to insert */
    int startchar; /* offset to first char */
    int movenochars; /* no. of chars to move down */
    Rect mover;
    struct scrcmap * scrp;
    GrafPtr saveport;

    GetPort(&saveport);
    SetPort(viwind->ptr);

    startchar = vr.virowmin * vr.vicols;
    charptr = vr.vitextbase + startchar;
    insnochars = vr.vicols;
    movenochars = vr.visize - startchar - insnochars;

    if (vr.vitouched)
        updviscreen(viwind);

    scrp = vr.scrcmap + vr.virowmin;
    BlockMove(scrp, scrp + 1,
              (Size) (sizeof(struct scrcmap) * (vr.virowmax - vr.virowmin - 1)));
    scrp->hit = vr.vicols;
    scrp->drawn = 0;
    /* fix the scrcmap info */

    BlockMove(charptr, charptr + insnochars, (Size) movenochars);
    BlockMove(charptr + vr.virsize, charptr + insnochars + vr.virsize,
              (Size) movenochars);

    /* blank out new lines */
    blankchars(charptr, insnochars);
    if (vr.cursor) {
        InvertRect(&vr.cursrect);
        vr.cursor = FALSE;
    }

    mover.top = vr.virowmin * vr.lineheight + vr.fontdescent + VITOP;
    mover.left = VILEFT;
    mover.bottom = vr.virowmax * vr.lineheight + vr.fontdescent + VITOP;
}

```

```

mover.right = viwind->ptr->portRect.right;
ScrollRect(&mover, 0, vr.lineheight, updateRgn);
SetPort(saveport);
}

/* delete a number of lines starting at a linenum */
deleteln(int linenum, int nolines)
{
    register char * charptr;
    register int delnochars; /* no. of chars to delete */
    int startchar; /* offset to first char */
    int movenochars; /* no. of chars to move up */

    if (linenum > vr.virowmax)
        /* wholly bad argument--ignore it */
        return(-1);
    if (linenum + nolines > vr.virowmax)
        /* poor argument--rephrase it! */
        nolines = vr.virowmax - linenum;

    if (nolines <= 0)
        return(-1);

    startchar = linenum * vr.vicols;
    charptr = vr.vitextbase + startchar;
    delnochars = nolines * vr.vicols;
    movenochars = vr.visize - startchar - delnochars;

    BlockMove(charptr + delnochars, charptr, (Size) movenochars);
    BlockMove(charptr + delnochars + vr.virsize, charptr + vr.virsize,
              (Size) movenochars);

    /* now blank out the new lines at the end */
    charptr += movenochars;
    blankchars(charptr, delnochars);

    /* TODO is this even better done by bit scrolling? */

    setmap(linenum, vr.virowmax);
}

/* special case scroll up with bit scrolling */

viscrup()
{
    register char * charptr;
    register int delnochars; /* no. of chars to delete */
    int startchar; /* offset to first char */
    int movenochars; /* no. of chars to move up */
    Rect mover;
    struct scrmap * scrp;
    GrafPtr saveport;

    GetPort(&saveport);
    SetPort(viwind->ptr);

    startchar = vr.virowmin * vr.vicols;
    charptr = vr.vitextbase + startchar;
    delnochars = vr.vicols;
    movenochars = vr.visize - startchar - delnochars;

    if (vr.vitouched)
        updviscreen(viwind);

    BlockMove(vr.scrmap + vr.virowmin + 1, vr.scrmap + vr.virowmin,
              (Size) (sizeof(struct scrmap) * (vr.virowmax - vr.virowmin)));
    scrp = vr.scrmap + vr.virowmax;
    scrp->hit = vr.vicols;
    scrp->drawn = 0;
    /* fix the scrmap info */

    BlockMove(charptr + delnochars, charptr, (Size) movenochars);
    BlockMove(charptr + delnochars + vr.virsize, charptr + vr.virsize, (Size) movenochars);

    /* now blank out the new lines at the end */
    charptr += movenochars;

    blankchars(charptr, delnochars);

    /* now move around the bits */
    if (vr.cursor) {

```

```

InvertRect(&vr.cursrect);
vr.cursoron = FALSE;
}
mover.top = vr.virowmin * vr.lineheight + vr.fontdescent + VITOP;
mover.left = VILEFT;
mover.bottom = vr.virowmax * vr.lineheight + vr.fontdescent
+ VITOP + vr.lineheight;
mover.right = viwind->ptr->portRect.right;

ScrollRect(&mover, 0, - vr.lineheight, updateRgn);
SetPort(saveport);
}

/* tell the host what we are */

weare()
{
/* device attribute query
second parameter is option enabled
0 no options
1 processor option (STP)
2 advanced video (AVO)
3 STP+AVO
4 graphics option (GO)
5 STP+GO
6 AVO+GO
7 STP+AVO+GO
*/
portline(&modem, 7, "\033[?1;0c", FALSE, FALSE);
/* parameter 2 == 2 if 24X132 screen available */
/* we are a vt100 */
}

home()
{
/* home it */
if (vr.minorigin) {
/* in DECOM mode, origin is within scrolling region */
vr.xpos = 0;
vr.ypos = vr.virowmin;
vr.vitextp = vr.vitextbase + vr.virowmin * vr.vicols;
}
else {
/* origin is at 0,0 */
vr.xpos = vr.ypos = 0;
vr.vitextp = vr.vitextbase;
}
}

escprep()
{
escapemode = outputwind->escapemode = TRUE;
vr.foundback = FALSE;
vr.foundqmark = FALSE;
vr.foundpound = FALSE;
vr.foundparen = FALSE;
vr.foundbang = FALSE;
}

/* move the cursor down a line, scrolling if necessary */

movedown()
{
if (vr.ypos < vr.virowmax) {
vr.vitextp += vr.vicols;
vr.ypos++;
#endif DOUPD
if (vr.ypos < vr.virowmax - 1)
/* if on line next to bottom, don't draw immediately */
upviscreen(viwind);
#endif
}
else {
if (bitscroll) {
viscrup();
}
else {
deletelns(vr.virowmin, 1);
}
}
}

long blinkat = 0;

```

```

cursblink()
{
    GrafPtr saveport;
    WindowPtr frontwind;

    frontwind = FrontWindow();

    if (((WindowPeek) frontwind)->windowKind < userKind)
        return(-1);
    if (swindptr(frontwind)->type != VIWIND)
        return(-1);

    if (!viblink) {
        if (vr.cursoron == FALSE) {
            updcursor();
        }
        return(0);
    }
    if (TickCount() >= blinkat) {
        if (vr.cursoron) {
            InvertRect(&vr.cursrect);
            vr.cursoron = FALSE;
        }
        else {
            updcursor();
        }
        blinkat = TickCount() + *carettime;
    }
}

vioutput(count)
int count;
{
    extern char mask;
    register char maskb;
    register unsigned char bufchar;
    register char * bufp;
    register char * bufend;

    if (outputwind != viwind) {
        /* copy the globals about: minimizes pointer mush in code */
        BlockMove(&vr, &iwind->virec, (Size) sizeof(struct virecord));
        iwind = outputwind;
        BlockMove(&iwind->virec, &vr, (Size) sizeof(struct virecord));
        VIsize(iwind, &virect);
    }
    maskb = mask;
    bufend = stake + count;
    for (bufp = stake; bufp < bufend; bufp++) {
        bufchar = *bufp &= maskb;
#ifdef UW
        if (uwmode) {
            uwmode = FALSE;
            stake++;
            if (uhandle(bufchar))
                return(bufend - stake);
            continue;
        }
#endif
        if (bufchar < SP || bufchar == DEL) {
            /* DEL is skipped */
            if (bufp != stake) {
                /* spit out normal text skipped so far */
                putvichars(bufp);
            }
            /* now we've put out the plain text, we can handle the */
            /* ASCII control codes */
            switch (bufchar) {
                case NUL: {
                    /* do nothing */
                    break;
                }
#ifdef UW
                case SOH: {
                    /* next byte is a UW command */
                    uwmode = TRUE;
                    break;
                }
#endif
                case VT:
                case FF:
                case LF: {
                    extern char saveonlf;

```

```

extern int fcapture;
extern int capwindtype;

movedown();
if (checksave())
    return(bufend - stake);
if (! switchnl)
    break;
/* if in switchnl mode, fall through to do a CR */
}

case CR: {
    if (vr.inlastcol) {
        if (vr.ypos > 0 && vr.xpos == 0) {
            /* let's be extra cautious */
            vr.xpos = vr.vicols;
            --vr.ypos;
        }
        vr.inlastcol = FALSE;
    }
    vr.vitextp -= vr.xpos;
    vr.xpos = 0;
    vr.vitouched = TRUE; /*
    break;
}

case BS: {
    vr.inlastcol = FALSE;
    if (vr.xpos > 0) {
        --vr.vitextp;
        --vr.xpos;
    }
    updcursor();
    break;
}

case BEL: {
    beeper();
    break;
}

case TAB: {
    vr.inlastcol = FALSE;
    if (vr.xpos < vr.vicolmax) {
        count = ((vr.xpos + 8) & ~7) - vr.xpos;
        vr.vitextp += count;
        vr.xpos += count;
    }
    break;
}

case SUB:
case CAN: {
    /* cancel current escape sequence */
    escapemode = outputwind->escapemode = FALSE;
    break;
}

case ESC: {
    escprep();
    break;
}

case GS: {
    /* go into graphics mode */
    if (tekwind != NULL) {
        outputwind = tekwind;
        stake++;
        return(bufend - stake);
    }
    break;
}

case XOFF: {
    hostxoff = TRUE;
    break;
}

case XON: {
    hostxoff = FALSE;
    break;
}

default: {
    escapemode = outputwind->escapemode = FALSE;
    /* risk of bad sequence should be avoided */
}
}

stake++;

}

else if (escapemode) {
    if (eschandle(bufchar))
        escapemode = outputwind->escapemode = FALSE;
    stake++;
    continue;
}

/* end if < SPACE; else do nothing but increment bufp in for */

```

```

}

if (bufp > stake) {
    /* we process plain text not yet written out */
    putvichars(bufp);
}

/* this was replaced by calls on LF's <rowmax & at end of input
 * drawscreen();
 */

return(0);
}

/* draw the vi screen from the character map in memory, but only those parts
 * that have been marked dirty in scrmap */

updviscreen(thewind)
struct window * thewind;
{
    register char * attrp;
    register struct scrmap * scrp; /* contents set < vicols if changed */
    register char * textp;

    int rblanks; /* # blanks not to draw at end of line */
    GrafPtr saveport;
    int yloc; /* x, y pen locations */
    int xloc;
    static int oldoffset;
    Struct scrmap * scrend;
    Rect killrect; /* erase old text */

    GetPort(&saveport);
    SetPort(thewind->ptr);

    vr.vitouched = FALSE;

    if (vr.cursor) {
        InvertRect(&vr.cursrect);
        vr.cursor = FALSE;
    }

#endif VIWRAP
/* handle cursor calculations */
vr.cursrect.top = VITOP + (vr.ypos + 1) * vr.lineheight + vr.fontdescent - 1;
/* underline cursor */
textp = vr.vitextp - vr.xpos;
vr.cursrect.left = TextWidth(textp, 0, (int) vr.xpos) + VILEFT + vr.scroffset;
/* -1 for funny horizontally offset character origins ? */
vr.cursrect.bottom = vr.cursrect.top + 2;
vr.cursrect.right = vr.cursrect.left + CharWidth(*vr.vitextp);
if (vr.cursrect.left == vr.cursrect.right)
    /* if someone handed us a zero width char, handle it */
    vr.cursrect.right = vr.cursrect.left + 6;

/* calculate pen offset so long lines are shown--cursor
 * is guaranteed onscreen
 * TODO this screen offset calculation has to be done properly for large windows
 */
if (!vr.viwrap) {
    /* no divides by 0, please */
    vr.offset = 0;
    /* normal case */
}
else {
    vr.offset = -vr.viwrap * (vr.cursrect.left / vr.viwrap);
    if (vr.offset) {
        vr.offset += 100;
        vr.cursrect.left += vr.offset;
        vr.cursrect.right += vr.offset;
    }
    if (vr.offset != oldoffset) {
        /* redraw whole screen */
        virefresh();
        oldoffset = vr.offset;
    }
}
#endif

/* set up initial line kill rectangle for row 0 */

killrect.top = VITOP + vr.fontdescent;
killrect.bottom = killrect.top + vr.lineheight;

yloc = VITOP + vr.lineheight; /* pen vert pos */
textp = vr.vitextbase;

for (scrp = vr.scrmap, scrend = vr.scrmap + vr.virows; scrp < scrend; scrp++) {
    if (scrp->hit < vr.vicols) {

```

```

/* there's been a change, draw the changed part of the line */
xloc = TextWidth(textp, 0, scrp->hit) + VILEFT + vr.offset + vr.scroffset;
    /* left end of changed text */
killrect.left = xloc;
killrect.right = scrp->drawn; /* last location of pen */
EraseRect(&killrect);
/* would try to avoid drawing left blanks here if necessary */
MoveTo((int) xloc, (int) yloc);
rblanks = virblanks(textp + vr.vicols);

/* TODO current rblanks test may leave attributes undrawn */
{
    register int drawcount;
    register char attcurrent; /* current attrib used by the grafport */
    register int downcount;
    register int startpos;

    downcount = vr.vicols - scrp->hit - rblanks + 1;
    startpos = (int) scrp->hit;
    attp = textp + startpos + vr.virsize;
    attcurrent = *attp;
    TextFace(attcurrent);

    for (drawcount = 0; --downcount > 0; attp++, drawcount++) {
        if (*attp != attcurrent) {
            if (drawcount > 0) {
                /* draw what's there */
                DrawText(textp, startpos, drawcount);
                startpos += drawcount;
                drawcount = 0;
            }
            attcurrent = *attp;
            TextFace(attcurrent);
        }
        switch (attcurrent) {
            case underlineStyle:
            case boldStyle: {
                TextFace(attcurrent);
                break;
            }
            case invstyle: {
                /* go into SrcXOR mode? */
                break;
            }
        }
    }
    if (drawcount > 0) {
        DrawText(textp, startpos, drawcount);
    }
}
scrp->drawn = qd.thePort->pnlLoc.h;
scrp->hit = vr.vicols; /* set scrmap to no cols changed */
}
killrect.top += vr.lineheight;
killrect.bottom += vr.lineheight;
yloc += vr.lineheight;
textp += vr.vicols;
}

#endif VIWRAP
InvertRect(&vr.cursrect);
vr.cursor = TRUE;
#endif
SetPort(saveport);
}

updcursor()
{
    if (vr.cursor) {
        InvertRect(&vr.cursrect);
    }

    /* handle cursor calculations */
    vr.cursrect.top = VITOP + (vr.ypos + 1) * vr.lineheight + vr.fontdescent - 1;
    /* underline cursor */
    vr.cursrect.left = TextWidth(vr.vitextp - vr.xpos, 0, (int) vr.xpos) + VILEFT + vr.scroffset;
    /* -1 for funny horizontally offset character origins ? */
    vr.cursrect.bottom = vr.cursrect.top + 2;
    vr.cursrect.right = vr.cursrect.left + CharWidth(*vr.vitextp);
    if (vr.cursrect.left == vr.cursrect.right)
        /* if someone handed us a zero width char, handle it */
        vr.cursrect.right = vr.cursrect.left + 6;

    InvertRect(&vr.cursrect);
    vr.cursor = TRUE;
}

```

```

}

/* reset the map of locations needing to be redrawn */

setmap(firstrow, lastrow)
int firstrow, lastrow;
{
    register struct scrmap * scrp;
    register int count;

    /* reset the screen map to force redraw */
    for (scrp = vr.scrmap + firstrow, count = lastrow - firstrow + 2;
         --count > 0; scrp++)
        scrp->hit = 0;
    vr.vitouched = TRUE;
}

resetmap()
{
    register struct scrmap * scrp;
    register int count;

    /* reset the screen map to force redraw */
    for (scrp = vr.scrmap, count = vr.virows; --count; scrp++) {
        scrp->drawn = 0;
        scrp->hit = vr.virows;
    }
}

virefresh()
{
    setmap( 0, vr.virowmax);
}

/* put out the characters from the global stake up to
 * the end of non-control chars in the buffer
 */
/* TODO this thing is lousy with globals, should be cleaned up a little ? */
/* TODO putvichars(thewind, bufs, bufend)? */

extern char * stake;
/* holds the last un-interpreted position in the buffer */

putvichars(bufstop)
register char * bufstop;
{
    register char * vitempp = vr.vitextp;
    register char * attempp = vitempp + vr.virsize;
    register char * staketemp = stake;
    register int ypost = vr.ypos;
    register int xpost = vr.xpos;
    register char inserton = vr.insertmode;
    /* we use local temps to speed things up */

    vr.vitouched = TRUE;
    /* set the column change number if < old change pos */
    for (; staketemp < bufstop; staketemp++) {
        if (inserton) {
            /* push line over for new character */
            BlockMove(vitempp, vitempp + 1, (Size) (vr.vicolmax - xpost));
            BlockMove(attempp, attempp + 1, (Size) (vr.vicolmax - xpost));
        }
        if (vr.scrmap[ypost].hit == vr.vicols && !inserton) {
            /* insert off and no mods to line, ok to try to avoid drawing */
            if (staketemp != *vitempp || vr.attrib != *attempp) {
                /* if buffer & vitext are different, let it be redrawn */
                vr.scrmap[ypost].hit = xpost;
            }
        }
        else {
            if (xpost < vr.scrmap[ypost].hit)
                vr.scrmap[ypost].hit = xpost;
        }
        *vitempp++ = *staketemp;
        *attempp++ = vr.attrib;

        /* TODO for xn glitch, place this if first & just increment xpos here */
        if (++xpost > vr.vicolmax) {
            vr.inlastcol = TRUE;           /* for newline glitch */
            if (ypost < vr.virowmax) {
                vitempp += vr.vicols - xpost;
                attempp += vr.vicols - xpost;
                /* to beginning of next line */
                ypost++;
            }
        }
    }
}

```

```

    else {
        if (bitscroll ? !viscrup() : !deletelns(vr.viowmin, 1)) {
            /* back to beginning of this new line */
            vitempp -= xpost;
            atttempp -= xpost;
        }
        xpost = 0;
    }
    else
        vr.inlastcol = FALSE;           /* for newline glitch */
}
}

vr.vitextp = vitempp;
stake = staketemp;
vr.ypos = ypos;
vr.xpos = xpost;
/* update the globals */
}

```

```

clearscreen(clearmode)
int clearmode;
{
    /* clear screen variations */
    register char * blankend;
    register char * tempp;

```

```

    if (vr.vitouched) {
        updviscreen(viwind);
        wait(20);
    }
    temp = vr.vitextbase;
    blankend = vr.vitextend;
    if (clearmode == 0) {
        /* from cursor */
        tempp = vr.vitextp;
        vr.scrmap[vr.ypos].hit = vr.xpos;
        setmap((vr.ypos + 1), vr.virows - 1);
    }
    else if (clearmode == 1) {
        /* to cursor */
        blankend = vr.vitextp;
        setmap(0, vr.ypos);
    }
    else if (clearmode == 2) {
        /* whole screen */
        setmap(0, vr.virows - 1); */
        resetmap();
        EraseRect(&virect);
        vr.cursor = FALSE; /* by necessity! */
    }
    else
        return(-1);
    blankchars(tempp, blankend - tempp);
    /* do the work */
}

```

```

eraseline(erasemode)
int erasemode;
{
    register char * tempp;
    register int count;

    /* erase line variations */
    if (vr.vitouched)
        updviscreen(viwind);
    if (erasemode == 0) {
        /* erase from cursor */
        count = vr.vicols - vr.xpos;
        tempp = vr.vitextp;
        /* set the column change number */
        if (vr.xpos < vr.scrmap[vr.ypos].hit)
            vr.scrmap[vr.ypos].hit = vr.xpos;
    }
    else if (erasemode == 1) {
        /* erase to cursor */
        count = vr.xpos + 1;
        tempp = vr.vitextp - vr.xpos;
        /* beginning of line to cursor */
        vr.scrmap[vr.ypos].hit = 0;
    }
    else if (erasemode == 2) {
        /* erase line */
    }
}

```

```

count = vr.vicols;
temp = vr.vitextp - vr.xpos;
/* beginning of line */
vr.scrmap[vr.ypos].hit = 0;
}
else
    return(-1);

/* now erase it */
blankchars(temp, count);
vr.vitouched = TRUE;
}

blankchars(charp, count)
register char * charp;
register int count;
{
    register char * attp;

    attp = charp + vr.virsize;
    count++;
    /* setup for predecrement */

    while (--count > 0) {
        /* blank the new spaces */
        *charp++ = ' ';
        *attp++ = '\0';
    }
}

/* this routine saves the current vi context in the current window's vi record
   and sets the context to the arg's vi record
   the scrmap array has to be copied also
*/
vicontext(thewind)
struct window * thewind;
{
    BlockMove(&vr, &viwind->virec, (Size) sizeof(struct virecord));
    BlockMove(vr.scrmap, viwind->virec.scrmap,
              (Size) (vr.virows * sizeof(struct scrmap)));

    viwind = thewind;
    BlockMove(&viwind->virec, &vr, (Size) sizeof(struct virecord));
    BlockMove(viwind->virec.scrmap, vr.scrmap,
              (Size) (vr.virows * sizeof(struct scrmap)));
    VIsize(thewind, &virect);
}

virblanks(nblankp)
register char * nblankp;
{
    register int rblanks;

    for (rblanks = 0; *--nblankp == SP && rblanks <= vr.vicols; rblanks++)
        /* determine # blanks at right to avoid drawing */
    ;
    return(rblanks);
}

wait(ticks)
int ticks;
{
    long waitfor;

    waitfor = TickCount() + ticks;
    while (TickCount() < waitfor)
        ;
}

#endif NOLBLANKS
/* unnecessary since quickdraw doesn't draw 'em anyway */
int lblank; /* # blanks not to draw at beginning of line */

/* try to avoid drawing blanks on left end of line */
for (temp = textp + scrp->hit, lblank = 0;
     *temp++ == SP && lblank <= vr.vicols; lblank++)
    /* determine # blanks at left to avoid drawing */
    ;
    xloc += lblank * CharWidth(' ');
#endif

```

```

#include "Types.h"

long * keytrans; /* actually void * (*keytrans)() ? but who cares ? */
long * key1tran;

#asm
; this routine resets the option key flag so NO option key remapping is done by
; the standard macintosh key configuration code; the option-key-down bit is
; cleared

    public _tsaveA5
    public _trestoreA5
    public _controlmode

OrigA5 EQU $904

cseg

    public _keytrap
_keytrap:
_tsaveA5:
    move.l  a5,a3
    move.l  (OrigA5),A5
    ; end of save routine
    tst.b   _controlmode
    beq     _norm
    bclr   #2,d1  ; clear option key bit when controlmode on
_norm:
    move.l  _key1tran,a2
    jsr     (a2)
_trestoreA5:
    move.l  a3,a5
    rts

#endifasm

keyinit()
{
    /* add our key filter to preprocess keys before the standard one */
    keytrans = (ProcPtr) 0x29e;
    key1tran = (long *) *keytrans;
#define KEYFILTER
#ifndef KEYFILTER
    /* *keytrans = (long *) keytrap; */
#endif
#endifasm
#endiff
}

keyclose()
{
#endifasm
    move.l  _keytrans,a2
    move.l  _key1tran,(a2)
#endifasm
}

```

12/10/94 12:35

main.c

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

/*
 * a/k/a dumb virtue
 *
 * all stuff in this mac/term directory is
 * copyright 1984 1985 1986 Kevin Eric Saunders
 *
 */

/*
 * functional serial communications with UNIX
 * TE in multiple windows
 * working TE Controls on multiple windows
 *
 */

#include "stddef.h"
#include "stdio.h"
#include "stdlib.h"

#include "defs.h"
#include "toolutils.h"
#include "quickdraw.h"
#include "controls.h"
#include "ctype.h"
#include "desk.h"
#include "dialogs.h"
#include "disks.h"
#include "errno.h"
#include "events.h"
#include "fcntl.h"
#include "fonts.h"
#include "memory.h"
#include "menus.h"
#include "OSEvents.h"
#include "osutils.h"
#include "packages.h"
// #include "pb.h"
#include "printing.h"
#include "resources.h"
#include "retrace.h"
#include "scrap.h"
#include "SegLoad.h"
#include "serial.h"
#include "stat.h"
#include "Errors.h"
#include "string.h"
#include "textedit.h"
#include "time.h"
#include "types.h"
#include "windows.h"

#include "comm.h"
#include "commdefs.h"
#include "menudefs.h"
#include "windmenu.h"
#include "windmenudefs.h"
#include "echomenudefs.h"
#include "cursormenudefs.h"
#include "uw.h"
#include "startup.h"
#include "mac.h"

#define BPORTIN -8
#define BPORTOUT -9
#define inZoomIn    7
#define inZoomOut   8

/* growRect & dragRect are bounds */
short growRect[] = { 20, 40, 5000, 5000 };
short dragRect[] = { 4, 4, 338, 508 };
short nullrect[] = { 0, 0, 0, 0 };

EventRecord myEvent;      /* big time party global--everybody uses */

short romtype;           /* type of ROM present on mac */
#define BADKEY '\317'
#define MAXKEYCODE 48

/* keycode -> control char mapping for option key
ASDFHGZCV
.BQWERYT.2
..6.....-
]OU[IP.LJ.
K.\./NM
```

```

*/
char optkeymap[MAXKEYCODE] = "\001\023\004\006\010\007\032\030\003\026\
\317\002\021\027\005\022\031\024\317\000\
\317\317\036\317\317\317\037\317\317\
\035\017\025\033\011\020\317\014\012\317\
\013\317\034\317\177\016\015";

struct scrmap {
    int hit;
    int drawn;
};

struct virecord {
    int virows;           /* real number of rows */
    int virowmin;         /* first active row */
    int virowmax;         /* scroll if row to become > max */
    int vicols;           /* number of cols in use */
    int vircols;          /* NOT USED the REAL number of cols in the array */
    int vicolmax;
    int visize;           /* from 0 to maxrow * VICOLS; varies with scroll window size */
    int virsize;

    int lineheight;
    int fontheight;
    int fontdescent;

/* maps changes to screen by row with column change start position (xpos) */
/* and last pen location for eraserect.right */

    char insertmode;      /* insert on flag */
    char vitouched;       /* flag indicating redraw needed */
    /* does the screen need to be redrawn when out of input data? */
    char inlastcol;        /* TRUE when x=80 & nlglitch in effect */
    char minorigin;        /* TRUE when home == virowmin */

    int xpos;              /* horizontal cursor location */
    int ypos;              /* vertical cursor location */
    Rect cursrect;         /* vi cursor rectangle */
    char curson;           /* cursor state, on or off */

    char * vitextp;         /* ptr to current vi character */
    char * vitextbase;       /* ptr to base of array */
    char * vitextend;        /* ptr to end of array */
    struct scrmap * scrmap; /* the map of the screen drawing state */
    char attrib;            /* screen attribute, normal */

/* for save and restore cursor */
    int oldx;
    int oldy;
    char * oldvip;

/* ANSI interpretation modes */
    char foundbrack;        /* TRUE if '[' found after ESC */
    char foundqmark;        /* TRUE if '?' found, indicating DEC modes */
    char foundpound;        /* TRUE if '#' found, indicating DEC double modes */
    char foundparen;        /* TRUE if '(' found, indicating undocumented DEC mode */
    char foundbang;         /* TRUE if '!' found, indicating my private commands */

    int scroffset;           /* offset for scrollbar loc so user can scroll */
    int viwrap;              /* pixel at which vi does wrap */
    int offset;               /* negative offset for virtual screen scrolling when drawing */

    int arg1;
    int arg2;
    int * argptr;
};

/* the basic window definition */

struct window {
    WindowPtr   ptr;
    TEHandle    hTE;
    struct virecord virec;
    char escapemode; /* TRUE in escape sequence */
    ControlHandle hscroll;
    ControlHandle vscroll;
    char scroll;      /* scroll on or off */
    char dirty;        /* text window has been touched if TRUE */
    short type;
    short fvol;        /* volume ref of open file */
    long fdir;         /* directory ref of open file */
    int cstate;        /* REP UIO status of a window */
} window[MAXWIND];

```

```

/* TODO ptr == TEptr; add viptr, tekptr? or should there just be more structs?
add TE context list */

struct window * inputwind, * outputwind, * bugwind, * viwind, * tekwind;

CursHandle curtext;           /* custom text beam */
CursHandle curvi; /* vi target */
CursHandle curtek; /* tek crosshair */
CursHandle watch; /* wait */

char error[50]; /* global error message buffer */

unsigned long clearprompt = 0; /* timer for clearing prompt */
/* gatetick -- flag to indicate adequate buffer accumulation
* to minimize TEInsert overhead
* the program needs larger TEInsert calls to take advantage of this
*/

#ifndef VRT
VBLTask gateVBL;
char gatetick = TRUE; /* do portRead when TRUE */
#else
unsigned long thistick = 0;
unsigned long nexttick = 0; /* do portRead when == tickcount */
#endif

int readlag = 1; /* ticks gatekeeper waits before doing Read */
int writelag = 0; /* lag in ticks between Writes when sending lines */
int captimeout = 1800; /* 30 second timeout in ticks for fcapture */

long blockNBytes = 400; /* # bytes gatekeeper waits before writing text */
/* reset by setspeed */
long nbytes = 0; /* no. of bytes ready -- long ex. SUMACC */
long onbytes = 0; /* previous serial byte-ready count -- long ex. SUMACC */

/* communications modes defaults, used by handlekey() in comm.c */

char logmode = FALSE;
char local = TRUE;
char skipprompt = FALSE;
char vion = FALSE;
char noserd = FALSE; /* no ram serial driver available */
char doflow = TRUE; /* do flow control */
char xoff = FALSE;
char hostxoff = FALSE; /* host has xoff-ed transmission */

/* kludge to avoid DrawControls when using menu autokeying for cursor motion */
char autokeycommand = FALSE;

char controlmode = TRUE; /* Option key produces control chars when true */
char allcontrol = FALSE; /* when TRUE all keys interpreted as menu keys */
extern char useselbottom; /* special menu interpretation mode */

char keyedmenu = FALSE; /* menu was selected by a keystroke */
char shifton = FALSE; /* shift key was depressed--MenuKey stupidity */
GrafPtr wport;

char macroson = TRUE; /* key macros are on */

extern longtoa();
extern struct window * makewindow();
char windswitch = FALSE; /* wait until things settle down to do output */
char tewindswitch = FALSE; /* make a new output window */

int nummasters = 20;

#include "vi.h"

main()
{
    WindowPtr mousewind; /* result of Findwindow */

    /* the following stuff is supposed to stay with segment 1 */
    MaxApplZone();

    while (--nummasters)
        MoreMasters();

    InitGraf(&qd.thePort); /* == &QD->QDthePort */
    InitFonts();
    InitWindows();
    TEInit();
    InitDialogs((ProcPtr)NULL);
    InitMenus();
}

```

```

init();
#ifndef SEGMENT
UnloadSeg(init);
#endif
{
    /* free up memory */
    long max, grow;

    max = MaxMem(&grow);
}
while (TRUE) {
    SystemTask();
#ifdef VRT
    if (gatetick) {
        if (readlag > 0)
            gatetick = FALSE;
    } else
        if (thistick = TickCount()) >= nexttick || !readlag) {
            /* Wait for timeout to check input amount & do occasional stuff */
            nexttick = thistick + readlag;
#endif
    setcursor();
    /* setcursor may not be fast enough if gatetick too long */
    cursblink(); /* do vi blink */
    doserial();
    TEIdle(inputwind->hTE);
}
if (printer.transon) {
    /* if we're sending text to printer, keep it up */
#ifdef ASYNCWRITE
    if (printer.transcont) {
        /* continue flag raised */
        if (endline(&printer))
            sendline(&printer);
    }
#else
    sendline(&printer);
#endif
}
if (clearprompt) {
    if (clearprompt < TickCount())
        menurestore();
}
#endif TEWINDSWITCH
if (tewindswitch) {
    struct window * awind;
    int count;

    for (count = 0; ++count < VIWINDNUM; ) {
        if (window[count].ptr == NULL) {
            awind = makewindow(count, TEXTWIND, deffont, defsize, 0, 0, 0, 0);
            if (awind != NULL) {
                ShowWindow(awind->ptr);
                outputwind = awind;
                selwind(awind->ptr);
                windswitch = TRUE;
                break;
            }
        }
    }
    tewindswitch = FALSE;
}
#endif
if ( ! GetNextEvent(everyEvent, &myEvent) ) {
    /* event is null or has been handled by SystemEvent */
#endif DORES
    /* TODO how to fix document loading under MultiFinder? */
    if (myEvent.what == nullEvent) {
        static account = 0;

        /* load files, after startup when everything settled */
        if (account < appcount) {
            AppFile appfile;

            if (account >= VIWINDNUM) {
                /* no more windows left, drop the rest */
                appcount = 0;
            }
            /* get text from files & load into window 0, 1, etc. */
            GetAppFiles(account + 1, &appfile);
            if (makewindow(account, TEXTWIND, deffont, defsize,
                           0, 0, 0, 0)) {
                ShowWindow(window[account].ptr);
                gettext(&window[account], &appfile.fName,
                       (int) appfile.vRefNum);
            }
        }
    }
}

```

```

        window[account].fvol = SFSaveDisk;
        window[account].fdir = CurDirStore;
        window[account].dirty = FALSE;
    }
    account++;
}
#endif
continue;
}

switch (myEvent.what) {
    case app4Evt: {
        if (myEvent.message & 0x01000000L) {
            if (myEvent.message & 1) {
                /* resume after Switcher suspension */
                keyinit();
            }
            else {
                /* shut things off--Switcher suspension */
                keyclose();
            }
        }
        break;
    }
    case mouseDown: {
        WindowPtr frontwind;
        long windowpart;

        if (clearprompt)
            /* clear off the prompt line */
            menurestore();
        frontwind = FrontWindow();
        windowpart = FindWindow(myEvent.where, &mousewind);
        switch ((int) windowpart) {
            case inMenuBar: {
                long menuSel; /* long ex. SUMACC */
                extern int menurectsize; /* area of largest menu */

                if (!enoughmem(menurectsize))
                    /* don't allow MenuSelect if there's not enough
                     room to save the screen beneath the largest
                     menu rect! */
                    break;
                turnxoff();
                if ( (menuSel = MenuSelect(myEvent.where)) ) {
                    if (myEvent.modifiers & optionKey)
                        useselbottom = TRUE;
                    /* cursor motion moves from end of selection range */
                    domenu(menuSel);
                    HiliteMenu(0);
                    useselbottom = FALSE;
                }
                break;
            }
            case inSysWindow: {
                SystemClick(&myEvent, mousewind);
                break;
            }
            case inDrag: {
                /* "Eric, this just won't go over in the States!" */
                /* "teehee!...." */
                turnxoff();
                if (mousewind == frontwind)
                    DragWindow(mousewind, myEvent.where, &dragRect);
                selwind(mousewind);
                break;
            }
            case inGoAway: {
                turnxoff();
                if (TrackGoAway(mousewind, myEvent.where)) {
                    /* SendBehind(mousewind, (WindowPtr) NULL); */
                    /* should kill if dialog OKs? */
                    /* killwindow( swindptr(frontwind) ); */
                    SendBehind(mousewind, (WindowPtr) NULL);
                    selwind(FrontWindow());
                }
                break;
            }
            case inGrow: {
                turnxoff();
                if (mousewind == frontwind) {
                    long result; /* long ex. SUMMAC */
                    int height, width;

                    if ( (result = GrowWindow(mousewind,
                                              myEvent.where, &growRect)) ) {
                        height = result >> 16;

```

```

        width = result & 0xffff;
        sizewindow(mousewind, height, width);
    }
    selwind(mousewind);
    break;
}
case inContent: {
/* TODO should windows be set up so scrbars can be used
 * while window not on top?
 * how about double clicking as a bring to front signal?
 */
    if (mousewind == frontwind){
        turnxoff();
        clickinwind(swindptr(mousewind));
    }
    selwind(mousewind);
    break;
}
case inZoomIn:
case inZoomOut: {
    if (TrackBox(mousewind, myEvent.where, (int) windowpart)) {
        SetPort(mousewind);
        EraseRect(&mousewind->portRect);
        ZoomWindow(mousewind, (int) windowpart, (Boolean) FALSE);
        sizewindow(mousewind, mousewind->portRect.bottom, mousewind->portRect.right);
    }
    break;
}
break;
}
case keyUp: {
/* autokeycommand is over */
if (autokeycommand) {
    /* autokey over, draw controls */
    HiliteMenu(0);
    DrawControls(inputwind->ptr);
    autokeycommand = FALSE;
    TEActivate(inputwind->hTE);
}
break;
}
case keyDown:
case autoKey: {
/* this key conversion stuff is uncool, a more general approach
   takes up too much space though -- TODO this should not
   be cluttering up main...
*/
extern char litinon;
int makebs;           /* flag for handlekey */
char thechar;
char keycode;
struct window * frontwind;
extern char menukeys();

#ifdef KEYMACROS
Handle macroh;
extern Handle macromatch();
int count, total;
#endif
if (((WindowPeek) FrontWindow())->windowKind < userKind)
    break;

frontwind = swindptr(FrontWindow());
thechar = myEvent.message & 0xff;
keycode = (myEvent.message >> 8) & 0xff;

#ifdef KEYMACROS
if (macroson && ( (macroh = macromatch()) != (Handle) NULL ) {
    /* put out a key macro */
    for (count = 0, total = GetHandleSize(macroh); count++ < total; ) {
        char mchar;
        char nchar;

        mchar = *(*macroh + count);
        if (mchar == '\\') {
            if (count++ >= total)
                break;
            mchar = *(*macroh + count);
            if (mchar == '\\') {
                handlekey('\\', FALSE);
                break;
            }
            if (count++ >= total)
                break;
            nchar = *(*macroh + count);
        }
    }
}

```

```

        docommand((int) mchar, (int) (nchar - 'A') );
        /* menus labeled "AA" through "ZZ"*/
    }
    handlekey(mchar, FALSE);
    break;
}
#endif

if ((thechar = keyswitch(keycode, thechar)) == 0)
    break;

if (allcontrol)
    myEvent.modifiers |= cmdKey;
if (myEvent.modifiers & optionKey) {
    /* cursor motion moves from end of selection range */
    useselbottom = TRUE;
}
if (myEvent.modifiers & cmdKey) {
    /* command key down */
    long menusel;

    if ( (menuSel = MenuKey(menukeys(keycode))) ) {
        if (myEvent.what == autoKey)
            autokeycommand = TRUE;
        else
            HiliteMenu(0);
        keyedmenu = TRUE;
        domenu(menuSel);
        keyedmenu = FALSE;
    }
    useselbottom = FALSE;
    break;
}

/* separate the real BS key from ^H */
if (thechar == BS)
    makebs = TRUE;
else
    makebs = FALSE;
if (controlmode && useselbottom) {
/*
 * If controlmode has been set, put out control chars
 * rather than making option-key extended chars.
 */
if (thechar == BS) {
    /* ^BS == DEL */
    thechar = DEL;
}
else if (makecontrol(&thechar, keycode)) {
    useselbottom = FALSE;
    break;
}
else if (!litinon) {
    /* don't allow these into handlekey */
    if (frontwind->type == VIWIND
        || frontwind->type == TEKWIND
        || !local) {
        /* all control chars echoed immediately */
        charout(thechar);
    }
    useselbottom = FALSE;
    break;
}
/* otherwise, let handlekey incorporate it */
}
handlekey(thechar, makebs);
useselbottom = FALSE;
break;
}
case activateEvt: {
    extern int DAreenum;
    extern long scraptote();

    SetPort((WindowPtr) myEvent.message);
    if (((WindowPeek)myEvent.message)->windowKind < userKind)
        break;
    if (DAreenum) {
        keyinit();
        scraptote();
        DAreenum = 0;
    }
    windactivate(myEvent.message, myEvent.modifiers & activeFlag);
    break;
}
case updateEvt: {

```

```

if (((WindowPeek)myEvent.message)->windowKind < userKind)
    break;
updatewind((WindowPtr) myEvent.message);
if (windswitch
    && ((WindowPtr) myEvent.message == outputwind->ptr) )
/* no more waiting for the new output window */
windswitch = FALSE;
break;
}
/* end switch */
}
}

#ifndef GROWTHEZONE
long
growout(thesize)
long thesize;
{
    errstr("Memory low--close window or *Delete* some text\r");
    return((long) 0) ;
}
#endif

#ifndef VRT
/* this vertical retrace routine is an attempt to deal with the
* speed problems caused by many small TEInsert() calls.
* when gatetick is set, the serial buffer is checked.
*/
pascal void
gatekeeper()
{
    #asm
OrigA5 EQU $904

    move.l  a5,a3
    move.l  (OrigA5),A5
    ; end of save routine
#endifasm

    gatetick = TRUE;
    gateVBL.vblCount = readlag > 0 ? (short) readlag : 1; /* 2, maybe more for networks with delays */
    /* setcursor may not be fast enough if gatetick is long */
#asm
    move.l  a3,a5
#endifasm

    return;
}
#endif

/* Makes the cursor an I-beam inside the front text window's view region */

setcursor()
{
    Point mouseloc;
    WindowPtr frontwind;
    struct window * thewind;
    int type;

    if ( ( (WindowPeek) (frontwind = FrontWindow()) )->windowKind >= userKind) {
        thewind = swindptr(frontwind);
        GetMouse(&mouseloc);
        if ((type = thewind->type) == VIWIND){
            if ( PtInRect(mouseloc, &virect) )
                SetCursor(*curvi);
            else
                SetCursor(&qd.arrow);
        }
#endifdef TEKEXISTS
        else if (type == TEKWIND){
            if ( PtInRect(mouseloc, &frontwind->portRect) )
                SetCursor(*curtek);
            else
                SetCursor(&qd.arrow);
        }
#endif
        else if (type == DOWNWIND) {
            SetCursor(&qd.arrow);
        }
        else if (type == TEXTWIND || type == BUGWIND) {

```

```

    if ( PtInRect(mouseloc, &(*thewind->hTE)->viewRect) )
        SetCursor(*curtext);
    else
        SetCursor(&qd.arrow);
}
}

#define LABELLEN 50

infobar()
{
    int windnum;
    Point drawloc;
    GrafPtr saveport;
    Rect eraser;

    Str255 telength; /* length of TE text, < 6 chars */
    char line[LABELLEN], *lineptr;
    int count;
    char redirsym;
    struct window *frontwind;
    char inwind;
    char outwind;

    extern char useTEKey; /* delete selection behavior */
    WindowPtr topwind;

    topwind = FrontWindow();
    if (((WindowPeek) topwind)->windowKind) < userKind)
        return;

    frontwind = swindptr(topwind);
    windnum = frontwind - window;
    GetPort(&saveport);

    SetPort(frontwind->ptr);

    /* number the window */
    drawloc.h = 11;
    drawloc.v = -6;
    LocalToGlobal(&drawloc);

    /* draw strings giving state of menu switches and top window */
    lineptr = line;

    inwind = inputwind - window + '0';
    outwind = outputwind - window + '0';

    *lineptr++ = (inwind == ';' ? 'S' : inwind);
    *lineptr++ = '>';
    *lineptr++ = (outwind == ';' ? 'S' : outwind);

    *lineptr++ = ' ';
    *lineptr++ = (logmode ? 'R' : ' ');
    *lineptr++ = (local ? 'E' : ' ');
    *lineptr++ = (skipprompt ? 'P' : ' ');

    *lineptr++ = ' ';
    *lineptr++ = ((makeinput && frontwind->cstate & WS_INPUT) ? 'I' : ' ');
    /* this window is only temporarily the input window if !INPUT */
    *lineptr++ = ((frontwind->cstate & WS_OUTPUT) ? 'O' : ' ');
    *lineptr++ = ' ';

    /* get size of TE text */
    NumToString( (long) (*frontwind->hTE)->teLength, telength);

    eraser.top = frontwind->ptr->portRect.top - 14;
    eraser.left = frontwind->ptr->portRect.left + 20;
    eraser.bottom = eraser.top + 9;
    eraser.right = eraser.left + 72 + 18 + 30;

    LocalToGlobal((struct Point *)(&eraser.top));
    LocalToGlobal((struct Point *)(&eraser.bottom));

    /* now do all the drawing */
    /* fix clip so no drawing in menu bar */
    getwport(20);

    MoveTo(drawloc.h, drawloc.v);
    if (windnum == BUGWINDNUM) {
        /* ScratchPad */
        DrawChar('S');
    }
}

```

```

else if (windnum < VIWINDNUM)
    DrawChar( (char) (windnum + 48) );
    /* draw window number in go-away */
else
    DrawChar(' ');

EraseRect(&eraser);
Move(6, 0);
DrawText(line, 0, lineptr - line);
DrawString(telength);
if (frontwind->dirty)
    DrawChar('*');
else
    DrawChar(' ');
if ((*frontwind->hTE)->crOnly)
    DrawChar('\015');
else
    DrawChar(' ');
if (useTEKey)
    DrawChar('X');

retwport();
SetPort(saveport);
}

/* do the serial business */

doserial()
{
    extern int fcapture;
    extern long captime;
    extern struct virecord vr;

    /* receive */
    if (modem.in) {
        /* modem port has been opened */
        SerGetBuf(modem.in, &nbytes);
        if (doflow && noserd) {
            /* do xon/xoff manually */
            if (xoff) {
                /* is there enough room to resume? */
                if (nbytes < TURNON) {
                    charout(XON);
                    xoff = FALSE;
                }
            }
            else {
                if (nbytes > SHUTDOWN) {
                    charout(XOFF);
                    xoff = TRUE;
                }
            }
        }
        /* if no growth or a fair # bytes accumulated, allow write */
        if (nbytes) {
            if ((nbytes > blockNBytes || nbytes == onbytes) && !windswitch) {
                /* write it out */
                if (!noserd || nbytes == onbytes)
                    onbytes = nbytes - portRead(modem.in, nbytes);
                else
                    onbytes = nbytes - portRead(modem.in, blockNBytes);
                /* limit number of bytes read so slow TE output won't
                   cause late sending of XOFF */

                /* TODO should look through list of viwindows? */
                if (vr.vitouched)
                    updviscreen(viwind);
                if (fcapture && (TickCount() > (long) (captime + captimeout) ) ) {
                    extern struct window * capturewind;

                    errstr("Capture timed out\r");
                    endcapture(capturewind);
                }
            }
            else {
                /* wait */
                onbytes = nbytes;
            }
        }
    }

    /* send */
    if (modem.transon) {
        /* if we're sending text to host, keep it up */

```

```

#define ASYNCWRITE
    if (modem.transcont) {
        static long lagtill;

        /* continue flag raised */
        if (TickCount() > lagtill) {
            /* to substitute for output xon/xoff, wait a user-specified while */
            lagtill = TickCount() + writelag;
            if (endline(&modem)) {
                sendline(&modem);
            }
        }
    }
#else
    sendline(&modem);
#endif
}

/* ensures that long user actions such as moving text w/ a scrollbar will
   not cause buffer overflow */

turnxoff()
{
    if (noserd && doflow && modem.in && !xoff) {
        charout(XOFF);
        xoff = TRUE;
    }
}

RgnHandle oldwclip;
GrafPtr prewport;

getwport(cliptop)
int cliptop;
{
    Rect widerect;
    short fontnum;

    GetPort(&prewport);
    SetPort(wport);      /* window mgr port */
    oldwclip = NewRgn();
    GetClip(oldwclip);
    widerect.top = cliptop;
    widerect.left = 0;
    widerect.bottom = 32000;
    widerect.right = 32000;
    ClipRect(&widerect);

    GetFNum("\PMona", &fontnum );
    TextFont(fontnum);
    TextSize(9);
}

retwport()
{
    /* reset old font/size */
    TextFont(systemFont);
    TextSize(12);

    SetClip(oldwclip);
    DisposeRgn(oldwclip);
    SetPort(prewport);
}

makecontrol(thechar, keycode)
char * thechar;
char keycode;
{
    if (keycode < MAXKEYCODE - 1) {
        if (Optkeymap[keycode] != BADKEY) {
            /* if good match */
            *thechar = optkeymap[keycode];
            return(0);
        }
    }
    return(-1);
}

RgnHandle oldclip;
GrafPtr oldclport;

```

```

closeclip(thewind)
struct window * thewind;
{
    GetPort(&oldclport);
    SetPort(thewind->ptr);
    oldclip = NewRgn();
    GetClip(oldclip);
    ClipRect(&nullrect);
}

openclip()
{
    SetClip(oldclip);
    DisposeRgn(oldclip);
    SetPort(oldclport);
}

selwind(windp)
WindowPtr windp;
{
    struct window * thewind;
    thewind = swindptr(windp);
/*
    if (thewind->type == TEXTWIND || thewind->type == BUGWIND) {
*/
        /* fiddle with input and output so mouse selections are the same as menu selections */
        blankiomarkers();
        if (thewind->cstate & WS_INPUT)
            inputwind = thewind;
        if (!uwon && thewind->cstate & WS_OUTPUT) {
            outputwind = thewind;
            MoveHHi((Handle) thewind->hTE);
            MoveHHi((*thewind->hTE)->hText);
            /* move the text record and contents to top of heap so they expand better */
        }
        setiomarkers(thewind);

        /* take care of setting the REP switches to what the window had before */
        if (thewind->cstate & WS_RECORD) {
            /* turn record mode on */
            logmode = TRUE;
        }
        else {
            logmode = FALSE;
        }
        if (thewind->cstate & WS_LOCAL) {
            /* turn local edit mode on */
            local = litinon = TRUE;
        }
        else {
            local = litinon = FALSE;
        }
        if (thewind->cstate & WS_PROMPT) {
            /* turn prompt skip mode on */
            skipprompt = TRUE;
        }
        else {
            skipprompt = FALSE;
        }
/*
    }
*/
    SelectWindow(windp);
}

/* return handle to macro string or NULL */

Handle macromatch()
{
    return((Handle) NULL);
}

/* takes a keycode and returns a reasonable key value (no extended char nonsense) */

#define MKEYCODES 93
char menuchars[MKEYCODES] = {
'a',
's',
'd',
'f',
'h',
'g',
}

```

```
'z',
'x',
'c',
've',
'\000', /* 10 */
'b',
'q',
'w',
'e',
'r',
'y',
't',
'1',
'2',
'3', /* 20 */
'4',
'6',
'5',
'=',
'9',
'7',
'-',
'8',
'0',
']', /* 30 */
'o',
'u',
[',
'i',
'p',
'\015',
'l',
'j',
'\'',
'k', /* 40 */
';',
'\\',
',',
'/',
'n',
'm',
'.',
'\011',
'`', /* 50 */
'\010',
'\000',
'\033',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'\000',
'+',
'+', /* 60 */
'\033',
',',
'\000',
'\000',
'/' ,
'\033',
'/' ,
'-',
'\000',
'\000', /* 80 */
'=',
'0',
'1',
'2',
'3',
'4',
'5',
'6',
'7',
'\000', /* 90 */
'8',
'9'
```

```

};

char menukeys(keycode)
char keycode;
{
    if (keycode < MKEYCODES - 1) {
        return(menuchars[keycode]);
    }
    return('\000');
}

/* check the keycode to see if early interpretation required
   returns 0 if handled here
*/
keyswitch(keycode, keychar)
char keycode;
char keychar;
{
    if (keycode == 52 || keycode == 76) {
        /* Enter key interpretation happens early -- no menu poss. */
        /* Enter is always interpreted, no matter the mode */
        /* This allows convenient use of Local Edit mode */
        if (myEvent.modifiers & shiftKey) {
            sendselect();
            return(0);
        }
        if (useselbottom) {
            /* Send the whole window */
            tesetsel((long) 0, (long) (*inputwind->hTE)->teLength, inputwind->hTE);
            sendselect();
            return(0);
        }
        /* TODO and why not sendselect() the output window or the printer? */
        else
            /* make it something useful */
            return(ESC);
    }

    if (outputwind->type == TEXTWIND || outputwind->type == BUGWIND) {
        switch (keycode) {
            /* left, right, up, down arrow keys */
            case 123:
            case 69:
            case 70: {
                /* '+' key */
                if (keychar == 0x1c) {
                    /* left arrow key on mac+ */
                    cursormenu(LEFT);
                    return(0);
                }
            }
            case 124:
            case 67:
            case 66: {
                /* '*' key */
                if (keychar == 0x1d) {
                    /* right arrow key on mac+ */
                    cursormenu(RIGHT);
                    return(0);
                }
            }
            case 126:
            case 75:
            case 77: {
                /* '/' key */
                if (keychar == 0x1e) {
                    /* up arrow key on mac+ */
                    cursormenu(UP);
                    return(0);
                }
            }
            case 125:
            case 81:
            case 72: {
                /* '=' key */
                if (keychar == 0x1f) {
                    /* down arrow key on mac+ */
                    cursormenu(DOWN);
                    return(0);
                }
            }
        }
    }
    else if (outputwind->type == VIWIND) {
}
}

```

```

/* send appropriate cursor motion sequences */
switch (keycode) {
    /* left, right, up, down arrow keys */
    case 123:
    case 69:
    case 70: {
        /* '+' key */
        if (keychar == 0x1c) {
            /* left arrow key on mac+ */
            mvleft();
            return(0);
        }
    }
    case 124:
    case 67:
    case 66: {
        /* '*' key */
        if (keychar == 0x1d) {
            /* right arrow key on mac+ */
            mvright();
            return(0);
        }
    }
    case 126:
    case 75:
    case 77: {
        /* '/' key */
        if (keychar == 0x1e) {
            /* up arrow key on mac+ */
            mvup();
            return(0);
        }
    }
    case 125:
    case 81:
    case 72: {
        /* '=' key */
        if (keychar == 0x1f) {
            /* down arrow key on mac+ */
            mvdown();
            return(0);
        }
    }
}
return(keychar);
}

windactivate(windptr, on)
WindowPtr windptr;
int on;
{
    struct window * thewind;
    int windnum;
    extern setsizemenu(), setfontmenu(), settopmenu();
    /* in menu.size.c, menu.c, menu.top.c */

    thewind = swindptr(windptr);
    if (on) {
#ifdef UW
        /* activate window for uw */
        if (uwon) {
            int windnum;

            windnum = thewind - window;
            if ((thewind->cstate & WS_UW) && windnum > 0 && windnum <= NWINDOW) {
                uwcmd(CB_FN_ISELW, windnum);
            }
#endif
        /* TODO MVI copy old vi vars into previous window,
           copy new windows vars into var block */

        VIsize(thewind, &virect);
        infobar();

        switch (thewind->type) {
            case BUGWIND: {
                static struct window * oldinput;
                extern long anchor;

                /* cursormenu(ANCHOR); */
                /* fall through to complete activation */
            }
            case TEXTWIND: {
                DrawGrowIcon(windptr);
            }
        }
    }
}

```

```
anchor = -1;
/* setfontmenu((*thewind->hTE)->txFont ); */
setsizemenu( (int) (*thewind->hTE)->txFont,
              (int) (*thewind->hTE)->txSize);
settopmenu(thewind);

TEActivate(thewind->hTE);
ShowControl(thewind->vscroll);
ShowControl(thewind->hscroll);
break;
}

case VIWIND: {
    viactivate(thewind);
    ShowControl(thewind->hscroll);
    /* setfontmenu( ((WindowPeek)(thewind->ptr))->txFont ); */
    setsizemenu( (int) thewind->ptr->txFont,
                  (int) thewind->ptr->txSize );
    break;
}

#endif TEKEXISTS
case TEKWIND: {
    DrawGrowIcon(windptr);
    setsizemenu( (int) thewind->ptr->txFont,
                  (int) thewind->ptr->txSize );
    tekactivate();
    break;
}
#endif
}

else {
    /* window is being deactivated */
    switch (thewind->type) {
        case BUGWIND: {
            /* fall through to complete deactivation */
        }
        case TEXTWIND: {
            DrawGrowIcon(windptr); */
            TEDeactivate(thewind->hTE);
            HideControl(thewind->vscroll);
            HideControl(thewind->hscroll);
            break;
        }
        case VIWIND: {
            HideControl(thewind->hscroll);
            videactivate(thewind);
            break;
        }
    #ifdef TEKEXISTS
        case TEKWIND: {
            tekdeactivate();
            break;
        }
    #endif
    }
}
}
```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "menudefs.h"
#include "windmenudefs.h"
#include "commmenudefs.h"
#include "echomenudefs.h"
#include "startup.h"
#include "mac.h"
#include "enccopynotice"

#define swindptr(A) ((struct window *) ((WindowPeek) A)->refCon)

extern longtoa();

#define KEVICON 256
#define ABOUT 1
#define ALLCONTROL 2
#define SKIPAPPLE 3

MenuHandle menu[NUMMENU];
int DAreftnum;

extern scrollTE(), resetspeed();
extern filemenu(), editmenu(), cursormenu(), windmenu(), echomenu(), commmenu();
extern sizemenu(), topmenu();

extern char allcontrol;

domenu(menuresult)
long menuresult;
{
    int menu, item;

    menu = menuresult >> 16;
    item = menuresult & 0xffff;
    docommand(menu, item);
    return;
}

docommand(theMenu, theItem)
{
    extern char autokeycommand; /* kludge for avoiding DrawControls */
    extern char clearscreen;

    switch (theMenu) {
        case appleMenu: {
            switch (theItem) {
                case ABOUT: {
                    errstr(copynotice);

                    /* causes redrawing?
                     Rect plotrect;

                     plotrect.top = 220;
                     plotrect.left = 420;
                     plotrect.bottom = 236;
                     plotrect.right = 436;
                     PlotIcon(&plotrect, GetIcon(KEVICON));
                    */
                    break;
                }
                case ALLCONTROL: {
                    if (allcontrol) {
                        allcontrol = FALSE;
                    }
                    else {
                        allcontrol = TRUE;
                    }
                    invertmenu();
                    CheckItem(menu[APPLE], theItem, (Boolean) allcontrol);
                    break;
                }
                case SKIPAPPLE: {
                    break;
                }
                default: {
                    Str255 DName;

                    HLock(TEScrHandle);
                    tetoscrap(*TEScrHandle, TEScrLen);
                    HUnlock(TEScrHandle);
                    GetItem(menu[APPLE], theItem, DName);
                    keyclose();
                    DAreftnum = OpenDeskAcc(DName);
                }
            }
        }
    }
}
```

```

        }
    }
break;
}
case fileMenu: {
    filmenu(theItem);
break;
}
case editMenu: {
    editmenu(theItem);
break;
}
case cursorMenu: {
    cursormenu(theItem);
break;
}
case windMenu: {
    windmenu(theItem);
break;
}
case echoMenu: {
    echomenu(theItem);
break;
}
case commMenu: {
    commmenu(theItem);
break;
}
case fontMenu: {
    /* change font in front window */
    struct window * thewind;      /* front window record */
    short fontnum;
    Str255 name;

    if ( ((WindowPeek) FrontWindow())->windowKind < userKind )
        /* don't mess with system windows! */
        break;
    thewind = swindptr(FrontWindow());
    GetItem(menu[FONT], theItem, name);
    GetFNum(name, &fontnum);
    setfont(thewind, fontnum);
    break;
}
case sizeMenu: {
    sizemenu(theItem);
break;
}
case topMenu: {
    topmenu(theItem);
break;
}
}

setfont(thewind, fontnum)
struct window * thewind;
int fontnum;
{
    int size;

    switch (thewind->type) {
        case BUGWIND:
        case TEXTWIND: {
            (*thewind->hTE)->txFont = fontnum;
            size = (*thewind->hTE)->txSize;
            break;
        }
        case TEKWIND:
        case VIWIND: {
            thewind->ptr->txFont = fontnum;
            size = thewind->ptr->txSize;
            break;
        }
    }
    if (setsizemenu(fontnum, size))
        setfontsize(thewind, (short) size);
}
}

```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "comm.h"
#include "commdefs.h"
#include "windmenu.h"
#include "echomenudefs.h"
#include "commmenudefs.h"
#include "windmenudefs.h"
#include "startup.h"

int newspeed;
int newparity;
int newstop;
int newdata;

extern long blockNBytes;           /* sets TEInsert wait-for-size */

extern struct window * makewindow();

extern char * serconf;

commmenu(theItem)
int theItem;
{
    newspeed = defport->speed;
    newparity = defport->parity;
    newstop = defport->stop;
    newdata = defport->data;

    switch (theItem) {
        extern portclose(), portinit();

        case MODEMOFF: {
            if (defport == &modem) {
                defport = &printer;
            }
            else {
                defport = &modem;
            }
            break;
        }
        case MODEIMPORT:
        case PRINTERPORT: {
            extern char local;

            if (defport->in) {
                if (theItem == portmenuid()) {
                    /* turn it off and leave! */
                    portclose(defport);
                    resetcomm();
                    break;
                }
                else
                    portclose(defport);
                    /* close the one, open the other one */
            }
            if (portinit(theItem, defport)) {
                errstr("no port open\n");
                break;
            }
        }
#define OLDOLD
        if (viwind == NULL && defport == &modem) {
            struct window * newwind;

            /* make a vi window */
            newwind = &window[VIWINDNUM]; /* TODO parameterize */
            if (newwind->ptr == NULL) {
                newwind = makewindow(VIWINDNUM, VIWIND, defvifont, defvisize, 0, 0, 0, 0);
                ShowWindow(newwind->ptr);
            }

            /* make window 1 and shuffle around window 0 */
            blankiomarkers();
            outputwind = makewindow(1, TEXTWIND, deffont, defsize, 0, 0, 0, 0);
            if (outputwind == NULL) {
                killwindow(newwind);
                break;
            }
            oldout = WINDMBASE + 1;
            setiomarkers();
            ShowWindow(outputwind->ptr);

            sizewindow(window[0].ptr, 90, screenBits.bounds.right + 8);
        }
    }
}

```

```

    if (window[0].cstate & WS_LOCAL)
        docommand(echoMenu, LOCAL);
        /* leave local edit mode */
    if (!(window[0].cstate & WS_RECORD))
        docommand(echoMenu, LOGMODE);
        /* put inputwind in record mode */
    if (!(window[0].cstate & WS_INPUT))
        docommand(echoMenu, INPUT);
        /* make it the input window */
        selwind(window[0].ptr);
    }

#endif
    break;
}
case B57600: {
    newspeed = baud57600;
    break;
}
case B19200: {
    newspeed = baud19200;
    break;
}
case B9600: {
    newspeed = baud9600;
    break;
}
case B4800: {
    newspeed = baud4800;
    break;
}
case B2400: {
    newspeed = baud2400;
    break;
}
case B1200: {
    newspeed = baud1200;
    break;
}
case B300: {
    newspeed = baud300;
    break;
}
case NOPARITY: {
    newparity = noParity;
    break;
}
case EVENPARITY: {
    newparity = evenParity;
    break;
}
case ODDPARITY: {
    newparity = oddParity;
    break;
}
case ONESTOP: {
    newstop = stop10;
    break;
}
case ONEHALFSTOP: {
    newstop = stop15;
    break;
}
case TWOSTOP: {
    newstop = stop20;
    break;
}
case SIXDATA: {
    extern char mask;

    mask = 0x3f;
    newdata = data6;
    break;
}
case SEVENDATA: {
    extern char mask;

    mask = 0x7f;
    newdata = data7;
    break;
}
case EIGHTDATA: {
    extern char mask;

    mask = 0xff;
    newdata = data8;
    break;
}
}

```

```

}

if (theItem >= B57600) {
    setCW();
    setblock();
    /* only good for speed reset, but what the hell */
}
resetcomm();
}

setCW()
{
    int config;

    config = newspeed | newdata | newparity | newstop;
    if (defport->in) {
        if (SerReset(defport->in, config) || SerReset(defport->out, config)) {
            errstr(serconf);
            return(-1);
            /* bomb out if modem is on and the setting fails */
        }
    }
    defport->speed = newspeed;
    defport->data = newdata;
    defport->parity = newparity;
    defport->stop = newstop;
    return(0);
}

resetcomm()
{
    int count;
    int oldspeed;
    int oldparity;
    int oldstop;
    int olldata;

    /* blank all slots in menu */
    for (count = LASTCOMM; --count; )
        CheckItem(menu[COMM], count, FALSE);

    /* set to current values */
    SetItem(menu[COMM], MODEMOFF,
        (defport == &modem) ? "\PSet Modem" : "\PSet Printer");
    if (defport->in)
        CheckItem(menu[COMM], portmenuid(), TRUE);

    switch (defport->speed) {
        case baud57600: {
            oldspeed = B57600;
            break;
        }
        case baud19200: {
            oldspeed = B19200;
            break;
        }
        case baud9600: {
            oldspeed = B9600;
            break;
        }
        case baud4800: {
            oldspeed = B4800;
            break;
        }
        case baud2400: {
            oldspeed = B2400;
            break;
        }
        case baud1200: {
            oldspeed = B1200;
            break;
        }
        case baud300: {
            oldspeed = B300;
            break;
        }
    }
    CheckItem(menu[COMM], oldspeed, TRUE);
    switch (defport->data) {
        case data8: {
            olldata = EIGHTDATA;
            break;
        }
        case data7: {
            olldata = SEVENDATA;
            break;
        }
    }
}

```

```

    }
    case data6: {
        olddata = SIXDATA;
        break;
    }
}
CheckItem(menu[COMM], olddata, TRUE);
switch (defport->parity) {
    case noParity: {
        oldparity = NOPARITY;
        break;
    }
    case oddParity: {
        oldparity = ODDPARITY;
        break;
    }
    case evenParity: {
        oldparity = EVENPARITY;
        break;
    }
}
CheckItem(menu[COMM], oldparity, TRUE);
switch ((unsigned) defport->stop) {
    case stop10: {
        oldstop = ONESTOP;
        break;
    }
    case stop15: {
        oldstop = ONEHALFSTOP;
        break;
    }
    case stop20: {
        oldstop = TWOSTOP;
        break;
    }
}
CheckItem(menu[COMM], oldstop, TRUE);
}

setblock()
{
    extern struct Port modem;
    extern int fcapture;

    switch (modem.speed) {
        case baud57600: {
            blockNBytes = 1000;
            break;
        }
        case baud19200: {
            blockNBytes = 1000;
            break;
        }
        case baud9600: {
            blockNBytes = 1000;
            break;
        }
        case baud4800: {
            blockNBytes = 500;
            break;
        }
        case baud2400: {
            blockNBytes = 240;
            break;
        }
        case baud1200: {
            blockNBytes = 120;
            break;
        }
        case baud300: {
            blockNBytes = 60;
            break;
        }
    }
    if (noserd && !vion && !switchnl && blockNBytes > 400) {
        /* slow TE mode, make sure xon/xoff gets attention */
        blockNBytes = 400;
    }
    if (vion)
        blockNBytes = 400;
}

portmenuid()
{
    if (defport->in == -6)

```

12/7/94 10:28

menu.comm.c

5

```
    return(MODEMPORT);
else if (defport->in == -8)
    return(PRINTERPORT);
else
    return(0);
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "comm.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "commdefs.h"
#include "echomenudefs.h"
#include "windmenudefs.h"
#include "uw.h"
#include "vi.h"

long buildtime, drawtime;

#ifndef VIDUMP
char resetstr[] = "\033[2J\033[H";
char teststring[] = " The IBM-mode macro processing routines allow you to produce a sequence of \r\n\
\"actions\" when you press a key-combination (such as Command Option A). \r\n\
Individual characters stand for themselves (except for '!' & '\\', which can be \r\n\
entered by using '!!' and '\\\\'). The Return key will be mapped to the Enter \r\n\
key automatically. Other actions are described by an exclamation point \r\n\
followed by a pair of characters; the first character selects a class of \r\n\
action, and the second the particular type of this action. (A three-digit \r\n\
decimal number preceded by '\' can be used in place of a character). The \r\n\
action classes and associated characters available in IBM mode are as follows: \r\n\
The IBM-mode macro processing routines allow you to produce a sequence of \r\n\
\"actions\" when you press a key-combination (such as Command Option A). \r\n\
Individual characters stand for themselves (except for '!' & '\\', which can be \r\n\
entered by using '!!' and '\\\\'). The Return key will be mapped to the Enter \r\n\
key automatically. Other actions are described by an exclamation point \r\n\
followed by a pair of characters; the first character selects a class of \r\n\
action, and the second the particular type of this action. (A three-digit \r\n\
decimal number preceded by '\' can be used in place of a character). The \r\n\
action classes and associated characters available in IBM mode are as follows: \r\n\
The IBM-mode macro processing routines allow you to produce a sequence of \r\n\
\"actions\" when you press a key-combination (such as Command Option A). \r\n\
Individual characters stand for themselves (except for '!' & '\\', which can be \r\n\
entered by using '!!' and '\\\\'). The Return key will be mapped to the Enter \r\n\
key automatically. Other actions are described by an exclamation point \r\n\
followed by a pair of characters; the first character selects a class of ";
#endif

extern char switchnl;
extern char controlmode;
extern char bstodel;

echomenu(theItem)
int theItem;
{
    struct window * thewind;
    WindowPtr frontwind;

    frontwind = FrontWindow();
    if (((WindowPeek) frontwind)->windowKind < userKind)
        return(-1);

    thewind = swindptr(frontwind);
    switch (theItem) {
        case LOGMODE: {
            if (logmode) {
                logmode = FALSE;
                thewind->cstate &= ~WS_RECORD;
                /* skipprompt = TRUE; */
            }
            else {
                logmode = TRUE;
                thewind->cstate |= WS_RECORD;
                /* skipprompt = FALSE; */
            }
            /* CheckItem(menu[MECHO], LOGMODE, (Boolean) logmode); */
            /* CheckItem(menu[MECHO], SKIPPROM, (Boolean) skipprompt); */
            infobar(); /* update info on top wind drag */
            break;
        }
        case LOCAL: {
            extern char litinon;

            if (local) {
                local = FALSE;
                thewind->cstate &= ~WS_LOCAL;
            }
            else {
                local = TRUE;
            }
        }
    }
}

#ifndef LITINMENU
litinon = FALSE;
#endif

```

```

        thewind->cstate |= WS_LOCAL;
#ifndef LITINMENU
    litinon = TRUE;
#endif
}
/* CheckItem(menu[MECHO], LOCAL, (Boolean) local); */
infobar(); /* update info on top wind drag */
break;
}
case SKIPROM: {
    if (skipprompt) {
        skipprompt = FALSE;
        thewind->cstate &= ~WS_PROMPT;
    }
    else {
        skipprompt = TRUE;
        thewind->cstate |= WS_PROMPT;
    }
/* CheckItem(menu[MECHO], SKIPROM, (Boolean) skipprompt); */
infobar(); /* update info on top wind drag */
break;
}
case BREAK: {
    long ticks;
    char nothing;

#ifdef OLD
    SerSetBrk(modem.out);
    Delay((long) 40, &ticks);
    /* TODO this bombs out for some reason */
    SerClrBrk(modem.out);
#else
    Control(modem.out, 12, &nothing); /* set break */
    Delay((long) 60, &ticks);
    Control(modem.out, 11, &nothing); /* reset break */
#endif
    break;
}
case BSTODEL: {
    if (bstodel)
        bstodel = FALSE;
    else
        bstodel = TRUE;
    CheckItem(menu[MECHO], BSTODEL, (Boolean) bstodel);
    break;
}
case DECESC: {
    if (decesc)
        decesc = FALSE;
    else
        decesc = TRUE;
    CheckItem(menu[MECHO], DECESC, (Boolean) decesc);
    break;
}
case SWITCHNL: {
    extern char putskip1;

    if (switchnl) {
        switchnl = FALSE;
        /* setting NOSKIP fixes linecount problems */
        putskip1 = NOSKIP;
    }
    else {
        switchnl = TRUE;
        putskip1 = CR;
    }
    setblock();
    /* fix blocking factor */
    CheckItem(menu[MECHO], SWITCHNL, (Boolean) switchnl);
    break;
}
#endif LITINMENU
case LITIN: {
    extern char litinon;

    if (litinon)
        litinon = FALSE;
    else
        litinon = TRUE;
    CheckItem(menu[MECHO], LITIN, (Boolean) litinon);
    break;
}
case LITOUT: {
    extern char literal;

    if (literal)

```

```

        literal = FALSE;
    else
        literal = TRUE;
    CheckItem(menu[MECHO], LITOUT, (Boolean) literal);
    break;
}

#ifndef UW
case UWTOGGLE: {
/* do uw stuff */
if (uwon) {
    uwcmd(CB_FN_MAINT, CB_MF_EXIT);
    EnableItem(menu[WIND], OUTPUT);
}
if (uwon)
    uwon = FALSE;
else
    uwon = TRUE;
CheckItem(menu[MECHO], UWTOGGLE, (Boolean) uwon);
break;
}
#endif

case SETINLAG: {
extern int readlag;

ParamText("\PSet input lag to:", "\Pticks", "\P", "\P");
readlag = dosetnumdial(readlag);
break;
}
case SETOUTLAG: {
extern int writelag;

ParamText("\PSet output lag to:", "\Pticks", "\P", "\P");
writelag = dosetnumdial(writelag);
break;
}
case SWITCHLOCK: {
if (controlmode)
    controlmode = FALSE;
else
    controlmode = TRUE;
CheckItem(menu[MECHO], SWITCHLOCK, (Boolean) controlmode);
break;
}
case WRAPOUT: {
extern int wrapwidth;

ParamText("\PDo output wrap at:", "\Pcolumns", "\P", "\P");
wrapwidth = dosetnumdial(wrapwidth);
break;
}
case SETVIWRAP: {
ParamText("\PSet terminal wrap at:", "\Ppixels", "\P", "\P");
vr.viwrap = dosetnumdial(vr.viwrap);
break;
}
case DOFLOW: {
extern char doflow;

if (doflow)
    doflow = FALSE;
else doflow = TRUE;
if (!noserd)
    setshake(modem.in, modem.out, (int) doflow, (int) doflow);
CheckItem(menu[MECHO], DOFLOW, (Boolean) doflow);
break;
}

#ifndef VIDUMP
case VIDUMP: {
long testlen;
extern char * readbuf;
extern char * stake;

stake = readbuf;

testlen = (long) strlen(resetstr);
BlockMove(resetstr, readbuf, testlen);
vioutput((int) testlen);
updviscreen(viwind);

testlen = (long) strlen(teststring);
BlockMove(teststring, readbuf, testlen);

buildtime = TickCount();
stake = readbuf;
vioutput((int) testlen);
buildtime = TickCount() - buildtime;
}
#endif

```

```
drawtime = TickCount();
updviscreen(viwind);
drawtime = TickCount() - drawtime;

sprintf(error, "build -- %ld, draw -- %ld\r", buildtime, drawtime);
errstr(error);
break;
}

#endif
}
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "videfs.h"
#include "vi.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "editmenudefs.h"
#include "cursormenudefs.h"
#include "mac.h"

char * unimplemented = "unimplemented\r";
char * ramout = "out of RAM\r";
extern char * textfull;
extern struct virecord vr;

int cutsize;           /* size of current cut/copy attempt */

#define TAGNUM 'z' - 'a' + 1 + 1
/* + 1 for extra undo tag */
struct texttags {
    struct window * tagwind;
    short tagstart;
    short tagend;
} texttag[TAGNUM];
/* + 1 for undo tag */
#define UNDOTAG (TAGNUM - 1)

/* the previous selection and the newly-pasted selection for undoing paste */
struct window * unwind;

int lastedop; /* last undo-relevant editing operation */
Handle unhandle; /* handle for undo text */
int uncutlen; /* length of the undo text */
Handle cuthandle; /* handle for holding all cut text */

long anchor = -1;

extern char useselbottom;

editmenu(theItem)
int theItem;
{
    register TEHandle toptexthand; /* top window text record */
    register char * TPos;
    register char * TBase;
    register int start;
    register int selsize;
    struct window * thewind; /* front window record */
    int selstart;
    char tabdelete;

    tabdelete = FALSE;
    if (SystemEdit(theItem - 1))
        return(0);
    if ( ((WindowPeek) FrontWindow())->windowKind >= userKind ) {
        thewind = swindptr(FrontWindow());
        if (thewind->type == TEXTWIND || thewind->type == BUGWIND) {
            toptexthand = thewind->hTE;
            selstart = (*toptexthand)->selStart;
            cutsize = (*toptexthand)->selEnd - selstart;
        }
    }
    if (myEvent.what == autoKey && theItem < DELRIGHT)
        /* don't accept autokeying in this menu on cut/paste ops */
        return(-1);

    switch (theItem) {
        /* the active editing commands apply to the top window;
         * the motion commands to the input window
         * perhaps these should be put under separate menu headings
         */

        case UNDO: {
#ifdef USEUNDO
            int uncutstart; /* start of cut text */

            uncutstart = texttag[UNDOTAG].tagstart;
            if (!unwind) {
                errprompt("nothing to undo");
                break;
            }
            if (unwind->ptr != FrontWindow())
                selwind(unwind->ptr);
        }
        switch (lastedop) {

```

```

case CUTINS:
case CUT: {
    tesetsel((long) uncutstart, (long) uncutstart, unwind->hTE);
    tagadjust(unwind, TADD, uncutstart, TEScrLen);
    texttag[UNDOTAG].tagstart = uncutstart;
    texttag[UNDOTAG].tagend = uncutstart + TEScrLen;

    TEPaste(unwind->hTE);
    if (lastedop == CUTINS) {
        /* delete the character that deleted the selection
         * don't bother to restore char on undo */
        tesetsel((long) (uncutstart + TEScrLen), (long) (uncutstart + TEScrLen + 1), unwind->hTE);
        TEDelete(unwind->hTE);
    }
    tesetsel((long) uncutstart, (long) (uncutstart + TEScrLen), unwind->hTE);

    /* restore the old scrap */
    if (unhandle != (Handle) NULL) {
        DisposHandle(TEScrHandle);
        TEScrHandle = unhandle;
        HandToHand(&TEScrHandle);
        TEScrLen = uncutlen;
        lastedop = UNDOCUT;
    }
    break;
}
case COPY: {
    /* restore the old scrap */
    if (unhandle != (Handle) NULL) {
        DisposHandle(TEScrHandle);
        TEScrHandle = unhandle;
        HandToHand(&TEScrHandle);
        TEScrLen = uncutlen;
        errprompt("previous scrap restored");
        lastedop = UNDOCOPY;
    }
    break;
}
case PASTE: {
    int tagsize;
    Handle htemp;

    tagsize = texttag[UNDOTAG].tagend - uncutstart;

    /* kill the pasted text, saving a copy */
    tesetsel((long) uncutstart, (long) texttag[UNDOTAG].tagend, unwind->hTE);
    PtrToHand((Ptr) (*(*toptexthand)->hText + selstart), &htemp, (long) tagsize);
    TEDelete(unwind->hTE);

    tagadjust(unwind, TCUT, uncutstart, tagsize);
    tagadjust(unwind, TADD, uncutstart, uncutlen);
    texttag[UNDOTAG].tagstart = uncutstart;
    texttag[UNDOTAG].tagend = uncutstart + uncutlen;

    /* paste in the text that was killed and select it */
    TEInsert(unhandle, (long) uncutlen, unwind->hTE);
    tesetsel((long) uncutstart, (long) (uncutstart + uncutlen), unwind->hTE);

    /* save the copy into unhandle */
    DisposHandle(unhandle);
    unhandle = htemp;
    HandToHand(&unhandle);
    uncutlen = tagsize;
    DisposHandle(htemp);
    lastedop = UNDOPASTE;
    break;
}
case UNDOCUT: {
    tesetsel((long) uncutstart, (long) texttag[UNDOTAG].tagend, unwind->hTE);
    editmenu(CUT);
    break;
}
case UNDOCOPY: {
    break;
}
case UNDOPASTE: {
    tesetsel((long) uncutstart, (long) texttag[UNDOTAG].tagend, unwind->hTE);
    editmenu(PASTE);
    break;
}
}

#ifndef __CINT__
#endif
#endif
errstr(unimplemented);
SetCtlMax(unwind->vscroll, newvctlmax(thewind));
seekselect(unwind);
break;

```

```

}

case CUT:
case CUTINS:
case CLEAR: {
    if (theItem == CUT || theItem == CUTINS) {
        /* check memory space here */
        if (cutsize && enoughmem(cutsize + TEScrLen - uncutlen)) {
            tagadjust(thewind, TCUT, selstart, cutsize);
    }
#endif ANCHORHIGH
    anchor = -1;
#endif
#ifndef USEUNDO
    /* take care of undo */
    lastedop = theItem;
    texttag[UNDOTAG].tagwind = unwind = thewind;
    texttag[UNDOTAG].tagstart = texttag[UNDOTAG].tagend = selstart;

    if (unhandle != (Handle) NULL)
        DisposHandle(unhandle);
    unhandle = TEScrHandle;
    uncutlen = TEScrLen;
    HandToHand(&unhandle);
#endif
    TECut(toptexthand);
    /* add to CUT buffer, one big accumulation of cuts */
    if (enoughmem(cutsize)) {
        HLock(TEScrHandle);
        PtrAndHand(*TEScrHandle, cuthandle, (Size) cutsize);
        HUnlock(TEScrHandle);
    }
}
else {
#endif ANCHORHIGH
    anchor = -1;
#endif
    tagadjust(thewind, TCUT, selstart, cutsize);
    TEDelete(toptexthand);
}
SetCtlMax(thewind->vscroll, newvctlmax(thewind));
seekselect(thewind);
break;
}

case COPY: {
    if (thewind->type == VIWIND) {
        cutsize = vitotescrap(vr.vitextbase);
    }
    else if (thewind->type == TEXTWIND || thewind->type == BUGWIND) {
        if (cutsize && enoughmem(cutsize + TEScrLen - uncutlen)) {
#endif USEUNDO
        /* take care of undo */
        lastedop = COPY;
        if (unhandle != (Handle) NULL)
            DisposHandle(unhandle);
        unhandle = TEScrHandle;
        uncutlen = TEScrLen;
        HandToHand(&unhandle);
#endif
    }
}
anchor = -1;
#endif
TECopy(toptexthand);
}
break;
}

case PASTE: {
    if (enoughmem(TEScrLen + cutsize - uncutlen)) {
        Handle htext;
        if ( ( (*thewind->hTE)->teLength - cutsize + TEScrLen) > 0) {
#endif ANCHORHIGH
        anchor = -1;
#endif
        tagadjust(thewind, TCUT, selstart, cutsize);
        tagadjust(thewind, TADD, selstart, TEScrLen);
#endif USEUNDO
        /* take care of undo */
        lastedop = PASTE;
        texttag[UNDOTAG].tagwind = unwind = thewind;
        texttag[UNDOTAG].tagstart = selstart;
        texttag[UNDOTAG].tagend = selstart + TEScrLen;
        /* last pos in new text */

        if (unhandle != (Handle) NULL)
            DisposHandle(unhandle);
        PtrToHand((Ptr) (*toptexthand)->hText + selstart), &unhandle, (long) cutsize);
    }
}

```

```

        uncutlen = cutsize;
#endif
        TEPaste(toptexthand);
        if (useselbottom)
            tesetsel((long) selstart, (long) (selstart + TEScrLen), toptexthand);
            /* select the pasted text */
        SetCtlMax(thewind->vscroll, newvctlmax(thewind));
        seekselect(thewind);
    }
    else {
        errprompt(textfull);
    }
}
break;
}
case SELECTALL: {
#endif ANCHORHIGH
    anchor = -1;
#endif
    tesetsel((long) 0, (long) (*toptexthand)->teLength, toptexthand);
    break;
}
#endif OLDSCRAP
case SCRAPTOTE: {
    extern long scaptote();

    scaptote();
    break;
}
case TETOSCRAP: {
    HLock(TEScrHandle);
    tetoscrap(*TEScrHandle, TEScrLen);
    HUnlock(TEScrHandle);
    break;
}
#endif
case SHIFTLEFT: {
    if (!useselbottom)
        tabdelete = TRUE;
        /* if this is a keyed menu, useselbottom means shift right */
case SHIFTRIGHT: {
    int beginning;

#ifdef ANCHORHIGH
    anchor = -1;
#endif
    start = beginning = selstart;

    TEbase = *(*toptexthand)->hText;
    TEpso = TEbase + start;
    selsize = cutsize;

    while (selsize > 0) {
        if (tabdelete) {
            /* kill a TAB to shift left */
            if (*TEpos == TAB) {
                tesetsel( (long) start + 1, (long) start + 1, toptexthand);
                TEKey(BS, toptexthand);
                --selsize;
                tagadjust(thewind, TCUT, start, 1);
            }
        }
        else {
            /* insert TAB to shift right, unless it's a #def line or blank */
            if (*TEpos != '#' && *TEpos != CR) {
                tesetsel( (long) start, (long) start, toptexthand);
                TEKey(TAB, toptexthand);
                selsize++;
                tagadjust(thewind, TADD, start, 1);
            }
        }
        TEbase = *(*toptexthand)->hText;
        TEpso = TEbase + start;

        /* go right, stop for end of selection or CR */
        while (--selsize && *TEpos++ != CR)
            ;
        start = TEpso - TEbase;
    }
    tesetsel( (long) beginning, (long) start + 1, toptexthand);
    break;
}
case SELKILL: {
    extern char useTEkey;

    if (useTEkey)

```

```

useTEKey = FALSE;
else
    useTEKey = TRUE;
SetItem(menu[EDIT], SELKILL, useTEKey ? "\PKey deletes selection" : "\PKey precedes selection");
infobar();
break;
}
case DUMPCUT: {
/* dump the cuthandle text and reset it */
dumpcut();
break;
}
case DELRIGHT: {
int selend;

selstart = (*inputwind->hTE)->selStart;
/* make sure it's from the start even if useselbottom */
selend = (*inputwind->hTE)->selEnd;
if (useselbottom)
    --selstart;
else
    selend++;
cutsize = selend - selstart;
if (enoughmem(cutsize + TEScrLen - uncutlen)) {
#endif ANCHORHIGH
    anchor = -1;
#endif
    tesetsel((long) selstart, (long) selend, inputwind->hTE);
#endif USEUNDO
/* take care of undo */
lastedop = CUT;
texttag[UNDOTAG].tagwind = unwind = inputwind;
texttag[UNDOTAG].tagstart = texttag[UNDOTAG].tagend = selstart;

if (unhandle != (Handle) NULL)
    DisposHandle(unhandle);
unhandle = TEScrHandle;
uncutlen = TEScrLen;
HandToHand(&unhandle);
#endif
TECut(inputwind->hTE);
tagadjust(inputwind, TCUT, selstart, cutsize);
}
break;
}
}
}

cursormenu(theItem)
int theItem;
{
register char * TPos;
register char * TEmax;
register char * TEbase;
register int count;
register char thechar;
TERec * intexptr;
TEHandle intexthand; /* input window text record */
int selstart;

if (SystemEdit(theItem - 1) )
    return;
if ( ((WindowPeek) FrontWindow())->windowKind >= userKind ) {
    HLock((Handle) inputwind->hTE);
    intexthand = inputwind->hTE;
    intexptr = *inputwind->hTE;
    if (useselbottom)
        selstart = intexptr->selEnd;
    else
        selstart = intexptr->selStart;
}
switch (theItem) {
/* add windowtype switches ? */

/* the active editing commands apply to the top window;
 * the motion commands to the input window
 * perhaps these should be put under separate menu headings
 */

case MATCHBRACK: {
    if ((selstart = intexptr->selStart) != intexptr->selEnd) {
        /* move one over to match the next pair of brackets, since we
         * may be enclosing a pair already */
        if (!useselbottom) {
            selstart++;
        }
    else {
}

```

```

        selstart = intextptr->selEnd;
    }
}
matchbrack(intexthand, selstart);
seekselect(inputwind);
break;
}
case FINDCURS: {
    if (seekselect(inputwind))
        /* if selection is already on current page, reset the range */
        tesetsel((long) selstart, (long) selstart, intexthand);
    break;
}
case SETCURS: {
    long line, midpage;

    line = (inputwind->ptr->portRect.top - intextptr->destRect.top
            + (inputwind->ptr->portRect.bottom
               - inputwind->ptr->portRect.top
               ) / 2
            ) / intextptr->lineHeight;
    if (line > intextptr->nLines)
        line = intextptr->nLines;
    midpage = intextptr->lineStarts[line];
    tesetsel(midpage, midpage, intexthand);
    break;
}
case LEFT: {
    if (selstart) {
        --selstart;
        tesetsel((long) selstart, (long) selstart, intexthand);
        seekselect(inputwind);
    }
    break;
}
case DOWN: {
    int selbottom;
    int diff;

    TEbase = *intextptr->hText;
    TEmax = TEbase + intextptr->teLength;
#endif ANCHORHIGH
    if (anchor >= selstart)
#endif
        TEpso = TEbase + selstart;
#endif ANCHORHIGH
    else
        /* if anchor is set, go from end */
        TEpso = TEbase + intextptr->selEnd;
#endif

    if (TEpos == TEmax)
        break;
    /* go back to this line's CR */
    count = 1; /* setup for predecrement */
    if (TEpos == TEbase) {
        --TEpos;
    }
    else {
        while (TEpos > TEbase) {
            if (*--TEpos == CR)
                break;
            count++;
        }
    }
    /* go forward to this line's CR */
    while (++TEpos < TEmax) {
        if (*TEpos == CR) {
            break;
        }
    }
    while (++TEpos < TEmax && *TEpos != CR && --count > 0)
        ;
    tesetsel((long) (TEpos - TEbase), (long) (TEpos - TEbase), intexthand);
    if (useselbottom)
        seekselect(inputwind);
    else {
        selbottom = seltop(intexthand) + intextptr->lineHeight;
        if (selbottom > intextptr->viewRect.bottom) {
            (*inputwind->vscroll)->ctrlValue +=
                (diff = selbottom - intextptr->viewRect.bottom);
            TEScroll(0, -diff, intexthand);
            if (myEvent.what != autoKey)
                DrawControls(inputwind->ptr);
        }
    }
    break;
}

```

```

}

case UP: {
    int top;
    int diff;

    count = 0;
    TEbase = *intextptr->hText;
    TEpso = TEbase + selstart;
    /* go back to this line's CR */
    while (TEpos > TEbase) {
        count++;
        if (*--TEpos == CR)
            break;
    }
    while (TEpos > TEbase) { /* previous line's CR */
        if (*--TEpos == CR) {
            TEpso++;
            break;
        }
    }
    while (--count > 0 && *TEpos != CR)
        TEpso++;
    tesetsel((long) (TEpos - TEbase), (long) (TEpos - TEbase), intexthand);
/* TODO should all this ungodly curmotion scroll stuff be functionalized? */

    if (useselbottom)
        seekselect(inputwind);
    else {
        top = seltop(intexthand);
        if ((diff = intextptr->viewRect.top - top)
            > 0) {
            /* TODO should not allow < zero */
            (*inputwind->vscroll)->controlValue -= diff;
            TEScroll(0, diff, intexthand);
            if (myEvent.what != autoKey)
                DrawControls(inputwind->ptr);
        }
    }
    break;
}
case RIGHT: {
#ifdef ANCHORHIGH
    if (anchor >= selstart)
#endif
selstart++;
#ifdef ANCHORHIGH
    else
        /* if anchor is set, go from end */
        selstart = intextptr->selEnd + 1;
#endif
}
tesetsel( (long) selstart, (long) selstart, intexthand);
seekselect(inputwind);
break;
}
case PAGEUP: {
    scrollTE(inputwind, 0, intextptr->viewRect.bottom
             - intextptr->viewRect.top - intextptr->lineHeight);
    DrawControls(inputwind->ptr);
    break;
}
case PAGEDOWN: {
    scrollTE(inputwind, 0, - (intextptr->viewRect.bottom
                           - intextptr->viewRect.top
                           - intextptr->lineHeight));
    DrawControls(inputwind->ptr);
    break;
}
case WORDLEFT: {
    int diff;

    TEbase = *intextptr->hText;
    TEpso = TEbase + selstart;

    /* eat blanks immediately to left */
    --TEpos;
    while (TEpos > TEbase && (*TEpos == ' ' || *TEpos == TAB)) {
        --TEpos;
        /* use (*TEpos < 0) to skip control chars along with
           if (*TEpos == CR) {
               break;
           }
        */
    }
    if (TEpos >= TEbase)
        --TEpos;
    /* go back until break character is found */
    while (TEpos >= TEbase && *TEpos >= '0')

```

```

--TEpos;
TEpos++;
tesetsel((long) (TEpos - TEbase), (long) (TEpos - TEbase), intexthand);
seekselect(inputwind);
break;
}
case WORDRIGHT: {
int diff;

TEbase = *intextptr->hText;
TEmax = TEbase + intextptr->teLength;
#ifndef ANCHORHIGH
if (anchor >= selstart)
#endif
TEpos = TEbase + selstart;
#ifndef ANCHORHIGH
else
/* if anchor is set, go from end */
TEpos = TEbase + intextptr->selEnd;
#endif
/* eat characters immediately to right if on plain char */
if (*TEpos++ >= '0')
while (TEpos <= TEmax && *TEpos >= '0')
TEpos++;

/* eat whitespace if we're on white and not just going to word end */
if (!useselbottom)
while ((*TEpos == SP || *TEpos == TAB) && TEpos <= TEmax )
TEpos++;
tesetsel((long) (TEpos - TEbase), (long) (TEpos - TEbase), intexthand);
seekselect(inputwind);
break;
}
case SENTLEFT: {
char firstfound;
int diff;
int spaces;

TEbase = *intextptr->hText;
TEpos = TEbase + selstart;
firstfound = FALSE;
spaces = 0;

if (TEpos == TEbase)
break;

/* go left, stop for CR or ".  " */
while (TEpos > TEbase) {
if ((thechar = *--TEpos) == CR) {
if (firstfound)
break;
else {
firstfound = TRUE;
spaces = 0;
}
}
if ( (thechar == '.') ||
(thechar == ')') ||
(thechar == '?') ||
(thechar == '!') ||
(thechar == '"') ||
(thechar == ']') )
)
{
if (firstfound) {
/* has the end of the previous sentence been seen? */
if (spaces == 2) {
/* looks like a good place to stop */
break;
}
}
else if (spaces) {
firstfound = TRUE;
spaces = 0;
}
}
else if (thechar == ' ')
spaces++;
else {
/* ordinary character */
firstfound = TRUE;
spaces = 0;
}
}
if (!useselbottom && TEpos != TEbase)
/* reposition at beginning of next sentence */
TEpos += spaces + 1;
}

```

```

    tesetsel((long) (TEpos - TEbase), (long) (TEpos - TEbase), intexthand);
    seekselect(inputwind);
    break;
}
case SENTRIGHT: {
    char dotfound;
    int diff;
    int spaces;

    dotfound = FALSE;
    TEbase = *intextptr->hText;
    TEmax = TEbase + intextptr->teLength;
#endif ANCHORHIGH
    if (anchor >= selstart)
#endif
        TEpos = TEbase + selstart;
#ifndef ANCHORHIGH
    else
        /* if anchor is set, go from end */
        TEpos = TEbase + intextptr->selEnd;
#endif
    spaces = 0;

    /* go right, stop for CR or ".  " */
    /* or '?' or ')' etc. */
    while (TEpos < TEmax) {
        if (*TEpos == CR) {
            /* go forward one and skip blanks */
            for (TEpos++; TEpos < TEmax; TEpos++) {
                /* skip white space */
                if (*TEpos == SP || *TEpos == TAB)
                    continue;
                break;
            }
            break;
        }
        /* now increment and look for periods etc. */
        thechar = *++TEpos;
        if (      (thechar == '.')
            || (thechar == '}')
            || (thechar == '?')
            || (thechar == '!')
            || (thechar == '"')
            || (thechar == '['))
        ) {
            dotfound = TRUE;
            spaces = 0;
            continue;
        }
        if (thechar == ' ' || thechar == TAB) {
            if (dotfound)
                spaces++;
            continue;
        }
        /* if we get to here we've got an ordinary character */
        if (dotfound && spaces >= 2) {
            /* two would be a "real" sentence */
            break;
        }
        dotfound = FALSE;
    }
    if (useselbottom) {
        /* just stay at the end of the previous sentence */
        TEpos -= spaces + 1;
    }
    tesetsel((long) (TEpos - TEbase), (long) (TEpos - TEbase), intexthand);
    seekselect(inputwind);
    break;
}
case ANCHOR: {
    if (anchor == -1) {
        anchor = selstart;
    }
    else {
        if (anchor < selstart) {
            tesetsel(anchor, (long) selstart, intexthand);
        }
        else if (anchor > selstart) {
            tesetsel((long) selstart, anchor, intexthand);
        }
        anchor = -1;
    }
    break;
}
case TAG: {
    /* set a tag corresponding to the next char a-z entered */
    /* unlike others in cursor menu, acts on topmost window */
}

```

```

int tagnum;
struct window * thewind; /* front window record */

prompt("Tag? (a-z): ");
if ( (tagnum = gettagnum()) != -1) {
    thewind = swindptr(FrontWindow());
    texttag[tagnum].tagwind = thewind;
    texttag[tagnum].tagstart = (*thewind->hTE)->selStart;
    texttag[tagnum].tagend = (*thewind->hTE)->selEnd;
}
menurestore();
break;
}
case YANK: {
/* go to a tag corresponding to the next char entered */
int tagnum;

prompt("Yank? (a-z): ");
if ( (tagnum = gettagnum()) != -1) {
    if (texttag[tagnum].tagwind != NULL) {
#endif ANCHORHIGH
        anchor = -1;
#endif
        selwind(texttag[tagnum].tagwind->ptr);
        tesetsel((long) texttag[tagnum].tagstart,
                  (long) texttag[tagnum].tagend, texttag[tagnum].tagwind->hTE);
        seekselect(texttag[tagnum].tagwind);
    }
}
menurestore();
break;
}
}
HUnlock((Handle)intexthand);
}

tetoscrap(fromp, size)
char * fromp;
int size;
{
if (enoughmem( (int) size)) {
    if ( ! ZeroScrap())
        if ( ! PutScrap((long) size, 'TEXT', fromp) )
            if ( ! UnloadScrap())
                return(size);
}
}

/* changed VICOLS to vicols */
vitotescrap(fromp)
char * fromp;
{
Ptr newtext;
char * chptr; /* for blank counting */
char * linep; /* current position in new tescrap */
int virowcount;
long copysize; /* long ex. SUMACC */
int blanks;

copysize = vr.virows * vr.vicols;
if (enoughmem((int) copysize)) {
    HLock(TEScrHandle);
    DisposPtr(*TEScrHandle);
    ResrvMem(copysize);
    linep = newtext = NewPtr(copysize);
    for (virowcount = vr.virows + 1; --virowcount; fromp += vr.vicols) {
        /* clean off the blanks and add a CR */

        for (blanks = 0, chptr = fromp + vr.vicols - 1;
             chptr >= fromp && *chptr == SP; --chptr)
            blanks++;
        BlockMove(fromp, linep, (long) (vr.vicols - blanks) );
        linep += vr.vicols - blanks;
        *linep++ = CR;
    }
    SetPtrSize(newtext, (long) (linep - newtext) );
    *TEScrHandle = newtext;
    TEScrLen = (linep - newtext);
    HUnlock(TEScrHandle);
    return(linep - newtext);
}
return(0);
}

```

```

#define TOTUSEFUL
totescrap(fromp, size)
char * fromp;
int size;
{
    Ptr newtext;

    if (enoughmem(size * 2)) {
        newtext = NewPtr( (long) size);
        BlockMove(fromp, newtext, (long) size);
        HLock(TEScrHandle);
        DisposPtr(*TEScrHandle);
        *TEScrHandle = newtext;
        TEScrLen = size;
        HUnlock(TEScrHandle);
    }
}
#endif

gettagnum()
{
    EventRecord evt;
    int tagnum;

    while (TRUE) {
        GetNextEvent(keyDownMask, &evt);
        if (evt.what == keyDown)
            break;
    }
    tagnum = evt.message & 0xff;
    if (tagnum < 'a' || tagnum > 'z') {
        beeper();
        return(-1);
    }
    return(tagnum - 'a');
}

/* adjust the text selection tag array values for adds/cuts in text
   also marks text dirty, since this routine must be called on all text mods */

tagadjust(wind, op, start, length)
struct window * wind;
int op;
int start;
int length;
{
    int count;
    int selsize;

    if (wind->dirty == FALSE) {
        wind->dirty = TRUE;
        infobar();
    }
    for (count = 0; count < TAGNUM; count++) {
        if (texttag[count].tagwind == wind) {
            /* we found a tag in the window being changed */
            if (op == TADD) {
                /* we're to adjust for a text addition */
                if (start < texttag[count].tagstart) {
                    /* before tag, add to both tag locs */
                    texttag[count].tagstart += length;
                    texttag[count].tagend += length;
                }
                else if (start > texttag[count].tagend) {
                    /* do nothing */
                }
                else {
                    /* in the middle, add to tagend */
                    texttag[count].tagend += length;
                }
            }
            else {
                /* we're to adjust for a text cut */
                if (start < texttag[count].tagstart) {
                    /* before tag, subtract from both tag locs */
                    selsize = texttag[count].tagstart - start;
                    if (length > selsize) {
                        texttag[count].tagstart = start;
                        selsize = texttag[count].tagend - (start + length);
                        if (selsize > 0)
                            /* still some selection left in tag space */
                            texttag[count].tagend = start + selsize;
                    }
                    else
                        texttag[count].tagend = start;
                }
            }
        }
    }
}

```

```

        }
        else {
            texttag[count].tagstart -= length;
            texttag[count].tagend -= length;
        }
    }
    else if (start < texttag[count].tagend) {
        /* in the middle, subtract to tagend */
        selsize = texttag[count].tagend - start;
        if (length > selsize)
            texttag[count].tagend = start;
        else
            texttag[count].tagend -= length;
    }
    /* else no effect */
}
}

/* a trap so that we can set the *&%(*&#)&^435!!!! stupid cursor to
   appear at the beginning of the line rather than letting TE make
   up its own "mind"
*/
tesetsel(start, end, tehand)
long start;
long end;
TEHandle tehand;
{
    /* TODO set whatsis to whatever */

#ifdef ANCHORHIGH
    if (anchor != -1) {
        /* automatically extend the selection range */
        if (anchor < start) {
            start = anchor;
        }
        else if (anchor > start)
            end = anchor;
    }
#endif
    (*tehand)->clikStuff = 255;
    /* gruesome TE botch requires one to do this to get cursor at beginning
       of line
    */
    TESetSelect(start, end, tehand);
}

dumpput()
{
    long cutsize;
    char * cutend;

    if (cutsize = GetHandleSize(cuthandle)) {
        SetHandleSize(cuthandle, (Size) (cutsize + 1));
        cutend = ((char *) (*cuthandle)) + cutsize;
        *cutend = '\0';
        MoveHHi(cuthandle);
        HLock(cuthandle);
        errstr(*cuthandle);
        HUnlock(cuthandle);
        DisposHandle(cuthandle);
        cuthandle = NewHandle((Size) 0);
    }
}

matchbrack(texthand, selstart)
TEHandle texthand; /* must be locked */
int selstart;
{
    register char * TEpov;
    register char * TEmax;
    register char * TEbase;
    register char * bracktab;
    register char brack1;
    register char brack2;
    register int count;
    TERec * intextptr;
    int tcount;
    int level;
    int backwards;
    extern Handle brackhand;
}
```

```

if (brackhand == NULL)
    return(-1);

#ifndef ANCHORHIGH
    anchor = -1;
#endif
intextptr = *texthand;
TEbase = *intextptr->hText;
TEmax = TEbase + intextptr->teLength;
TEpos = TEbase + selstart;

level = 1;
brack1 = *TEpos;
tcount = count = (GetHandleSize(brackhand) / 2) + 1;

for ( ; TEpos <= TEmax; TEpos++) {
    brack1 = *TEpos;
    count = tcount;
    bracktab = *brackhand;
    if (useselbottom)
        /* match bracket end */
        bracktab++;
    for ( ; --count; bracktab += 2) {
        if (*bracktab == brack1)
            /* a match exists in the table, we found a bracket */
            goto BRFONUD;
    }
}
return(0);
BRFOUND:
selstart = TEpos - TEbase;
backwards = (bracktab - *brackhand) % 2;
if (backwards)
    brack2 = *(bracktab - 1);
else {
    brack2 = *(bracktab + 1);
    if (brack1 == brack2) {
        /* just find one match for "", ',', etc. */
        brack1 = 0;
        if (useselbottom) {
            backwards = TRUE;
        }
    }
}
if (backwards) {
    /* a right end match, go backwards */
    while (--TEpos >= TEbase) {
        if (*TEpos == brack1)
            ++level;
        else if (*TEpos == brack2) {
            if (--level == 0) {
                /* found it */
                tesetsel((long) (TEpos - TEbase), (long) selstart + 1, texthand);
                break;
            }
        }
    }
}
else {
    /* look left */
    while (++TEpos < TEmax) {
        if (*TEpos == brack1)
            ++level;
        else if (*TEpos == brack2) {
            if (--level == 0) {
                /* found it */
                tesetsel((long) selstart, (long) (TEpos - TEbase + 1), texthand);
                break;
            }
        }
    }
}
}
}

```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
/* 10/21/88 changed resettext() for MF compatibility */

#include "main.h"
#include "comm.h"
#include "commdefs.h"
#include "commmenudefs.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "filenodefs.h"
#include "uw.h"
#include "mac.h"

extern longtoa();
extern portclose();

extern char * ramout;
extern char useselbottom;

extern int linecount;
extern int pagecount;

filemenu(theItem)
int theItem;
{
    struct window * frontwind;
    TEHandle toptehand;
    TERec * toptepr;

    if ( ((WindowPeek) FrontWindow())->windowKind < userKind )
        return(-1);

    if (theItem <= PRINTSEL) {
        /* this is a TE item */
        frontwind = swindptr(FrontWindow());
        toptehand = frontwind->hTE;
        HLock((Handle) toptehand);
        toptepr = *toptehand;
    }

    switch (theItem) {
        case GET:
        case FILEDELETE: {
            Point here;
            SFReply reply;
            OSType types[4];

            if (!enoughmem(4000))
                /* the dialog crashes on out of memory condition */
                break;
            here.v = 28;
            here.h = 8;

            /* if this is a file delete, do it for all types and break */
            if (theItem == FILEDELETE) {
                SFGetFile(here, "\PDelete file:", (ProcPtr) NULL,
                          -1, types, (ProcPtr) NULL, &reply); /* -1 == all types */
                if (reply.good) {
                    int errno;

                    if (FSDelete(reply fName, reply.vRefNum) )
                        errstr("no delete\r");
                }
                break;
            }

            types[0] = 'TEXT';
            types[1] = 'A/UX';

            SFGetFile(here, "\PLoad into top window:", (ProcPtr) NULL,
                      2, types, (ProcPtr) NULL, &reply);
            if (reply.good) {
                gettext(frontwind, &reply fName, (short) reply.vRefNum);
                frontwind->fvol = SFSaveDisk;
                frontwind->fdir = CurDirStore;
                frontwind->dirty = FALSE;
            }
            break;
        }
        case SAVE: {
            Point here;
            SFReply reply;
            char title[256];

            if (!enoughmem(2048))
```

```

/* the dialog crashes on out of memory condition */
break;

here.v = 30;
here.h = 40;

if (useselbottom)
    errprompt("Appending...");

GetWTitle(frontwind->ptr, (Str255 *) title);
#ifndef PAIDFOR
    if (frontwind->fvol)
        SFSaveDisk = frontwind->fvol;
    if (frontwind->fdir)
        CurDirStore = frontwind->fdir;
#endif
SPPutFile(here, "\PSave top window as:", (Str255 *) title,
          (ProcPtr) NULL, &reply);
if (reply.good) {
    if (!savetext(reply.vRefNum, &reply.fName,
                  topteptr->hText, (long) 0, (long) topteptr->teLength)) {
        frontwind->fvol = SFSaveDisk;
        frontwind->fdir = CurDirStore;
        frontwind->dirty = FALSE;
        if (frontwind != bugwind)
            SetWTitle(frontwind->ptr, reply.fName);
        infobar();
    }
}
break;
}

case SAVESELECT: {
    Point here;
    SFReply reply;

    if (!enoughmem(2048))
        /* the dialog crashes on out of memory condition */
        break;

    here.v = 30;
    here.h = 40;

    if (useselbottom)
        errprompt("Appending...");

    SPPutFile(here, "\PSave selection as:", "\P", (ProcPtr) NULL, &reply);
    if (reply.good) {
        savetext(reply.vRefNum, &reply.fName,
                 topteptr->hText, (long) topteptr->selStart, (long) topteptr->selEnd);
        frontwind->fvol = SFSaveDisk;
        frontwind->fdir = CurDirStore;
    }
}
break;
}

case PRINTSEL: {
    /* put on 8 port with printing TRUE for LF addition & page ejects */
    /* TODO dis-extend porttext() for vi */
    short start, end;

    if (laserprint) {
        if (!printer.out) {
            PrDrvOpen();
            if (PrError())
                errprompt("Can't open printer driver");
            break;
        }
        printer.out = TRUE;
        /* so routines know a printer exists */
    }

    /* use Apple text-streaming printer commands */
    PrCtlCall(iPrDevCtl, (long) lPrDocOpen, (long) 0, (long) 0);
    if (PrError())
        errprompt("Can't open printer document");
    break;
}
PrCtlCall(iPrDevCtl, (long) lPrPageOpen, (long) 0, (long) 0);
if (PrError())
    errprompt("Can't open printer page");
break;
}
pagecount = 0;
linecount = 0;
}
if (printer.out) {
    start = topteptr->selStart;
    end = topteptr->selEnd;
}

```

```

        if (start == end) {
            start = 0;
            end = topteptr->teLength;
        }
        porttext(&printer, toptehand, start, end, TRUE);
        if (laserprint) {
            while (cendline(&printer))
                sendline(&printer);
        }
    }
break;
}
case PRINTRESET: {
/* initialize printer */
/* set left margin, TOF, & tab settings */
static char printinit[] = "\033L012\033v\033(004,008,012,020,028,036,044,052,060,068,076. ";

linecount = 0;
pagecount = 0;
/* TODO best font/size combo, ? */

if (!printer.out) {
    /* initialize the printer to default on printer port */
    portinit(PRINTERPORT, &printer);
}
portline(&printer, sizeof(printinit), printinit, FALSE, FALSE);
break;
}
case DBLSPACE: {
extern int linespacing;

ParamText("\PSet printer line spacing to:", "\Plines", "\P", "\P");
linespacing = dosetnumdial(linespacing);
break;
}
case PAGEWAIT: {
extern char pagewait;

if (pagewait)
    pagewait = FALSE;
else
    pagewait = TRUE;
CheckItem(menu[FILE], theItem, (Boolean) pagewait);
break;
}
case LASERPRINT: {
if (laserprint)
    laserprint = FALSE;
else
    laserprint = TRUE;
CheckItem(menu[FILE], theItem, (Boolean) laserprint);
break;
}
case NEWPAGE: {
/* set printer TOF */
if (printer.out)
    portWrite(&printer, FF);
linecount = 0;
break;
}
case PRINTHEADER: {
prompt("Type page header: ");
getline(headerstr, PRHEADMAX);
break;
}
case PRINTCANCEL: {
transcancel(&printer);
break;
}
case SENDCANCEL: {
transcancel(&modem);
break;
}
case SAVECONF: {
#endif DORES
#ifndef SEGMENT
    /* check for space */
    if (lenoughmem(6000))
        break;
#endif
    if (saveconf())
        errstr("save failed\r");
#endif SEGMENT
UnloadSeg(saveconf);
#endif
#else
extern char * unimplemented;

```

```

#endif
        errstr(unimplemented);
    }
}
case QUIT: {
    extern char keyedmenu;
    int windcount;
    char dirty;

    for (windcount = 0, dirty = FALSE; windcount < VIWINDNUM; windcount++) {
        if (window[windcount].dirty) {
            dirty = TRUE;
            break;
        }
    }
    if (keyedmenu || dirty) {
        if (dirty) {
            prompt("Text window(s) modified. Type Return to quit anyway...");
            SysBeep(2);
        }
        else {
            prompt("Type Return to quit...");
        }
        if (mygetchar() != CR) {
            menurestore();
            break;
        }
    }
    quit();
}
case CAPTURE: {
    Point here;
    SFReply reply;
    char title[256];
    extern int fcapture;
    extern short fdcapture;
    extern int capvol;
    extern char saveonlf;

    if (fcapture || saveonlf || !enoughmem(2048))
        /* the dialog crashes on out of memory condition */
        break;

    here.v = 30;
    here.h = 40;

    SFPutFile(here, "\PSave host output as:", "\P", (ProcPtr) NULL, &reply);
    if (reply.good) {
        long newlag;
        extern long captime;
        extern int captureout;
        extern int readlag;
        extern int oreadlag;
        extern long blockNBytes;

        if (! newfile(reply.vRefNum, &reply.fName, &fdcapture)) {
            saveonlf = TRUE;
            /* fcapture will be set on after next LF received */
            capvol = reply.vRefNum;
            captime = 0x00FFFFFFL; /* wait a long time at first */
        }
    }
    break;
}
case CAPKILL: {
    extern struct window * capturewind;

    if (fcapture)
        endcapture(capturewind);
    break;
}
case SENDLINE:
case SENDALL: {
    if (local)
        break;
    if (theItem == SENDALL || (keyedmenu && useselbottom))
        /* Send the whole window */
        TESetSelect((long) 0, (long) (*inputwind->hTE)->teLength, inputwind->hTE);
    sendselect();
    break;
}
if (theItem <= PRINTSEL)
    HUnlock((Handle) topehand);
}

```

```

/* fixes a TE record to display a new text */

resettext(wind)
struct window * wind;
{
    register TEHandle texthand;
    texthand = wind->hTE;

    /* clear text and fix TERecord for new text */
    TECalText(texthand);

    /* reset text to first page */
    (*texthand)->destRect.top = TEOFFSET;
    (*texthand)->destRect.left = TEOFFSET;
    /* TEUpdate(nullrect, texthand); */
    TEUpdate(&(*wind->ptr->visRgn)->rgnBBox, wind->hTE);
    /* This seems to induce crash when doc launching under MultiFinder!
    InvalRect(&wind->ptr->portRect);
    */
    TESetSelect((long) 0, (long) 0, texthand);
    TEKey(B5, texthand);
    /* fix the misplaced cursor*/

    /* reset controls */
    (*wind->vscroll)->contrlValue = 0;
    (*wind->hscroll)->contrlValue = 0;
    SetCtlMax(wind->vscroll, newvctlmax(wind));
    /* use setmax to draw it too */
}

savetext(volume, filename, savehand, startpos, endpos)
short volume;
Str255 * filename;
TEHandle savehand;
long startpos;
long endpos;
{
    long count;
    long fpos;
    int errno;
    int fd;
    char * fromp;

    errno = FALSE;

    if (newfile(volume, filename, &fd))
        return(-1);
    HLock((Handle) savehand);
    fromp = ((char *) *savehand) + startpos;
    SetCursor(*watch);

    count = endpos - startpos;
    if (filewrite(fd, &count, fromp))
        return(-1);
    HUnlock((Handle) savehand);
    GetFPos(fd, &fpos);
    SetEOF(fd, fpos);
    if ( FSClose(fd) ) {
        errstr("no close\r");
        errno = TRUE;
    }
    if (FlushVol( (Ptr) NULL, volume)){ /* CONV StringPtr */
        errstr("no update\r");
        errno = TRUE;
    }
    if (errno)
        return(-1);
}

int filesegment; /* load the nth 30000 byte portion of a file if >32K */
#define LOADSIZE 30000

gettext(wind, fname, volnum)
struct window * wind;
Str255 fname;
short volnum;
{
    Handle readbuf;
    long length; /* long ex. SUMACC */
    int errno;
    short fd;

    errno = FSOpen(fname, volnum, &fd);

```

```

if (errno) {
    errstr("no open\r");
    return(-1);
}
GetEOF(fd, &length);
if (length > TMAXLENGTH) {
    char temp[100];
    long tlen;
    long getlen;

    if (filesegment) {
        tlen = (unsigned long) filesegment * (unsigned long) LOADSIZE;
        SetFPos(fd, fsFromStart, tlen);
    }
    else {
        sprintf(temp, "file %ld bytes, loading 30K\r", length);
        errstr(temp);
        tlen = 0;
    }
    filesegment++;
    if ((length - tlen) > LOADSIZE)
        length = LOADSIZE;
    else {
        /* get just what's left */
        if ( (length -= tlen) < 0)
            /* the fileseg doesn't exist */
        length = 0;
    }
}

/* disposes of all text in TERecord */
if ( !(readbuf = NewHandle(length)) )
    errstr(ramout);
else {
    DisposeHandle((*wind->hTE)->hText);
    HLock(readbuf);
    SetCursor(*watch);
    if (FSRead(fd, &length, *readbuf))
        errstr("no read\r");
    if (wind != bugwind)
        SetWTitle(wind->ptr, fname);
    (*wind->hTE)->hText = readbuf;
    (*wind->hTE)->teLength = length;
    HUnlock(readbuf);

    resettext(wind);
}
if (FSClose(fd))
    errstr("no close\r");
}

newfile(volume, filename, fdp)
short volume;
Str255 * filename;
short * fdp;
{
    int errno;
    extern char useselbottom;

    /* TODO problems here with Open failure */
    if ( (errno = FSOpen(filename, volume, fdp))) {
        if ( Create( filename, volume, 'dumv', 'TEXT') ) {
            errstr("no create\r");
            *fdp = 0;
            return(-1);
        }
        else {
            FSOpen(filename, volume, fdp);
        }
    }
    if (useselbottom) {
        /* cheap Append mode! */
        SetFPos(*fdp, fsFromLEOF, 0L);
    }
}

filewrite(fd, count, fromp)
int fd;
long * count;
char * fromp;
{
    int errno;

    errno = FSWrite(fd, count, fromp);
    switch (errno) {
        case noErr:

```

```
        break;
    case dskFullErr:
        errstr("disk full\r");
        break;
    case flckdErr:
        errstr("file locked\r");
        break;
    case wPrErr:
        errstr("disk locked\r");
        break;
    default:
        errstr("no write\r");
}
return(errno);
}

quit()
{
    extern char xoff;
    extern char dofflow;
#endif VRT
    extern VBLTask gateVBL;
#endif

#endif VRT
    VRemove(&gateVBL);
#endif

if (modem.in) {
    if (noserd) {
        if (dofflow && xoff)
            portWrite(&modem, XON);
    }
    portclose(&modem);
}
if (printer.in) {
    portclose(&printer);
}
if (laserprint && printer.out) {
    PrDrvClose();
}
keyclose();
exit(0);
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "defs.h"
#include "main.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "videfs.h"
#include "vi.h"

#define MINSIZE    3
#define MAXSIZE   30

#define NUMSIZES 10 + 1
#define BLANK    11
#define LEADUP   12
#define LEADDOWN 13

int oldsize = 8;           /* TODO initialize thru the menu */
int leading = 0;
/* for vi leading so user can adjust to suit */

sizemenu(theItem)
int theItem;
{
    struct window *thewind;      /* front window record */

    Str255 sizename;
    long fontsize; /* long ex. SUMACC */

    if ( ((WindowPeek) FrontWindow())->windowKind < userKind )
        return;

    thewind = swindptr(FrontWindow());
#endif VILEAD
    if (theItem == LEADUP) {
        leading++;
        return(0);
    }
    if (theItem == LEADDOWN) {
        --leading;
        return(0);
    }
#endif
    GetItem(menu[SIZE], theItem, sizename);
    StringToNum(sizename, &fontsize);
    setfontsize(thewind, (short) fontsize);
    CheckItem(menu[SIZE], oldsize, (Boolean) FALSE);
    CheckItem(menu[SIZE], theItem, (Boolean) TRUE);
    oldsize = theItem;
}

/* sets up font size menu, returning TRUE if current size exists */

setsizemenu(fontnum, cursize)
register int fontnum;
int cursize;
{
    register int fontsize;
    int itemcount;
    int realsize; /* does current size exist? */
    Str255 name;

    fontsize = MINSIZE;
    realsize = FALSE;
    DeleteMenu(sizeMenu);
    DisposeMenu(menu[SIZE]);
    menu[SIZE] = NewMenu(sizeMenu, "\PSize");
    InsertMenu(menu[SIZE], topMenu);

    for (itemcount = 1; fontsize < MAXSIZE; fontsize++) {
        if (RealFont(fontnum, fontsize) ) {
            NumToString( (long) fontsize, name);
            AppendMenu(menu[SIZE], name);
            if (cursize == fontsize) {
                CheckItem(menu[SIZE], itemcount, (Boolean) TRUE);
                oldsize = itemcount;
                realsize = TRUE;
            }
            itemcount++;
        }
    }
    return(realsize);
}

```

```

setfontsize(thewind, fontsize)
struct window * thewind;
short fontsize;
{
    TEHandle texthand;
    FontInfo finfo;
    Rect view;

    if (fontsize <= 0)
        return(0);
    switch (thewind->type) {
        case BUGWIND:
        case TEXTWIND: {
            register TERec * textptr;

            HLock((Handle) thewind->hTE);
            texthand = thewind->hTE;
            textptr = *texthand;

            textptr->txSize = (short) fontsize;

            /* update TE to get the right fontinfo */
            TEUpdate(&nullrect, texthand);
            GetFontInfo(&finfo);
            textptr->fontAscent = finfo.ascent + finfo.leading;
            textptr->lineHeight = textptr->fontAscent + finfo.descent;

            /* update TE for lnheight */
            TEUpdate(&nullrect, texthand);

            /* reset text to top without changing selection point */
            /* WARNING this & scrollTE may be screwing up controls */
            textptr->destRect.top = TEOFFSET;
            (*thewind->vscroll)->controlValue = 0;

            sizewindow(thewind->ptr, qd.thePort->portRect.bottom,
                       qd.thePort->portRect.right);
            /* this should be a call to a subset of sizew */
            TEUpdate(&nullrect, texthand);
            EraseRect(&textptr->viewRect);
            TEUpdate(&textptr->viewRect, texthand);
            seekselect(thewind);
            SetCtlMax(thewind->vscroll, newvctlmax(thewind));
            HUnlock((Handle) thewind->hTE);
            break;
        }
        case VIWIND: {
            /* reset vi values by writing out :set commands ? */
            /* reset vimaxline */
            if (fontsize)
                TextSize( (int) fontsize);
            GetFontInfo(&finfo);

            if (thewind != viwind) {
                /* copy the globals about: minimizes pointer mush in code */
                vicontext(thewind);
            }
            vr.lineheight = finfo.ascent + finfo.descent + leading;
            /* + finfo.leading; */
            vr.fontheight = finfo.ascent + leading; /* + finfo.leading; */
            vr.fontdescent = finfo.descent;

            EraseRect(&virect);
            vr.cursor = FALSE;
            virefresh();
            updvscreen(thewind);
            break;
        }
    #ifdef TEKEXIST
        case TEKWIND: {
            /* should look for commonalities w/ VIWIND routine */
            TextSize( (int) fontsize);
            break;
        }
    #endif
    }
}

```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "defs.h"
#include "main.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "topmenudefs.h"
#include "utildefs.h"

#define CONTROLSZ 16

char useselbottom;

topmenu(theItem)
int theItem;
{
    struct window * thewind;      /* front window record */
    char promptstring;

    if ( ((WindowPeek) FrontWindow())->windowKind < userKind )
        return;
    thewind = swindptr(FrontWindow());
    promptstring = FALSE;

    /* top menu is only enabled when TEXTWIND or BUGWIND are on, so
       we don't need to check the type */
    switch (theItem) {
        case SELSIZE:   {
            long selsize;

            selsize = (*thewind->hTE)->selEnd - (*thewind->hTE)->selStart;
            longtoa(error, selsize);
            errprompt(error);
            break;
        }
        case FINDPROMPT:
            promptstring = TRUE;
        case FINDSEL:   {
            TEHandle texthand;      /* top window text record */
            long loc, count;
            int selstart, selend;
            int direction;
            extern findstring();
            extern char fstring[];
        #ifdef ANCHORHIGH
            extern long anchor;
            anchor = -1;
        #endif
            if (myEvent.modifiers & optionKey)
                direction = BACKWARD;
            else
                direction = FORWARD;
            texthand = thewind->hTE;
        #ifdef LINEPRT
            if (promptstring) {
                char matchstr[FINDMAX];

                if (direction == FORWARD)
                    prompt("Match: ");
                else
                    prompt("Match Back: ");
                if (getline(matchstr, FINDMAX) > 0)
                    strcpy(fstring, matchstr);
                else
                    break;
            }
            else {
        #endif
                selstart = (*texthand)->selStart;
                selend = (*texthand)->selEnd;
                if (selstart != selend) {
                    /* if text selected, copy select range to string buffer */
                    mystrncpy(*(*texthand)->hText + selstart, fstring,
                              selend - selstart, FINDMAX);
                }
        #ifdef LINEPRT
                }
        #endif
            if ( (count = findstring(texthand, &loc, direction)) > 0) {
                /* found a match of the fstring */
                tesetsel(loc, (long) (loc + count), texthand);
            }
            else
                /* reset selection to insertion point */
                tesetsel((long) (*texthand)->selStart,
                         (long) (*texthand)->selStart, texthand);
        }
    }
}

```

```

seekselect(thewind);
break;
}
case CURSTOLINE: {
#define LINESTRLEN 80
    char linestr[LINESTRLEN];
    long linenum, tnpos;
    int getresp;

    linenum = 0;
    prompt("# ");
    getresp = getline(linestr, LINESTRLEN);
    if (getresp == -1) {
        break;
    }
    else if (getresp > 0) {
        ctop(linestr);
        StringToNum((Str255 *) linestr, &linenum);
    }

    if (linenum == 0 || linenum > (*thewind->hTE)->nLines)
        tnpos = (long) (*thewind->hTE)->tLength;
    else
        tnpos = (*thewind->hTE)->lineStarts[linenum - 1];

    TESetSelect(tnpos, tnpos, thewind->hTE);
    seekselect(thewind);
    break;
}
case SCROLL: {
/* toggle output window scroll */
    if (thewind->scroll)
        thewind->scroll = FALSE;
    else
        thewind->scroll = TRUE;
    CheckItem(menu[TOP], theItem, (Boolean) thewind->scroll);
    break;
}
case WORDWRAP: {
/* toggle output window wordwrap */
    if ((*thewind->hTE)->crOnly >= 0) {
        (*thewind->hTE)->crOnly = -1;
        (*thewind->hTE)->destRect.right = 32000;
        CheckItem(menu[TOP], theItem, FALSE);
    }
    else {
        (*thewind->hTE)->crOnly = FALSE;
        (*thewind->hTE)->destRect.right =
            thewind->ptr->portRect.right - CONTROLSZ - 4;
        CheckItem(menu[TOP], theItem, TRUE);
    }
    infobar();
    break;
}
case REFORMAT: {
    resettext(thewind);
    break;
}
case FILESEGMENT: {
    extern int filesegment;

    ParamText("\PWhen loading file, get segment #", "\P", "\P", "\P");
    filesegment = dosetnumdial(filesegment);
    break;
}
case FREE: {
    long total, max, grow;
    extern char memfull;

    memfull = FALSE;
    total = FreeMem();
/*
    errstr("Free: ");
*/
    longtoa(error, total);
    errprompt(error);

    /* if (total > 5000)
       watch out for stack */
    max = MaxMem(&grow);
#endif SHOWMAX
    errstr("Now: ");
    longtoa(error, max);
    errstr(error);
    errstr("\r");
#endif
    break;
}

```

```

        }

}

settopmenu(thewind)
struct window * thewind;
{
    /* set scroll checkmark */
    CheckItem(menu[TOP], SCROLL, (Boolean) thewind->scroll);

    if (thewind->type == TEXTWIND || thewind->type == BUGWIND) {
        if ( (*thewind->hTE)->crOnly >= 0) {
            CheckItem(menu[TOP], WORDWRAP, TRUE);
        }
        else {
            CheckItem(menu[TOP], WORDWRAP, FALSE);
        }
    }
    return;
}

/* don't do this with a non-TEXT window! */
/* scroll to selection--avoid scroll if in window
   returns flag indicating whether range in window
 */

seekselect(thewind)
struct window * thewind;
{
    int top;
    int left;
    int leftright;
    long start, end;
    TEHandle texthand;
    int vshift = FALSE;

    texthand = thewind->hTE;
    top = seltop(texthand);
    leftright = offleft(texthand);
#ifdef OFFDEBUG
longtoa(error, (long) leftright);
errstr(error);
errstr("--offset\r");
#endif
    left = leftright + (*texthand)->destRect.left;

    if ( (top < (*texthand)->viewRect.top)
        || (top + (*texthand)->lineHeight > (*texthand)->viewRect.bottom ) ){
        scrollTE(thewind,
                  0,
                  - top
                  + ( (*texthand)->viewRect.bottom
                      - (*texthand)->viewRect.top
                      ) / 2 );
        DrawControls(thewind->ptr);
        vshift = TRUE;
    }
    if ( (left < (*texthand)->viewRect.left + 2) ||
        (left > (*texthand)->viewRect.right - 2) ) {
        int width;

        width = (*texthand)->viewRect.right - (*texthand)->viewRect.left;
        if (leftright < width)
            /* we can go all the way to the left margin */
            scrollTE(thewind, - ((*texthand)->destRect.left - TEOFSET), 0);
        else
            scrollTE(thewind, - left + width / 2, 0);
        DrawControls(thewind->ptr);
    }
    return(!vshift);
}

extern unsigned curline();
int theline;

seltop(texthand)
TEHandle texthand;
{
    return( ( (theline = curline(texthand)) * (*texthand)->lineHeight) + (*texthand)->destRect.top);
    /* we cleverly set theline because offleft() needs it too */
}

/* return the offset of the selection point from the left end of the text */
/* global theline needs to be set prior to call to offleft() */

```

```

offleft(texthand)
TEHandle texthand;
{
    int offset;
    int start;
    int pos;

    start = (*texthand)->lineStarts[theline];
#ifdef FINDDEBUG
longtoa(error, (long) start);
errstr(error);
errstr("--start\r");
#endif
    pos = useselbottom ? (*texthand)->selEnd : (*texthand)->selStart;
#ifdef FINDDEBUG
longtoa(error, (long) pos);
errstr(error);
errstr("--pos\r");
#endif
#endif
    if (theline <= (*texthand)->nLines) {
        HLock((*texthand)->hText);
        offset = TextWidth((*texthand)->hText, start, pos - start);
        HUnlock((*texthand)->hText);
    }
    else
        /* we're on the last line with 1 CR */
        offset = 0;
    return(offset);
}

unsigned int
curline(texthand)
TEHandle texthand;
{
    TERec * textrec;
    register int midway;
    register int count;
    register short selstart;
    register short * linestart;
    register short * linesend;
    int line;

    textrec = *texthand;
    selstart = useselbottom ? textrec->selEnd : textrec->selStart;
    midway = textrec->nLines;
    if ((textrec->lineStarts[midway - 1]) <= selstart) {
        /* we're on the last line, a common case */
        if (selstart == textrec->teLength
            && *(textrec->hText + selstart - 1) == CR)
            /* we're at the very end after a CR, nLines is one short */
            midway++;
        --midway;
        return(midway);
    }

    linesend = textrec->lineStarts + textrec->nLines;
    linestart = textrec->lineStarts + midway / 2;

    for (; (midway /= 2) > 1; ) {
        /* binary search n times to narrow the field */
        if (*linestart == selstart)
            break;
        else if (*linestart < selstart)
            linestart += midway;
        else
            linestart -= midway;
    }

    /* now find the exact match by linear methods */
    if (*linestart < selstart) {
        while (linestart < linesend && ++linestart <= selstart)
            ;
        --linestart;
        /* correct for overshoot */
    }
    else if (*linestart > selstart) {
        while (*--linestart > selstart)
            ;
    }
    line = linestart - textrec->lineStarts;
#ifdef FINDDEBUG
longtoa(error, (long) line);
errstr(error);
errstr("\r");
#endif
#endif
    return(line);
}

```

```
mystrncpy(source, destination, len, maxlen)
char * source;
char * destination;
int len;
register int maxlen;           /* maximum length in destination */
{
    register char * srcp;
    register char * destp;
    register char * endpos;
    register int count;

    count = 0;
    srcp = source;
    destp = destination;
    for (endpos = srcp + len; srcp < endpos; destp++, srcp++) {
        *destp = *srcp;
        if (++count >= maxlen) {
            /* we can't move the whole thing */
            *destp = '\0';
            return(-1);
        }
    }
    *destp = '\0';
}

#ifndef INSANE
/* use grotty TEDoText routine to set pen at cursor so we can find it */

setpen(thewind)
struct window * thewind;
{
    TERec * terec;

    HLock(thewind->hTE);
    terec = *thewind->hTE;
    #asm
        move.l -4(a6),a3
        moveq #-2,d7
        move.l $a70,a1
        jsr    (a1)
    #endasm
    HUnlock(thewind->hTE);
    return(0);
}
#endif
```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */
```

```
#include "defs.h"
#include "main.h"
#include "mainmenu.h"
#include "comm.h"
#include "menudefs.h"
#include "editmenudefs.h"
#include "echomenudefs.h"
#include "windmenudefs.h"
#include "cursormenudefs.h"
#include "uw.h"
#include "startup.h"
```

```
int oldin = WINDMBASE + 0;
int oldout = WINDMBASE + 0;
```

```
/* actions taken on user window selection in windmenu */
int makeinput = TRUE;
```

```
extern struct window * makewindow();
extern char useselbottom;
```

```
windmenu(theItem)
```

```
int theItem;
```

```
{
    struct window * thewind;
    WindowPtr frontwind;
```

```
    frontwind = FrontWindow();
    if (((WindowPeek) frontwind)->windowKind < userKind)
        return(-1);
```

```
    thewind = swindptr(frontwind);
    switch (theItem) {
        case INPUT: {
            if (thewind->cstate & WS_INPUT) {
                thewind->cstate &= ~WS_INPUT;
            }
            else {
                blankiomarkers();
                thewind->cstate |= WS_INPUT;
                inputwind = thewind;
                makeinput = TRUE;
                setiomarkers(thewind);
            }
            infobar();
            break;
        }
        case OUTPUT: {
            int wnum;
```

```
            wnum = thewind - window;
            if (thewind->cstate & WS_OUTPUT) {
                thewind->cstate &= ~WS_OUTPUT;
```

```
/* kill?
```

```
            if (thewind->cstate & WS_UW)
                uwcmd(CB_FN_KILLW, wnum);
        }
    }

```

```
    else {
        if (!uwon) {
            blankiomarkers();
            outputwind = thewind;
            thewind->cstate |= WS_OUTPUT;
            setiomarkers(thewind);
        }
        else if (!thewind->cstate & WS_UW) {
            /* alert UW about new window ? */
            if (wnum > 1 && wnum <= NWINDOW)
                uwcmd(CB_FN_NEWW, wnum);
            thewind->cstate |= WS_UW;
        }
    }
    infobar();
    break;
}
```

```
case WINDMBASE:
```

```
case WINDMBASE + 1:
case WINDMBASE + 2:
case WINDMBASE + 3:
case WINDMBASE + 4:
case WINDMBASE + 5:
case WINDMBASE + 6:
case WINDMBASE + 7:
case WINDMBASE + 8:
case WINDMBASE + 9: {
```

```

/* this module uses the menu item marks to keep track of types */
/* all mark arguments to GetItemMark & SetItemMark are char ex. SUMACC */
/* all menu item arguments to Get & SetItemMark are short ex. SUMACC */
int windnum;

windnum = theItem - WINDMBASE;
thewind = &window[windnum];
if (thewind->ptr == NULL) {
    /* make the window */
    /* using defaults to get reasonable placement by num */
    thewind = makewindow(windnum, TEXTWIND, deffont, defsize, 0, 0, 0, 0);
    if (thewind != NULL)
        ShowWindow(thewind->ptr);
    /* TODO should be ((WindowPeek) thewind->ptr)->visible = TRUE; */
    else
        break;
}
if (thewind->ptr != FrontWindow() ) {
    selwind(thewind->ptr);
}
else
    infobar();
break;
}

case NEXTDISPLAY: {
/* TODO vi should support multiple windows--add VI/TEK make option
   for use w/keys? */
/* this should switch the window mode to another mode?
   switch with one switch, or have separate keyboard commands?
*/
windactivate(frontwind, FALSE);
if (!useselbottom) {
    /* text, vi, tek order */
    if (thewind->type == TEXTWIND) {
        thewind->type = VIWIND;
    }
    else if (thewind->type == VIWIND) {
        thewind->type = TEKWIND;
    }
    else if (thewind->type == TEKWIND) {
        thewind->type = TEXTWIND;
    }
}
else {
    /* go backwards: text, tek, vi */
    if (thewind->type == TEXTWIND) {
        thewind->type = TEKWIND;
    }
    else if (thewind->type == VIWIND) {
        thewind->type = TEXTWIND;
    }
    else if (thewind->type == TEKWIND) {
        thewind->type = VIWIND;
    }
}
windactivate(frontwind, TRUE);
InvalRect(&frontwind->portRect);
break;
}
case SYSSWITCH: {
if (bugwind->ptr == FrontWindow() ) {
    SendBehind(bugwind->ptr, (WindowPtr) NULL);
    selwind(FrontWindow());
}
else {
    selwind(bugwind->ptr);
}
break;
}
}

blankiomarkers()
{
    /* now set the window according to the window menu settings */

    /* blank out the old settings */
    SetItemMark(menu[WIND], oldin, '\000');
    SetItemMark(menu[WIND], oldout, '\000');
}

setiomarkers(thewind)
struct window * thewind;
{
    oldin = inputwind - window + WINDMBASE;

```

```
oldout = outputwind - window + WINDMBASE;

/* set the input/output markers in the window menu */
SetItemMark(menu[WIND], oldin, '<');

if (!uwon) {
    /* don't set any output markers when unix windows on */
    if (oldin == oldout)
        SetItemMark(menu[WIND], oldin, '=');
    else {
        SetItemMark(menu[WIND], oldout, '>');
    }
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "defs.h"
#include "main.h"
#include "videfs.h"
#include "vi.h"
#include "mac.h"

#define VISCRAWT 500
#define CONTROLSZ 16

struct window * paswind; /* a global for the ROM called scroll loops */
int telineheight; /* get TE value just once! */

scrbarclick(thewind)
register struct window * thewind;
{
    int CntlPart;
    pascal void upscroll(), downscroll(), leftscroll(), rightscroll();
    ControlHandle thecontrol;
    register WindowPtr windptr;

    windptr = thewind->ptr;
    if (! (CtlPart = FindControl(myEvent.where, thewind->ptr, &thecontrol)))
        return;
    if (thecontrol == NULL)
        return;

    switch ((int) (*thecontrol)->contrlRfCon) {
        case HSCROLL: {
            /* could TODO H & V be combined with ?:'s ? */
            if (thewind->type == TEXTWIND) {
                switch (CtlPart) {
                    case inUpButton: {
                        if (TrackControl(thecontrol, myEvent.where, (ProcPtr) leftscroll))
                            DrawControls(windptr);
                        break;
                    }
                    case inDownButton: {
                        if (TrackControl(thecontrol, myEvent.where, (ProcPtr) rightscroll))
                            DrawControls(windptr);
                        break;
                    }
                    case inPageUp: {
                        pagescroll(thecontrol, CntlPart, windptr->portRect.right - CONTROLSZ, 0);
                        break;
                    }
                    case inPageDown: {
                        pagescroll(thecontrol, CntlPart, - (windptr->portRect.right - CONTROLSZ), 0);
                        break;
                    }
                    case inThumb: {
                        int oldthumb, scrollamount;

                        oldthumb = (*thecontrol)->contrlValue;
                        if (TrackControl(thecontrol, myEvent.where,
                                         (ProcPtr) NULL) )
                        {
                            scrollamount = (*thecontrol)->contrlValue - oldthumb;
                            TEScroll(-scrollamount, 0, thewind->hTE);
                        }
                        break;
                    }
                }
            }
            else if (thewind->type == VIWIND) {
                if (CtlPart == inPageUp) {
                    /* set offset to the right, adding text on the left */
                    (*thecontrol)->contrlValue -= VISCRAWT;
                    vr.scroffset += VISCRAWT;
                }
                else if (CtlPart == inPageDown) {
                    /* set offset back to the left, adding text at right */
                    (*thecontrol)->contrlValue += VISCRAWT;
                    vr.scroffset -= VISCRAWT;
                }
                else
                    break;
                InvalRect(&qd.thePort->portRect);
            }
            break;
        }
        case VSCROLL: {
            telineheight = (*thewind->hTE)->lineHeight;
            switch (CtlPart) {

```

```

    case inUpButton: {
        if ( TrackControl(thecontrol, myEvent.where, (ProcPtr) upscroll) )
            DrawControls(windptr);
        break;
    }
    case inDownButton: {
        if ( TrackControl(thecontrol, myEvent.where, (ProcPtr) downscroll) )
            DrawControls(windptr);
        break;
    }
    case inPageUp: {
        pagescroll(thecontrol, CntlPart, 0, (int)
        {
            (*thewind->hTE)->viewRect.bottom
            - (*thewind->hTE)->viewRect.top
            - telineheight
        });
        break;
    }
    case inPageDown: {
        pagescroll(thecontrol, CntlPart, 0, (int)
        -
            (*thewind->hTE)->viewRect.bottom
            - (*thewind->hTE)->viewRect.top
            - telineheight
        );
        break;
    }
    case inThumb: {
        int oldthumb, scrollamount;

        oldthumb = (*thecontrol)->contrlValue;
        if (TrackControl(thecontrol, myEvent.where,
                          (ProcPtr) NULL) )
        {
            scrollamount = (*thecontrol)->contrlValue - oldthumb;
            TEScroll(0, -scrollamount, thewind->hTE);
        }
        break;
    }
    break;
}
}

pascal void
upscroll(control, theCode)
ControlHandle control;
int theCode;
{
    if (theCode != inUpButton)
        return;
    scrollTE(paswind, 0, telineheight);
}

pascal void
downscroll(control, theCode)
ControlHandle control;
int theCode;
{
    if (theCode != inDownButton)
        return;
    scrollTE(paswind, 0, -telineheight);
}

pascal void
leftscroll(control, theCode)
ControlHandle control;
int theCode;
{
    if (theCode != inUpButton)
        return;
    scrollTE(paswind, 30, 0);
}

pascal void
rightscroll(control, theCode)
ControlHandle control;
int theCode;
{
    if (theCode != inDownButton)
        return;

```

```

    scrollTE(paswind, -30, 0);
}

pagescroll(pagecontrol, samepart, dh, dv)
ControlHandle pagecontrol;
int samepart, dh, dv;
{
    Point mousePt;
    long tickcount, ticks; /* long ex. SUMACC */

    do {
        GetMouse(&mousePt);
        tickcount = TickCount();
        if (TestControl(pagecontrol, mousePt) != samepart)
            continue;
        scrollTE(paswind, dh, dv);
        DrawControls(paswind->ptr);
        Delay((long) (4 * KeyRepThresh - (TickCount() - tickcount)), &ticks);
        /* wait for factor of key repeat interval */
    }
    while (StillDown());
}

/* the general purpose scrolling routine */
/* TODO scrollTE should have a "no-round" to telineheight option */

scrollTE(theWindp, dh, dv)
struct window * theWindp;
register int dh, dv;
{
    if (dv) {
        /* check bounds */
        int newvalue;
        int lineHeight;

        lineHeight = (*theWindp->hTE)->lineHeight;
        dv = (dv / lineHeight) * lineHeight;
        /* make vertical scroll amount an integral factor of lineHeight */

        if (dv > 0) {
            /* scroll up */
            if ((newvalue = (*theWindp->vscroll)->ctrlValue - dv) < 0) {
                dv += newvalue;
                /* move just to the top if too big a scroll */
            }
        }
        else if (dv < 0) {
            /* scroll down */
            if ((newvalue =
                ((*theWindp->vscroll)->ctrlValue - dv
                 - (*theWindp->vscroll)->ctrlMax)) > 0) {
                dv += newvalue;
                /* move just to the bottom if too big a scroll */
            }
        }
    }
    if (dh) {
        /* check bounds */
        int newvalue;

        if (dh > 0) {
            if ((newvalue = (*theWindp->hscroll)->ctrlValue - dh) < 0) {
                dh += newvalue;
                /* move just to the left edge if too big a scroll */
            }
        }
        else if (dh < 0) {
            if ((newvalue =
                ((*theWindp->hscroll)->ctrlValue - dh
                 - (*theWindp->hscroll)->ctrlMax)) > 0) {
                dh += newvalue;
                /* move just to the right edge if too big a scroll */
            }
        }
    }
    if (dh == 0 && dv == 0)
        return(0);
    TEScroll(dh, dv, theWindp->hTE);

    /* direct control struct access is used rather than
       Set & GetCtrlValue for speed */
    (*theWindp->vscroll)->ctrlValue -= dv;
}

```

```

(*thewindp->hscroll)->ctrlValue -= dh;

/* DrawControls(thewindp->ptr);
   ----- too slow on repeated small moves to be in the loop */
}

newvctlmax(thewind)
struct window * thewind;
{
    register TERec * textptr;
    register int length;
    register int newmax;

    textptr = *(thewind->hTE);
    length = (textptr->nLines + 1) * textptr->lineHeight;
    if (textptr->destRect.top + length < textptr->viewRect.bottom)
        /* there is white space at the bottom, size of text above is max */
        newmax = - textptr->destRect.top;
    else
        /* total pixel length - pagesize */
        newmax = length - (textptr->viewRect.bottom - textptr->viewRect.top);
    if (newmax < 0)
        newmax = 0;
    return(newmax);
}

long clickscroll()
{
    Point mouseloc;
    int telineheight;
    int touched;

    /* save regs first */
    #asm
    movem.l d1/d3/a1/a2,-(SP)
    #endasm

    touched = FALSE;
    telineheight = (*paswind->hTE)->lineHeight;
    GetMouse(&mouseloc);

    if (mouseloc.v < (*paswind->hTE)->viewRect.top) {
        scrollTE(paswind, 0, telineheight);
        touched = TRUE;
    }
    else if (mouseloc.v > (*paswind->hTE)->viewRect.bottom) {
        scrollTE(paswind, 0, - telineheight);
        touched = TRUE;
    }
    if (mouseloc.h < (*paswind->hTE)->viewRect.left) {
        scrollTE(paswind, 50, 0);
        touched = TRUE;
    }
    else if (mouseloc.h > (*paswind->hTE)->viewRect.right) {
        scrollTE(paswind, - 50, 0);
        touched = TRUE;
    }

    if (touched)
        DrawControls(paswind->ptr);
    telineheight = 0;
    /* useless statement to get compiler to recognize next asm statement! */

    #asm
    movem.l (SP)+,d1/d3/a1/a2
    #endasm
    return(1L);
}

```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "mainmenu.h"
#include "comm.h"
#include "commdefs.h"
#include "menudefs.h"
#include "filemenudefs.h"
#include "windmenudefs.h"
#include "commmenudefs.h"
#include "echomenudefs.h"
#include "editmenudefs.h"
#include "uw.h"
#include "vi.h"
#include "videfs.h"
#include "mac.h"

#define NEG      (-1)
#define VER1CONF 2
#define VER2CONF 3
#define CURCONF  3

#define CONFID   0
#define DEFFONT  8      /* Mona */
#define DEFSIZE  9

Str255 inputstring = "\PInput  (<)";
Str255 outputstring = "\POutput (>)";
int menurectsize = 0;           /* area of largest menu, for enoughmem test */
RgnHandle updateRgn;
Handle brackhand;  /* handle to bracket table resource */

extern Handle cuthandle; /* holds all cut text */

extern unsigned char copynotice[];
extern int readlag;
extern int writelag;
extern int wrapwidth;
extern char controlmode;
extern char doflow;
extern char pagewait;
extern char useTEKey;

extern struct virecord vr;

short deffont = DEFFONT;        /* default font */
short defsize = DEFSIZE;        /* default font size */
short defvifont = DEFFONT;      /* default font */
short defvysize = DEFSIZE;      /* default font size */
int defvirows = VIROWS;
int defvicols = VICOLS;

#ifndef DORES
short resfid = NULL;
short appresfid = NULL;
short appcount = 0;             /* number of files to open */
#endif

struct conf {
    short version;            /* version of configuration resource */

    /* vi parameters */
    short virows;              /* default rows */
    short vicols;              /* default columns */
    short vifont;              /* default font */
    short vifsize;              /* default size */
    /* text */
    short font;                /* default font */
    short size;                /* default size */
    /* echo modes */
    char nocontrolmode;
    char litout;
    char bstodel;
    char decesc;
    char crlf;
    char unixwind;
    char doflow;
    short lagin;
    short layout;
    short outwrap;
    short termwrap;
    /* modem port parameters */
    short modport;
    short modspeed;
    short moddata;
    short modstop;
    short modparity;
    /* printer port parameters */

```

```

short prport;
short prspeed;
short prdata;
short prstop;
short prparity;
char pagewait;
char usetekey; /* use TEKey */
};

getconf()
{
    Handle hconf;
    struct conf * confp;

    brackhand = GetResource('BRCK', 0);
    hconf = GetResource('CONF', CONFID);
    if (hconf == (Handle) NULL) {
        hconf = NewHandle((Size) sizeof (struct conf));
        HLock(hconf);
        confp = (struct conf *) *hconf;
        defcomminit(confp);
        /* do rudimentary default communications initialization */
        HUnlock(hconf);
        return(-1);
    }
    HLock(hconf);
    confp = (struct conf *) *hconf;

    if (confp->version < VER1CONF)
        errprompt("obsolete configuration not used");
    if (confp->version >= VER1CONF) {

        /* first the echo menu values */
        /* set the numeric values */
        readlag = confp->largin;
        writelag = confp->lagout;
        wrapwidth = confp->outwrap;
        /* viwrap = confp->termwrap; */

        /* set states */

        /* font types and sizes */
        defsize = confp->size;
        deffont = confp->font;
        defvifont = confp->vifont;
        defvisize = confp->vifsize;
        defvirows = confp->virows;
        defvicols = confp->vicols;

        /* make the basic windows */
        windsetup();

        /* echo states */
        if (confp->nocontrolmode)
            echomenu(SWITCHLOCK);
        if (confp->litout)
            echomenu(LITOUT);
        if (confp->bstodel)
            echomenu(BSTODEL);
        if (confp->decesc)
            echomenu(DECESC);
        if (confp->crlf)
            echomenu(SWITCHNL);
        if (confp->unixwind)
            echomenu(UWToggle);
        if (!confp->doflow)
            echomenu(DOFLow);
        if (confp->pagewait)
            filemenu(PAGEWAIT);

        /* now do serial parameter setup */

        defport = &modem; /* set default port set to modem */

        /* printer */
        printer.speed = confp->prspeed;
        printer.parity = confp->prparity;
        printer.data = confp->prdata;
        printer.stop = confp->prstop;
        if (confp->prport) {
            commmenu(MODEMOFF); /* switch to printer config */
            commmenu(confp->prport); /* turn on the port */
            commmenu(MODEMOFF); /* switch to modem config */
        }

        /* modem */
    }
}

```

```

modem.speed = confp->modspeed;
modem.parity = confp->modparity;
modem.data = confp->moddata;
modem.stop = confp->modstop;
if (confp->modport) {
    commmenu(confp->modport); /* turn on the port */
}
resetcomm(); /* fix comm menu parameters */
}

if (confp->version >= VER2CONF) {
    if (!confp->useTEkey)
        editmenu(SELKILL);
}
HUnlock(hconf);
ReleaseResource(hconf);
if (resfid != appresfid)
    CloseResFile(resfid);
}

/* set a conf struct for current echo & comm menu values & options */

setconf(hconf)
Handle hconf;
{
    struct conf * confp;
    long ticks;

    if (hconf == (Handle) NULL)
        return(-1);
    /* HLock(hconf); */
    confp = (struct conf *) *hconf;

    confp->version = CURCONF;

    /* set the numeric values */
    confp->font = (*inputwind->hTE)->txFont;
    confp->size = (*inputwind->hTE)->txSize;
    confp->vifont = inputwind->ptr->txFont;
    confp->vifsize = inputwind->ptr->txSize;
    confp->virows = vr.virows;
    confp->vicols = vr.vicols;

    confp->llogin = readlag;
    confp->lagout = writelag;
    confp->outwrap = wrapwidth;
    confp->termwrap = vr.viwrap;

    /* get states */
    confp->nocontrolmode = !controlmode;
    confp->litout = literal;
    confp->bstdodel = bstodel;
    confp->decesc = decesc;
    confp->crlf = switchnl;
    confp->unixwind = uwon;
    confp->doflow = dofow;
    confp->pagewait = pagewait;

    /* printer */
    confp->prspeed = printer.speed;
    confp->prparity = printer.parity;
    confp->prdata = printer.data;
    confp->prstop = printer.stop;
    if (printer.in == -6)
        confp->prport = MODEMPORT;
    else if (printer.in == -8)
        confp->prport = PRINTERPORT;
    else
        confp->prport = 0;

    /* modem */
    confp->modspeed = modem.speed;
    confp->modparity = modem.parity;
    confp->moddata = modem.data;
    confp->modstop = modem.stop;
    if (modem.in == -6)
        confp->modport = MODEMPORT;
    else if (modem.in == -8)
        confp->modport = PRINTERPORT;
    else
        confp->modport = 0;
    confp->useTEkey = useTEKey;
    /* HUnlock(hconf); */
}

/* open the first document resource file */
openlaunchres()

```

```

{
    short newresfid;
    short mess;
    short refnum;
    AppFile appfile;

    resfid = appresfid = CurResFile();
    /* TODO add code to open startup resource file */
    CountAppFiles(&mess, &appcount);
    if (mess == appOpen) {
        if (appcount > 1) {
            GetAppfiles(1, &appfile);
            /* NOTE only do first one! */
            SetVol((StringPtr) NULL, appfile.vRefNum);
            if ((newresfid = OpenRFPerm(appfile.fName, appfile.vRefNum, fsRdWrPerm) ) == NEG ) {
                return(-1);
            }
            else
                resfid = newresfid;
        }
    }
    /* TODO handle "please print file" case */
    return(0);
}

#ifndef NECESSARY
/* allow user to get a different configuration */

openresource()
{
    if (resfid)
        CloseResFile(resfid);

    /* use get file code from menu.file.c, needs to be functionalized */
}
#endif

saveconf()
{
    int newresfid;
    Point here;
    SFReply reply;
    OSType type, creator;
    short confid;
    Handle hconf;
    Str255 title;

    here.v = 30;
    here.h = 40;

    GetWTitle(inputwind->ptr, title);
    SFPutFile(here, "\PSave configuration in:", title,
              (ProcPtr) NULL, &reply);
    if (reply.good) {
        FInfo finfo;

        if ( (newresfid = OpenRFPerm(reply.fName, reply.vRefNum, fsRdWrPerm) ) == NEG) {
            /* make new resource file */
            SetVol((StringPtr) NULL, reply.vRefNum);
            CreateResFile(reply.fName);
            if (ResError()) {
                return(-1);
            }
            newresfid = OpenRFPerm(reply.fName, reply.vRefNum, fsRdWrPerm);
            if (newresfid == NEG && ResError()) {
                return(-1);
            }
        }
        resfid = newresfid;

        if ( (hconf = GetResource('CONF', CONFID)) != (Handle) NULL) {
            /* if it exists, kill it */
            RmveResource(hconf);
            UpdateResFile(resfid);
            DisposHandle(hconf);
        }
        /* now add new resource */
        if ( (hconf = NewHandle( (Size) sizeof(struct conf))) == (Handle) NULL) {
            return(-1);
        }
        AddResource(hconf, 'CONF', CONFID, "\P");
        if (ResError()) {
            DisposHandle(hconf);
            return(-1);
        }
        setconf(hconf);
    }
}

```

```

ChangedResource(hconf);
UpdateResFile(resfid);
if (resfid != appresfid)
    CloseResFile(resfid);
if (ResError()) {
    ReleaseResource(hconf);
    DisposHandle(hconf);
    return(-1);
}
if (resfid != appresfid) {
    /* fix the file type and creator if it's not the application itself */
    GetFInfo(reply fName, reply.vRefNum, &finfo);
    finfo.fdType = 'TEXT';
    finfo.fdCreator = 'dumv';
    SetFInfo(reply fName, reply.vRefNum, &finfo);
}
ReleaseResource(hconf);
DisposHandle(hconf);
}
return(0);
}

#endif

defcomminit(confp)
struct conf * confp;
{
    defport = &modem; /* set default port set to modem */
    /* TODO set by default to 1200 etc.--should get parameters from disk? */

    modem.speed = baud1200;
    printer.speed = baud9600;
    printer.data = modem.data = data7;
    printer.parity = modem.parity = noParity;
    printer.stop = modem.stop = stop20;

    /* make the windows */
    windsetup();
}

/* setup windows */

windsetup()
{
    extern struct window * makewindow();

    viwind = inputwind = outputwind = makewindow(0, TEXTWIND, deffont, defsize, 38, 1, 0, 0);
    BlockMove(&viwind->virec, &vr, (Size) sizeof(struct virecord));
    /* set up the viwind current global context structure */
    SelectWindow(inputwind->ptr);
    ShowWindow(inputwind->ptr);

    bugwind = makewindow(BUGWINDNUM, BUGWIND, deffont, defsize, 140, 20, 0, 0);
    ShowWindow(bugwind->ptr);
    /* HNoPurge((inputwind->hscroll)->contrlDefProc & 0x00ffff); */
    /* don't let the scrollbar proc get dumped ! You always need it again */
}

/* the primary initialization routine */
extern GrafPtr wport;
extern VBLTask gateVBL;
extern void gatekeeper();

#define CURTEXT 256
#define CURVI 257
#define CURTEK 258

init()
{
    Ptr * nilptr; /* will have value NIL */
    Ptr * stackbaseptr;
    /* address of low memory global holding stack address */
    short * romptr;
    /* pointer to version of ROM being used */
    extern setshake();
    extern long scraptote();
    extern CursHandle curtext;
    extern CursHandle curvi;
    extern CursHandle curtek;

    /* FlushEvents(everyEvent,0); not really desirable? */

#ifndef NEED
    /* set application limit to give more room for stack */
    stackbaseptr = (Ptr *) 0x908;
    SetApplLimit((Ptr) (*stackbaseptr - 8192));
    /* 8K for stack */

```

```

#endif
romptr = (short *) 0x28e;
romtype = *romptr;

nilptr = (Ptr *) NULL;
*nilptr = (Ptr) 1;
/* will cause 02 error if null handle dereferenced */

/* miscellaneous system setup */
SetEventMask(everyEvent);
/* fix event mask for autokeycommand--may cause full event queue! */
GetWMgrPort(&wport);
SetCursor(&qd.arrow);
curtext = GetCursor(CURTEXT);
curtek = GetCursor(CURTEK);
curvi = GetCursor(CURVI);

watch = GetCursor(watchCursor);
scraptote(); /* get scrap */
updateRgn = NewRgn();
cuthandle = NewHandle((Size) 0);

/* application initialization */

menuinit();
keyinit();
openlaunchres();
getconf();

#ifndef VRT
/* set up the input lag timer */
gateVBL.qLink = (VBLTask *) NULL;
gateVBL.Qtype = (short) vType;
gateVBL.vblAddr = gatekeeper;
gateVBL.vblCount = (short) 1;
gateVBL.vblPhase = (short) 0;
VInstall(&gateVBL);
#endif
}

menuinit()
{
    int count;
    register unsigned char * theptr;
    register long maxsize, newsize;

    maxsize = 0;

    for (theptr = copynotice; *theptr; theptr++)
        *theptr >= 1;

    menu[APPLE] = GetMenu(appleMenu);
    /* AppendMenu(menu[APPLE], "\PAbout dv...;Free memory/-;Key->Menu     Space-/ ;(-");*/
    AddResMenu(menu[APPLE], 'DRV1');

    menu[FILE] = GetMenu(fileMenu);
    menu[EDIT] = GetMenu(editMenu);
    menu[CURSOR] = GetMenu(cursorMenu);
    menu[WIND] = GetMenu(windMenu);
    menu[MECHO] = GetMenu(echoMenu);
    menu[COMM] = GetMenu(commMenu);
    menu[FONT] = GetMenu(fontMenu);
    AddResMenu(menu[FONT], 'FONT');
    menu[SIZE] = NewMenu(sizeMenu, "\PSize");
    menu[TOP] = GetMenu(topMenu);

    for (count = 0 ; count < NUMMENUS ; count++) {
        if (menu[count] == (MenuHandle) NULL) {
            errprompt("Bad menu");
            continue;
            // quit();
        }
        InsertMenu(menu[count], 0);
        newsize = (*menu[count])->menuWidth * (*menu[count])->menuHeight;
        if (maxsize < newsize)
            maxsize = newsize;
    }
    menurectsize = (maxsize / 8L) + 4000; /* bytes in the largest menu rect */
    DrawMenuBar();
    /* setsizemenu( short applFont, thePort->txSize); */

    SetItem(menu[WIND], INPUT, inputstring);
    SetItem(menu[WIND], OUTPUT, outputstring);
}

```

```
long
scraptote()
{
    long size;
    long offset;

    if ( (size = GetScrap( (Handle) NULL, 'TEXT', &offset) ) > 0 ) {
        if (enoughmem(size * 2)) {
            /* * 2 for extra copy GetS makes */
            size = GetScrap(TEScrHandle, 'TEXT', &offset);
            if (size >= 0) {
                TEScrLen = size;
                return(size);
            }
        }
    } else if (size == noTypeErr)
        return(0L);
}
```

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

/*
 * Desk accessories have weird names that begin with a leading zero.
 * Since this fools the automatic C/Pascal string conversion stuff, we
 * have this little gem; given a C string, it returns a Pascal string
 * with leading zero. Luckily I know of no other dependencies on embedded
 * zeros in Macintosh strings.
 */
char *DName(s)
    char *s;
{
    static char ps[32];
    int count;
    register char *cp,*dp;

    count = 1;
    cp = s;
    dp = &ps[2];
    while ((*dp++ = *cp++))
        count++;
    ps[0] = count;
    return (isapstr(ps));
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "defs.h"
#include "comm.h"
#include "utildefs.h"

extern char controlmode;
extern char allcontrol;

#define DSSETNUM    1000 /* the "set a number" dialog */
#define DSSET        1
#define DSCANCEL    2
#define DSPLUS      3
#define DSMINUS     4
#define DSPROMPT    5
#define DSNUM       6

dosetnumdial(oldnum)
{
    long newnum;
    short type;
    short numitem;
    Str255 numstring; /* Str255 ex. SUMACC */
    Rect box;
    Handle numhandle;
    DialogPtr numdial;

    if ( (numdial = GetNewDialog(DSSETNUM, (ProcPtr) NULL, (WindowPtr) -1) )
        == NULL)
        return(-1);

    GetDItem(numdial, DSNUM, &type, &numhandle, &box);
    SelIText(numdial, DSNUM, 1000, 1000);

    newnum = oldnum;
    SelIText(numdial, DSNUM, 1000, 1000);

    while (TRUE) {
        NumToString(newnum, numstring);
        SetIText(numhandle, numstring);
        if (newnum == 0)
            /* set select at end */
            SelIText(numdial, DSNUM, 1000, 1000);
        ModalDialog( (ProcPtr) NULL, &numitem);
        switch (numitem) {
            case DSSET: {
                DisposDialog(numdial);
                return( (int) newnum);
                break;
            }
            case DSCANCEL: {
                DisposDialog(numdial);
                return( oldnum);
                break;
            }
            case DSPLUS: {
                newnum++;
                break;
            }
            case DSMINUS: {
                --newnum;
                break;
            }
            case DSPROMPT: {
                break;
            }
            case DSNUM: {
                GetIText(numhandle, numstring);
                StringToNum(numstring, &newnum);
                break;
            }
        }
    }
}

longtoad(string, num)
char * string;
long num;
{
    char sign;
    char backarr[20];
    register char * charr, * end;

    sign = FALSE;

```

```

if (num == 0) {
    *(string++) = '0';
    *string = '\0';
    return;
}
if (num < 0) {
    sign = TRUE;
    num = -num;
}
end = &backarr[20];
/* take decimals off the right end of the number */
/* and stick them in this "reverse" order into backarr */
for (charp = backarr; charp < end && num > 0; num /= 10)
    *(charp++) = num % 10 + '0';
/* go backwards through backarr, adding chars to string */
if (sign)
    *(string++) = '-';
for(charp--; charp >= backarr; )
    *(string++) = *(charp--);
*string = '\0';
return;
}

```

```
#ifdef LINEPRT
```

```
Rect charrect;
Rect prtrect;
Point prtloc;
```

```
/* get a line from the user, putting the input in the menu bar area */
```

```
getline(buf, length)
char * buf;
int length;
{
    char thechar;
    char keycode;
    EventRecord evt;
    int current;

    current = 0;
    getwport(0);

    MoveTo(prtloc.h, prtloc.v);
    charrect.top = prtloc.v - 8;
    charrect.left = prtloc.h;
    charrect.bottom = prtloc.v + 2;
    charrect.right = prtloc.h + 6;
    InvertRect(&charrect);
```

```

while (TRUE) {
    while (!GetNextEvent(keyDownMask|autoKeyMask, &evt))
        SystemTask();
    thechar = evt.message & 0xff;
    keycode = (evt.message >> 8) & 0xff;
    if (evt.modifiers & cmdKey) {
        /* command key down, interpret control chars */
        switch (thechar) {
            case 'U':
            case 'u': {
                /* clear entry */
                while (current) {
                    prtdel(buf[--current]);
                }
                continue;
            }
            case 'C':
            case 'c': {
                /* completely punt this line */
                buf[0] = '\0';
                retwport();
                menurestore();
                return(-1);
            }
            default:
                /* ignore control chars */
                continue;
        }
    }
    if (controlmode && evt.modifiers & optionKey) {
        /* make a control key */
        if (makecontrol(&thechar, keycode))
            continue;
    }
    else {

```

```

/* interpret non-command control chars */
if (thechar == CR) {
    buf[current] = '\0';
    EraseRect(&charrect);
    /* move cursor to beginning? */
    retwport();
    menurestore();
    return(current);
}
else if (thechar == BS) {
    if (current) {
        length++;
        prtdel(buf[--current]);
    }
    continue;
}
if (length) {
    --length;
}
else {
    /* no space in destination buffer */
    beeper();
    continue;
}
prtchar(thechar);
buf[current++] = thechar;
}

/* print a character on the print line */

prtchar(thechar)
char thechar;
{
    int width = 6;

    if (thechar == TAB)
        width = 18;
    EraseRect(&charrect);      /* blank cursor */
    DrawChar(thechar);
    charrect.left += width;
    charrect.right += width;
    InvertRect(&charrect); /* invert cursor */
}

/* bs and blank a character on the print line */

prtdel(thechar)
unsigned char thechar;
{
    int width = 6;

    EraseRect(&charrect);      /* blank cursor */
    if (thechar == TAB) {
        width = 18;
    }
    charrect.left -= width;
    charrect.right -= 6;
    EraseRect(&charrect);      /* blank deleted char */
    Move(-width, 0);
    if (thechar == TAB)
        charrect.right = charrect.left + 6;
    InvertRect(&charrect); /* invert cursor */
}

/* uses quickdraw calls into top of em window--must have 6 bit wide font as
   the default window font */

extern unsigned long clearprompt; /* clear the message line when this tickcount reached */

menurestore()
{
    clearprompt = 0L;
    DrawMenuBar();
    if (printer.transon)
        drawflag(&printer);
    if (modem.transon)
        drawflag(&modem);
    if (allcontrol)
        invertmenu();
}

```

```

promptclear()
{
    getwport(0);
    prtrect.top = 0;
    prtrect.left = 0;
    prtrect.bottom = 19;
    prtrect.right = qd.screenBits.bounds.right - 30;

    EraseRect(&prtrect);
    retwport();
}

invertmenu()
{
    getwport(0);
    prtrect.top = 0;
    prtrect.left = 0;
    prtrect.bottom = 19;
    prtrect.right = qd.screenBits.bounds.right - 30;

    InvertRect(&prtrect);
    retwport();
}

/* Print an error message on the prompt line, setting a timer which will cause
it to be wiped out in 3 seconds. */

/* not to be supported? */
errprompt(string)
char *string;
{
    clearprompt = TickCount() + 180;
    prompt(string);
}

prompt(string)
char * string;
{
    int length;

    promptclear();
    ObscureCursor();
    getwport(0);

    /* draw it */
    MoveTo(10, 14);
    length = mystrlen(string);
    DrawText(string, 0, length);

    /* save the most recent penloc for getline() */
    prtloc.h = qd.thePort->pnLoc.h;
    prtloc.v = qd.thePort->pnLoc.v;

    retwport();
}

mygetchar()
{
    char thechar;
    char keycode;
    EventRecord evt;

    while (!GetNextEvent(keyDownMask|autoKeyMask, &evt))
        ;
    thechar = evt.message & 0xff;
    keycode = (evt.message >> 8) & 0xff;
    if (Controlmode && evt.modifiers & optionKey) {
        /* make a control key */
        if (makecontrol(&thechar, keycode))
            thechar = 0;
    }
    return(thechar);
}
#endif

mystrlen(string)
char * string;
{
    register char * strp;

    strp = string;
    while (*strp != '\0')
        strp++;
}

```

```

    return(strp - string);
}

char fstring[FINDMAX + 1] = "dumb virtue copyright (c) 1986, 1987 by Kevin Eric Saunders";
findstring(texthand, foundat, dir)
TEHandle texthand;
long * foundat;
int dir;
{
    char * text;
    int start;
    int end;
    int length;
    int count;
    int offset;
    int skip;

    count = mystrlen(fstring); /* TODO should be eliminated */

    text = (*texthand)->hText;
    start = (*texthand)->selStart;
    end = (*texthand)->selEnd;
    length = (*texthand)->teLength;
    skip = (start == end) ? 0 : 1;
    if (dir == FORWARD) {
        if ((offset = match(fstring, text + start + skip, text + length, dir)) >= 0) {
            *foundat = start + skip + offset;
            return(count);
        }
        /* Not found on first try; wrap and search to initial position */
        if ((offset = match(fstring, text, text + start + count, dir)) >= 0) {
            *foundat = offset;
            return(count);
        }
    } else {
        if ((offset = match(fstring, text, text + start, dir)) >= 0) {
            *foundat = offset;
            return(count);
        }
        /* Not found on first try; wrap and search to initial position */
        if ((offset = match(fstring, text + start, text + length, dir)) >= 0) {
            *foundat = start + offset;
            return(count);
        }
    }
    /* no match */
    return(0);
}

/* find a match, returning the offset from the start value */

match(string, startp, endp, dir)
char * string;
register char * startp;
register char * endp;
int dir;
{
    register int count;
    register int total;
    register char * strp;
    register char * fstrp;

    count = 0;
    total = mystrlen(string);
    if (dir == FORWARD) {
        /* search to end */
        strp = startp;
        for (fstrp = string; strp < endp; strp++) {
            if (*fstrp == *strp) {
                /* contents equal, try next match */
                if (++count == total) {
                    /* search succeeds, return offset from start */
                    return(strp - startp - count + 1);
                }
                fstrp++;
                continue;
            }
        }
        else {
            /* a number of matches have failed */
            fstrp = string;
            strp -= count;
            count = 0;
        }
    }
}
```

```

        }
    } else {
        /* search backwards */
        strp = endp;
        fstrp = string + total - 1;
        /* on last character in string */
        while (-strp > startp) {
            if (*fstrp == *strp) {
                /* contents equal, try next match */
                --fstrp;
                if (++count == total) {
                    /* search succeeds, return offset from start */
                    return(strp - startp);
                }
            }
            continue;
        }
        else {
            /* a number of matches have failed to pan out */
            fstrp = string + total - 1;
            strp += count;
            count = 0;
        }
    }
    return(-1);
}

/* ring de bell, but dont' distoib de nayboors! */
beeper()
{
    static long nextbeep;

    if (TickCount() > nextbeep) {
        SysBeep(4);
        nextbeep = TickCount() + 20;
    }
}

#define SLACK 2000

enoughmem(amount)
int amount;
{
    extern char * ramout;

    if (amount > (FreeMem() - SLACK)) {
        errprompt(ramout);
        return(FALSE);
    }
    else return(TRUE);
}

hilock(thehand)
Handle thehand;
{
    MoveHHi(thehand);
    HLock(thehand);
}

```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "main.h"
#include "comm.h"
#include "defs.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "windmenudefs.h"
#include "windmenu.h"
#include "uw.h"
#include "startup.h"

/* interpret a unix windows command byte; returns TRUE if output window changed */

char addmetabit = FALSE;

uwhandle(thechar)
register unsigned char thechar; /* no sign extension, thank you */
{
    register int windnum;

    if ((thechar & CB_DIR) == CB_DIR_MTOH)
        /* undesired echo of mac command sequence */
        return(FALSE);
    windnum = thechar & CB_WINDOW;
    switch (thechar & CB_FN) {
        case CB_FN_NEWW: {
            /* make new window */
            if (makewindow(windnum, VIWIND, deffont, defsize, 0, 0, 0, 0))
                ShowWindow(window[windnum].ptr);
            window[windnum].cstate |= WS_UW;
            break;
        }
        case CB_FN_KILLW: {
            /* kill a window */
#ifdef DUNNO
            if (windnum == 7)
                killwindow(viwind);
            else
                killwindow(&window[windnum]);
#endif
            window[windnum].cstate &= ~WS_UW;
            break;
        }
        case CB_FN_OSELW: {
            /* select a window for output, don't make it the top wind */
            struct window * oldoutput;
            extern char escapemode;

            oldoutput = outputwind;
            outputwind = &window[windnum];

            if (outputwind != oldoutput) {
                escapemode = outputwind->escapemode;
                return(TRUE);
            }
            break;
        }
        case CB_FN_META: {
            /* add high bit to next char */
            addmetabit = TRUE;
            break;
        }
        case CB_FN_CTLCH: {
            /* low 3 bits hold control char */
            break;
        }
        case CB_FN_MAINT: {
            /* entry, ? anything else ? */

            if ((thechar & CB_MF) == CB_MF_ENTRY) {
                uwcmd(CB_FN_ISELW, 1);
                window[1].cstate |= WS_UW;
/* MVI TODO should go through window list, make all open windows? */
                if (window[VIWINDNUM].ptr != NULL)
                    uwcmd(CB_FN_NEWW, 0);
                /* fix window menu so no output marker */
                if (oldin == oldout)
                    SetItemMark(menu[WIND], oldin, '<');
                else {
                    SetItemMark(menu[WIND], oldout, '\000');
                }
            }
            break;
        }
    }
}

```

2/6/88 6:42

uwint.c

2

```
    return(FALSE);  
}  
  
uwcmd(command, code)  
{  
    portWrite(&modem, SOH);  
    portWrite(&modem, (char) (CB_DIR_MTOH | command | (code & CB_WINDOW)));  
    /* WARNING--CB_CC etc. covered by this CB_WINDOW; may not hold */  
}
```

12/10/94 12:34

windcreat.c

1

```
/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "defs.h"
#include "main.h"
#include "videfs.h"
#include "vi.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "windmenudefs.h"
#include "uw.h"
#include "startup.h"

extern longtoa();

short vsRect[] = { -1, 0, 0, 0 };
short hsRect[] = { 0, -1, 0, 0 };
extern char * ramout;

struct window *
makewindow(number, type, fontnum, fontsize, wtop, wleft, wbottom, wright)
int number, type;
short fontnum;
short fontsize; /* short ex. sumacc */
int wtop, wleft, wbottom, wright; /* short ex. sumacc */
{
    register int count;
    Rect wrect;
    struct window * thewind;
    GrafPtr ografport;
    extern setviewsize();
    extern Boolean clickscroll();
    extern Rect cursrect;
    long scrmapsiz;
    int vrsizetemp;

    extern char * ramout;
    extern struct scrmap * scrmap;
#ifdef TEKEISTS
    extern Point macloc;
#endif

    if (window[number].ptr != NULL) {
        return(&window[number]);
        /* window already exists, echo struct */
    }
    /* ok to create window, it doesn't exist */
// MoreMasters();
// MoreMasters();
    /* do this here, before we get into trouble */
    GetPort(&ograpfport);
    if (type == BUGWIND) {
        number = BUGWINDNUM;
    }

    thewind = &window[number];

    /* assume reasonable default values if fed 0 for window rect */
    wrect.top = wtop ? wtop : 140 + 2 * number;
    wrect.left = wleft ? wleft : 20 - 2 * number;
    wrect.bottom = wbottom ? wbottom : qd.screenBits.bounds.bottom + 8; /* 350 */
    if (qd.screenBits.bounds.right > 520)
        /* big screen, leave space for Finder ickons */
        wrect.right = wright ? wright : qd.screenBits.bounds.right - 60;
    else
        wrect.right = wright ? wright : qd.screenBits.bounds.right + 8; /* 520 */
    if (type == BUGWIND) {
        if (newwind(thewind, FrontWindow(), "\PScratchPad", &wrect,
                    VSCROLL, 0, HSCROLL, 10000) )
            return(NULL);
    }
    else {
        longtoa(error, (long) number);
        ctop(error);
        if (newwind(thewind, FrontWindow(), error, &wrect,
                    VSCROLL, 0, HSCROLL, 10000) )
            return(NULL);
    }

    ((WindowPeek) thewind->ptr)->windowKind = userKind + TEXTWIND;
    thewind->type = type;

    /* make TE record */
    thewind->hTE = TENew(&nullrect, &nullrect);
    TEActivate(thewind->hTE);
    thewind->scroll = TRUE;
    /* (*thewind->hTE)->crOnly = -1; TODO no wrap? */
}
```

```

/* set the text at the right position */
(*thewind->hTE)->destRect.top = TEOFSET;
(*thewind->hTE)->destRect.left = TEOFSET;
tesetsel((long) 0, (long) 0, thewind->hTE);
TEKey(BS, thewind->hTE);
/* fix TE cursor positioning problem--strange but true! */
(*thewind->hTE)->clikLoop = clickscroll;
/* setup mouse autoscrolling */

#ifndef UW
#endif DUNNO
/* only make a new window when user sets output */
if (uwon) {
    if (number > 1 && number < NWINDOW)
        /* don't do window 7, the VI window */
        uwcmd(CB_FN_NEWW, number);
}
#endif
#endif
thewind->cstate = 0;
thewind->cstate |= WS_INPUT;
thewind->cstate |= WS_LOCAL;
/* make Input window and Edit local mode by default */

/* make the vi window */
/*
wrect.top = wtop ? wtop : (qd.screenBits.bounds.bottom < 350 ? 20 : 38);
wrect.left = wleft ? wleft : 0;
wrect.bottom = wbottom ? wbottom : qd.screenBits.bounds.bottom + 8;
wrect.right = wright ? wright : qd.screenBits.bounds.right + 8;
*/
thewind->virec.virows = defvirows;
thewind->virec.vicols = defviccols;
thewind->virec.visize = thewind->virec.virsize
    = thewind->virec.virows * thewind->virec.vicols;
thewind->virec.virowmin = 0;
thewind->virec.virowmax = VT100ROWS - 1;
thewind->virec.vicolmax = VICOLS - 1;
thewind->virec.lineheight = 12;
thewind->virec.fontheight = 9;
thewind->virec.fontdescent = 2;
thewind->virec.viwrap = 500;

/* TODO make this a function if you want more vi windows */
/* initialize screen map */
/* set virecord up here */
ResrvMem( (long) (2 * thewind->virec.virsize));
if ( !(thewind->virec.vitextbase = NewPtr((long) (2 * thewind->virec.virsize)) ) ) {
    errstr(ramout);
    DisposPtr(thewind->ptr);
    thewind->ptr = NULL;
    return(NULL);
}

thewind->virec.vitextp = thewind->virec.vitextbase;
thewind->virec.vitextend = thewind->virec.vitextp + thewind->virec.virsize;

vrsizetemp = vr.virsize;
vr.virsize = thewind->virec.virsize;
blankchars(thewind->virec.vitextp, thewind->virec.virsize);
vr.virsize = vrsizetemp;
/* fill text with spaces, gotta set up blankchars first */

#ifndef OLD
thewind->virec.vitextend = thewind->virec.vitextbase + thewind->virec.virsize;
while (thewind->virec.vitextp < thewind->virec.vitextend) {
    *thewind->virec.vitextp++ = ' ';
}
while (thewind->virec.vitextp < thewind->virec.vitextend) {
    *thewind->virec.vitextp++ = '\0';
}

thewind->virec.vitextp = thewind->virec.vitextbase; /* reset to beginning */
thewind->virec.vitextend = thewind->virec.vitextbase + thewind->virec.virsize;
#endif

/* should set cursor?
vr.cursrect.top = vr.cursrect.left
    = vr.cursrect.bottom = vr.cursrect.right = 0;
*/

/* make space for the scrmap array */
scrmapsize = thewind->virec.virows * sizeof(struct scrmap);
ResrvMem(scrmapsize);
if ( !(thewind->virec.scrmap = (struct scrmap *) NewPtr(scrmapsize))) {
    errstr(ramout);
    DisposPtr(thewind->ptr);
}

```

```

DisposPtr(thewind->virec.vitextbase);
thewind->ptr = NULL;
return(NULL);
}
count = thewind->virec.virows;
while (count--) {
    thewind->virec.scrmap[count].hit = thewind->virec.scrmap[count].drawn = 0;
}

#ifndef TEKEXISTS
macloc.h = 2;
macloc.v = 13;
#endif

/* set font -- done earlier now */
SetFont(thewind, fontnum);
SetFontSize(thewind, fontsize);
SetViewSize(thewind);
thewind->fvol = 0;
thewind->fdir = 0;

/* if (thewind->type == VIWIND) */
scrbarmove(thewind); /* TOTEST fix blank viscroll? */

SetItemStyle(menu[WIND], number + WINDMBASE, 0);
/* make font normal to show window exists */
SetPort(ografport);
return(thewind);
}

killwindow(wind)
struct Window * wind;
{
    int windnum;

    windnum = wind - window;
    switch (wind->type) {
        case TEXTWIND: {
            /* don't kill active writing windows! */
            if (wind == inputwind || wind == outputwind )
                return;
            /* should check a window-dirty bit to see if prompt to save */
            TEDispose(wind->hTE);
            DisposeControl(wind->hscroll);
            DisposeControl(wind->vscroll);
            DisposeWindow(wind->ptr);
            wind->ptr = NULL;
#ifdef UW
            /* do uw stuff */
            if (uwon) {
                if ((wind->cstate & WS_UW) && windnum > 1 && windnum < NWINDOW)
                    uwcmd(CB_FN_KILLW, windnum);
            }
#endif
            SetItemStyle(menu[WIND], windnum + WINDMBASE, italic);
            /* make font italic to show window doesn't exist */
            break;
        }
        case BUGWIND: {
            SendBehind(bugwind->ptr, (WindowPtr) NULL);
            break;
        }
        case VIWIND: {
            /* TODO should actually dispose of window ?
             * call to kill can be put in display menu item
             */
            if (wind == outputwind)
                return(FALSE);
            DisposeControl(wind->hscroll);
            DisposeControl(wind->vscroll);
            DisposeWindow(wind->ptr);
            DisposPtr(vr.vitextbase);
            wind->ptr = NULL;
#ifdef UW
            /* do uw stuff */
            if (uwon) {
                uwcmd(CB_FN_KILLW, 7);
            }
#endif
            break;
        }
#ifndef TEKEXISTS
        case TEKWIND: {
            break;
        }
#endif
    }
}

```

```
}

#define zoomDocProc 8

newwind(thewind, behind, windname, windrect, vtype, vsiz, htype, hsize)
struct window * thewind;
WindowPtr behind;
Str255 windname;
Rect * windrect;
int vtype;
int vsiz;
int htype;
int hsize;
{
    ResrvMem((long) sizeof(WindowRecord));
    thewind->ptr = (WindowPtr) NewPtr((long) sizeof(WindowRecord));
    if (thewind->ptr == NULL) {
        errstr(ramout);
        return(TRUE);
    }
    NewWindow(thewind->ptr, windrect, windname, INVIS,
        romtype & ROM64 ? documentProc : zoomDocProc, behind, TRUE, (long) thewind);
    SetPort(thewind->ptr);
    ClipRect(&qd.thePort->portRect);

/* we always create vscroll only to maintain conformity with
 * routines in command.c; it is sometimes unused
 */
    thewind->hscroll = NewControl(thewind->ptr, (Rect *) &vsRect, "\P",
        INVIS, 0, 0, hsize, scrollBarProc, (long) htype);
    thewind->vscroll = NewControl(thewind->ptr, (Rect *) &hsRect, "\P",
        INVIS, 0, 0, vsiz, scrollBarProc, (long) vtype);
    return(FALSE);
}
```

```

/* copyright Kevin Eric Saunders 1986--All Rights Reserved */

#include "defs.h"
#include "main.h"
#include "uw.h"
#include "videfs.h"
#include "vi.h"
#include "comm.h"
#include "mainmenu.h"
#include "menudefs.h"
#include "filemenudefs.h"
#include "editmenudefs.h"
#include "echomenudefs.h"
#include "topmenudefs.h"
#include "cursormenudefs.h"

#define CONTROLSZ 16

/* placeholders used in switching to viwind & back */
struct window * oldoutputwind;

clickinwind(thewind)
register struct window * thewind;
{
    GlobalToLocal(&myEvent.where);
    switch (thewind->type) {
        case TEXTWIND:
        case BUGWIND: {
            extern struct window * paswind;

            paswind = thewind; /* set up for scrolling routines */
            if (PtInRect(myEvent.where, &(*thewind->hTE)->viewRect)) {
                /* in text */
                TEClick(myEvent.where,
                        (myEvent.modifiers & shiftKey) ? TRUE : FALSE, thewind->hTE);
            }
            else
                scrbarclick(thewind);
                /* may have been a control */
            break;
        }
        case VIWIND: {
            if (PtInRect (myEvent.where, &virect)) {
                /* TODO something to do here? */
                mvcursor(&myEvent.where);
            }
            else
                scrbarclick(thewind);
            break;
        }
#ifdef TEKEXISTS
        case TEKWIND: {
            break;
        }
#endif
    }
}

scrbarmove(thewind)
struct window * thewind;
{
    register Rect * portr;

    portr = &thewind->ptr->portRect;

    switch (thewind->type) {
#ifdef TEKEXISTS
        case TEKWIND:
#endif
        case BUGWIND:
        case TEXTWIND: {
            HideControl(thewind->vscroll);

            MoveControl(thewind->vscroll, portr->right - CONTROLSZ + 1,
                        portr->top - 1 );
            SizeControl(thewind->vscroll, CONTROLSZ,
                        portr->bottom - portr->top - CONTROLSZ + 3 ); /* was -13 */
            (*thewind->vscroll)->ctrlVis = TRUE;
            /* eq. to ShowControl(thewind->vscroll); */
            /* make control visible, but don't draw it twice ! */
            /* fall through */
        }
        case VIWIND: {
            HideControl(thewind->hscroll);
            MoveControl(thewind->hscroll, portr->left -1,
                        portr->bottom - CONTROLSZ + 1); /* was -1, +1 */
        }
    }
}

```

```

    SizeControl(thewind->hscroll,
                portr->right - portr->left - CONTROLSZ + 3, CONTROLSZ);
    (*thewind->hscroll)->ctrlVis = TRUE;
    /* eq. to ShowControl(thewind->hscroll); */
    /* make control visible, but don't draw it twice ! */
    break;
}
}

/* sizes window, fiddles with update region, sets TE & other size vars
TODO call sizewind w/ struct window * ? */

sizewindow(whichwind, height, width)
WindowPtr whichwind;
int height, width;
{
    Rect tRect;
    register struct window * thewind;
    GrafPtr oldgrafport;

    thewind = swindptr(whichwind);
    GetPort(&oldgrafport);
    SetPort(whichwind);

    /* Add the old "scroll bar area" to the update region so it will */
    /* be redrawn (for when the window is enlarged). */
    tRect.top = whichwind->portRect.top;
    tRect.bottom = whichwind->portRect.bottom;
    tRect.right = whichwind->portRect.right;
    tRect.left = tRect.right - CONTROLSZ;
    InvalRect(&tRect);

    tRect.left = whichwind->portRect.left;
    switch (thewind->type) {
        case TEKWIND:
        case VIWIND: {
            tRect.top = whichwind->portRect.bottom - CONTROLSZ;
            break;
        }
        case BUGWIND:
        case TEXTWIND: {
            tRect.top = (*thewind->hTE)->viewRect.bottom;
            break;
        }
    }
    InvalRect(&tRect);

    /* Now draw the newly sized window. */
    SizeWindow(whichwind, width, height, (Boolean) TRUE);
    ClipRect(&whichwind->portRect);
    scrbarmove(thewind);

    /* Add the new "scroll bar area" to the update region so it will */
    /* be redrawn (for when the window is made smaller). */
    tRect.top = whichwind->portRect.top;
    tRect.bottom = whichwind->portRect.bottom;
    tRect.right = whichwind->portRect.right;
    tRect.left = tRect.right - CONTROLSZ;
    InvalRect(&tRect);

    tRect.left = whichwind->portRect.left;
    /* continued; top is set in switch, trect invaled after switch */

    setviewsize(thewind);
    /* Adjust the view rectangle for TextEdit and vi */

    SetCtlMax(thewind->vscroll, newvctlmax(thewind));

    switch (thewind->type) {
        case BUGWIND:
        case TEXTWIND: {
            int newmax;

            tRect.top = (*thewind->hTE)->viewRect.bottom;
            break;
        }
        case VIWIND: {
            tRect.top = virect.bottom;
            break;
        }
    }

#endif TEKEXISTS
    case TEKWIND: {
        tRect.top = tRect.bottom - CONTROLSZ;
        break;
    }
}

```

```

        }
#endif
    }
    InvalRect(&tRect);
    SetPort(oldgrafport);
}

/* Draws the content region of the given window, after erasing whatever
   was there before */

updatewind(whichwind)
WindowPtr whichwind;
{
    GrafPtr saveport;
    struct window * thewind;
    extern int scrmap[];

    thewind = swindptr(whichwind);

    /* bracket the drawing with wmgr calls */
    GetPort(&saveport);
    SetPort(whichwind);
    /* draw it */
    if (thewind->type == VIWIND) {
        /* invalidate cursor so always drawn, before Begin Update sets visRgn */
        if (thewind != viwind) {
            vicontext(thewind);
        }
        InvalRect(&vr.cursrect);
    }
    BeginUpdate(whichwind);
    switch (thewind->type) {
#define TEKEXISTS
        case TEKWIND: {
            EraseRect(&whichwind->portRect);
            DrawGrowIcon(whichwind);
            DrawControls(whichwind);
            break;
        }
#endif
        case BUGWIND:
        case TEXTWIND: {
            EraseRect(&whichwind->portRect);
            DrawGrowIcon(whichwind);
            DrawControls(whichwind);
            TEUpdate(&(*whichwind->visRgn)->rgnBBox, thewind->hTE);
            break;
        }
        case VIWIND: {
            /* draw the stuff outside the vi screen */
            ClipRect(&whichwind->portRect);
            EraseRect(&whichwind->portRect);
            DrawGrowIcon(whichwind);
            vierasegrow();
            DrawControls(whichwind);

            /* draw the insides */
            ClipRect(&virect);
            virefresh();
            updviseen(thewind);
            FillRect(&vr.cursrect, &qd.black);
            vr.cursor = TRUE;
            break;
        }
    }
    EndUpdate(whichwind);
    SetPort(saveport);
}

/* change comm states & menus for vi */

viactivate(actviwind) /* TODO parameterize */
struct window * actviwind;
{
    int count;

    vion = TRUE;
    if (!uwon) {
        outputwind = actviwind;
    }

    /* disable all but a few options on edit menu */
    for (count = 1; count < EDITLAST; count++)

```

```

    DisableItem(menu[EDIT], count);
    for (count = 1; count < PRINTSEL; count++)
        DisableItem(menu[FILE], count);
    for (count = 1; count < CURSORLAST; count++)
        DisableItem(menu[CURSOR], count);
    for (count = 1; count < TOPLAST; count++)
        DisableItem(menu[TOP], count);

    EnableItem(menu[EDIT], COPY);
    EnableItem(menu[CURSOR], YANK);

    setblock();
    /* reset blocking factor for vi--don't wait very long for input */
}

/* take care of restoring menus & comm states of old text windows */

videactivate(deviwind)
struct window * deviwind;
{
    int count;

    vion = FALSE;

    /* enable all items on edit menu */
    for (count = 1; count < EDITLAST; count++)
        EnableItem(menu[EDIT], count);
    for (count = 1; count < PRINTSEL; count++)
        EnableItem(menu[FILE], count);
    for (count = 1; count < CURSORLAST; count++)
        EnableItem(menu[CURSOR], count);
    for (count = 1; count < TOPLAST; count++)
        EnableItem(menu[TOP], count);

    setblock();
    /* reset blocking factor */
}

#endif TEKEXISTS

/* change comm states & menus for tek window */

tekactivate()      /* TODO parameterize */
{
    int count;

#ifdef DOTHISJUNK
    if (!uwon) {
        /* do output reassignment */
        if (outputwind != tekwind)
            oldoutputwind = outputwind;
        outputwind = tekwind;
    }
#endif
    /* disable all but a few options on edit menu */
    for (count = 1; count < EDITLAST; count++)
        DisableItem(menu[EDIT], count);
    for (count = 1; count < PRINTSEL; count++)
        DisableItem(menu[FILE], count);
    for (count = 1; count < CURSORLAST; count++)
        DisableItem(menu[CURSOR], count);
    for (count = 1; count < TOPLAST; count++)
        DisableItem(menu[TOP], count);

    /* EnableItem(menu[EDIT], COPY); */
    EnableItem(menu[CURSOR], YANK);
}

/* take care of restoring menus & comm states */

tekdeactivate()
{
    int count;

#ifdef DOTHISJUNK
    if (!uwon && oldoutputwind != NULL)
        outputwind = oldoutputwind;
#endif

    /* enable all items on edit menu */
    for (count = 1; count < EDITLAST; count++)
        EnableItem(menu[EDIT], count);
}

```

```

for (count = 1; count < PRINTSEL; count++)
    EnableItem(menu[FILE], count);
for (count = 1; count < CURSORLAST; count++)
    EnableItem(menu[CURSOR], count);
for (count = 1; count < TOPLAST; count++)
    EnableItem(menu[TOP], count);
}

#endif

/* set the vars the system uses to keep track of viewscreen sizes
   according to the size of the window */

setviewsizethewind)
struct window * thewind;
{
    VIsizethewind, &virect;

    TEsizethewind, &(*thewind->hTE)->viewRect);
    if ((*thewind->hTE)->crOnly >= 0)
        /* wordwrap is on */
        (*thewind->hTE)->destRect.right =
            (*thewind->hTE)->viewRect.right - 4;
    else
        (*thewind->hTE)->destRect.right = 32000;

    switch (thewind->type) {
        case VIWIND: {
            /* set lines ? */
            ClipRect(&virect);
            break;
        }
#ifndef TEKWIND
        case TEKWIND: {
            Rect bigrect;

            bigrect.top = 0;
            bigrect.left = 0;
            bigrect.bottom = 4000;
            bigrect.right = 4000;
            ClipRect(&bigrect);
            break;
        }
#endif
    }
}

/* return the TE viewrect for the specified window */
TEsize(thewind, therect)
struct window * thewind;
Rect * therect;
{
    int telineheight;

    therect->top = thewind->ptr->portRect.top + TEOFSET;
    therect->left = thewind->ptr->portRect.left; /* was + TEOFSET; */

    /* make size integral factor of lineheight */
    telineheight = (*thewind->hTE)->lineHeight;
    therect->bottom = (
        (thewind->ptr->portRect.bottom - therect->top - CONTROLSZ + 1)
        / telineheight
    ) * telineheight
    /* == telineheight * nolines */
    + TEOFSET;
    therect->right = thewind->ptr->portRect.right - CONTROLSZ;
}

/* set the vi viewrect size for the top window */
VIsizethewind, therect)
struct window * thewind;
Rect * therect;
{
    therect->top = thewind->ptr->portRect.top + VITOP;
    therect->left = thewind->ptr->portRect.left + VILEFT;
    therect->bottom = thewind->ptr->portRect.bottom - CONTROLSZ + 1;
    therect->right = thewind->ptr->portRect.right;
}

/* this stuff might be useful someday? */
#ifndef EVENHEIGHT

```

```

/* sets overall window size to an even lineHeight */
height = (
    (height - CONTROLSZ - TEOFSET) / (*thewind->hTE)->lineHeight )
    * (*thewind->hTE)->lineHeight
)
+ CONTROLSZ + TEOFSET;
#endif

/* erase the silly vertical scrollbar frame DrawGrowIc also draws */
vierasegrow()
{
    PenPat(&qd.white);
    MoveTo(qd.thePort->portRect.right - CONTROLSZ + 1, qd.thePort->portRect.top);
    LineTo(qd.thePort->portRect.right - CONTROLSZ + 1, qd.thePort->portRect.bottom - CONTROLSZ);
    PenPat(&qd.black);
}

/* move cursor to location of mouse button down;
   the cursor is made into a crosshair when over vi window */
mvcursor(ptptr)
Point * pptr;
{
    register char * charrp;
    register count, charwidths, rend;
    int targetline, newxpos;

    targetline = (ptptr->y - VITOP) / vr.lineheight;
    if (targetline < 0)
        targetline = 0;
    else if (targetline > vr.virowmax)
        targetline = vr.virowmax;

    /* now figure out which character was selected as our new xpos */
    charrp = vr.vitextbase + (targetline * vr.vicols); /* was vircols */
    /* first char on the line in question */
    charwidths = 0;
    rend = pptr->h - vr.offset - VILEFT;
    /* TODO should this be + offset? */
    for (count = 0; rend > charwidths && count < vr.vicols; count++) {
        charwidths += CharWidth(*charrp++);
    }
    newxpos = count - 1;

    if (targetline != vr.ypos) {
        /* it's not on this line, go left and then up or down */
        count = vr.xpos + 1;
        while (--count) {
            mvleft();
        }

        if (targetline > vr.ypos) {
            /* move down */
            count = targetline - vr.ypos + 1;
            while (--count) {
                mvdown();
            }
        }
        else {
            /* move up */
            count = vr.ypos - targetline + 1;
            while (--count) {
                mvup();
            }
        }
    }

    /* now move right */
    count = newxpos + 1;
    while (--count) {
        mvright();
    }
}
else {
    /* go straight left or right */
    if (newxpos > vr.xpos) {
        count = newxpos - vr.xpos + 1;
        while (--count) {
            mvright();
        }
    }
    else {
        count = vr.xpos - newxpos + 1;
        while (--count) {
            mvleft();
        }
    }
}

```

12/7/94 11:17

windmaint.c

7

```
    }
}

char * ustring = "\033OA";
char * dstring = "\033OB";
char * rstring = "\033OC";
char * lstring = "\033OD";

mvleft()
{
    long count;

    count = 3;
    if (!modem.transon)
        FSWrite(modem.out, &count, lstring);
}

mvright()
{
    long count;

    count = 3;
    if (!modem.transon)
        FSWrite(modem.out, &count, rstring);
}

mvup()
{
    long count;

    count = 3;
    if (!modem.transon)
        FSWrite(modem.out, &count, ustring);
}

mvdown()
{
    long count;

    count = 3;
    if (!modem.transon)
        FSWrite(modem.out, &count, dstring);
}
```