KNOWLEDGE BASE FOR C3I

P.G. Ossorio and L.S. Schneider

October, 1987

LRI Report No. 40a

Volume II

Linguistic Research Institute
Box 1294, Boulder, CO 80306

KNOWLEDGE BASE FOR C3I

P.G. Ossorio and L.S. Schneider

October, 1987

LRI Report No. 40a

Volume II

Linguistic Research Institute
Box 1294, Boulder, CO 80306

SOFTWARE SOURCE LISTING

TABLE OF CONTENTS

1. PREFACE

The TT system is developed in Borland's Turbo Pascal (Version 3) and makes extensive use of various supporting library routines developed by Borland, Inc. for that environment. In deference to Borland, Inc., and its copyrights, the contents of those library routines that were used "as delivered" are incorporated by reference only. The contents of these routines may be purchased from Borland or any of its distributors at minimal cost. Library routines that were modified in any way whatsoever are included in full since there is no other source from which the modifications can be obtained. In a very small number of cases, the modifications were so minor that the routine is substantially the same as the original and its reproduction or distribution would, in spirit, compromise Borland, Inc.'s copyrights. Such routines have been marked by a vertical bar (¦) in the left margin to alert the reader that, for all intents and purposes <u>other</u> than their use in TT, they should be considered as Borland, Inc. proprietary code.

2. MAIN PROGRAM

```
{$I-,C-,K-,F20}
{
*****************************************************************************
Program:  TT                          Version 2.0          Date: 20 Nov 87
File:     tt.pas                      Version 2.00         Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                       under Contract F30602-C-85-0190

Function:  Main program.
*****************************************************************************
}

program TT;

{These are the constants for the Turbo Access Modules}

const

   MaxDataRecSize = 4000;
   MaxKeyLen      = 64;
   PageSize       = 24;
   Order          = 12;
   PageStackSize  = 8;
   MaxHeight      = 5;

{Turbo Access Modules - Borland proprietary code}

{.L-}
{$I \tp\access3.box}
{$I \tp\iotb\getkey.box}
{$I \tp\iotb\addkey.box}
{$I \tp\iotb\delkey.box}
{.L+}

{TT constants, types and variables}

{$I vars.tt}
procedure EditErrorMsg(Msgno : byte);forward;
```

```
{Buffer Management Module}

{$I bufman.tt}

procedure EditUserPush(s : VarString);forward;
procedure Pokechr(Ch : char);forward;

procedure UserError(var Msgno : byte);
{ user error handler hook }
begin
end;


procedure UserReplace(var ch : byte);
{ user replace handler hook }
begin
end;

procedure UserTask;
{ user multi-tasking hook }
begin
end;

{$I user.tt}
procedure UserUpdCmdLine;
{ user routine to display status of current window on command line}

{xxxxxxxxxxxxxxxxx,999,9999 IX:xxxxxxxx,99 S I A ** TX Log:9999 Cap:9999 **}
{12345678901234567890123456789012345678901234567890123456789012345678901234}
{0        1         2         3         4         5         6         7    }

var i,j  : integer;
    St   : Varstring;
    Num  : string[6];
    pt   : Plinedesc;

begin
   with Curwin^,StreamDef[Curwin^.Stream],BufMap[Curwin^.Stream] do
      begin
         St := UsrName;
         if Length(St) > 17 then
            begin
               while Length(St) > 16 do Delete(St,1,1);
               St := '~' + St;
            end;
         Str(RecLen,Num);
         St := St + ',' + Num;
         i := UsedRecs(DesHndl);
```

```
        Str(i,Num);
        St := St + ',' + Num + '  ';
        while Length(St) < 27 do St := St + ' ';
        if CurIx <> 0 then
           begin
           end
        else
           begin
              {St := St + 'IX:none';}
              St := St + 'IX:';
              pt := Edit(Stream,Curline,rd);
              i := pt^.BuffLen;
              Str(i,Num);
              St := St + Num;
           end;
        while Length(St) < 48 do St := St + ' ';
        with BatRec do
           if (FromStrm = Stream) and (FromRef > 0) then
              St[43] := 'S' else St[43] := ' ';
        if AI then St[47] := 'A' else St[47] := ' ';
        if InsertFlag = Insert then St[45] := 'I' else St[45] := ' ';
        if InTransaction then
           begin
              j := 0;
              for i := 1 to BufSize do
                 if Slot^[i].Dref <> 0 then
                    with Slot^[i].Dptr^ do
                       if (Flags and Dmask) <> 0 then
                          j := succ(j);
              Str(j,Num);
              St := St + '** TX Log:' + Num;
              while Length(St) < 63 do St := St + ' ';
              j := MemAvail div ((2*succ(RecLen) + SizeOf(LineDesc)) div 16);
              Str(j,Num);
              St := St + 'Cap:' + Num;
              while Length(St) < 72 do St := St + ' ';
              St := St + '**';
           end;
        EditZapCmdnam;
        EditAppCmdnam(St);
        if EditWrline (Cmdlinest, Physcrsig, Cmdcolor) then
           EditUpdrowasm (Physcrsig);
     end;
end;
procedure UserStatusline(Var wn : byte; Pwin : PWindesc);
{ user status line handler }

var whdr : textline;
```

```
    i,j  : integer;

begin
   if not (Pwin^.Formatted) then exit;      {Not a formatted file so do nothing}
   wn := 0;                                 {Tell EditUpdwinsl to do nothing   }
   With Pwin^ do                            {since we'll do it all here        }
      begin
         FillChar(whdr[1],Physcrcols,' '); {blank out the header}
         i := LeftEdge;                     {first column of SaeHeader to show}
         j := 1;                            {first column of status line}
         while (Header^[i] <> EndHdr)       {end of header indicator}
               and (j <= Physcrcols) do     {size of status line}
               begin
                  whdr[j] := Header^[i];
                  i := succ(i);
                  j := succ(j);
               end;
         if LeftEdge = 1 then whdr[1] := chr(Vertical) else
                              whdr[1] := chr(UpperRight);
         whdr[2] := chr(WindNo + ord('0'));
         whdr[3] := chr(LConnect);
         if EditWrline(whdr,Firstlineno,Bordcolor) then  {physically write it}
            EditUpdrowasm(Firstlineno);
      end;
end;


{$I screen.tt}                  {Display Management Routines}
{$I fastcom.tt}                 {cursor commands and miscellaneous others}
                                {removed from overlay area 1 because they}
                                {are referenced by other commands in area1}

{************** begin overlay area 1 ***************}

{$I debug.tt}                    {where overlay goes for debugging}
{$I init.tt}                     {initialization routines}
{$I kcmd.tt}                     {scan processing commands}
{$I ocmd.tt}                     {window processing commands}
{$I qcmd.tt}                     {text processing commands}
{$I relcmd.tt}                   {relational operator commands}
{$I sortcmd.tt}                  {sort and unique commands}
{$I slowcom.tt}                  {other commands}

{************** end overlay area 1 ****************}

procedure SeparateOverlayArea;
begin
end;
```

```
{************** begin overlay area 2 ***************}

{$I cp.tt}                          {all console processing routines}

{************** end overlay area 2 ****************}

{$I koq.tt}                         {dispatches 2nd ctrl char for ^K,^O,^Q cmds}

{$I pulldown.tt}                    {the pulldown menu system routines}

procedure UserCommand(var ch : byte);
{ user command processor hook }

const   Space = 32;
        Tilde = 126;

var pdms  : VarString;
    macfl : text;
    macnm : VarString;
    macch : char;
    WasCR : boolean;

begin
   if ch = 0 then
      begin
         ch := EditGetInput;
         if ch = 68 then
            begin
               pdms := PullDownMenu;
               EditUserPush(pdms);
            end;
         if ch = 67 then
            begin
               Asking := true;
               EditZapCmdnam;
               EditAppCmdnam('Execute File:');
               EditAskFor(macnm);
               if Abortcmd then
                  begin
                     ch := 255;
                     exit;
                  end;
               Assign(macfl,macnm);
               Reset(macfl);
               if not EditFileError then
                  begin
                     pdms := '';
```

```
                    macch := chr(Nul);
                    WasCR := false;
                    while not((ord(macch) = Tilde) or (EOF(macfl))) do
                       begin
                          read(macfl,macch);
                          {interpret upper case alpha as ctrl char}
                          if macch in ['A'..'Z'] then
                             macch := chr(ord(macch) and $1F);
                          case ord(macch) of
                                Ctrlm : begin
                                           {accept it but note that a ctrlj}
                                           {following ctrlm must be ignored}
                                           WasCR := true;
                                           pdms := pdms + macch;
                                        end;
                                Ctrlj : begin
                                           {we only accept ctrlj if the last}
                                           {char was not a CR otherwise we  }
                                           {assume it was simply a LineFeed }
                                           if not WasCR then
                                              pdms := pdms + macch;
                                           WasCR := false;
                                        end;
                                Ctrlz : begin
                                           {we accept ctrlz only if it isn't}
                                           {an end of file indicator}
                                           if not(EOF(macfl)) then
                                              pdms := pdms + macch;
                                           WasCR := false;
                                        end;
                                Space : begin end; {always ignore spaces}
                                Tilde : begin end; {pseudo EOF character}
                                else    begin
                                           pdms := pdms + macch;
                                           WasCR := false;
                                        end;
                                end;
                       end;
                    close(macfl);
                    EditUserPush(pdms);
                 end;
              Asking := false;
              EditZapCmdnam;
           end;
        ch := 255;
     end;
end;
```

```
{$I disp.tt}                        {dispatches first control character}
{$I task.tt}                        {task scheduling routines}
{$I input.tt}                       {keyboard handling routines}
begin
   if DefRecSize > MaxDataRecSize then
      begin
         writeln('TT:  DefRecSize > MaxDataRecSize');
         halt;
      end;
   CursorOn := GetCursorMode;
   EditCopyright;
   InitIndex;
   EditInitialize;
   Xposition :=1;
   Yposition :=1;
   InitMainMenu;
   ExitPullDown;
   EditSystem;
   ClrScr;
end.
{.PL6}
```

3. GLOBAL VARIABLES

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     vars.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                      under Contract F30602-C-85-0190

Function:  Declarations of all type definitions, global variables and
           constants including global structured typed constants.
*****************************************************************************
}

  type

    String80   = string [80];              {80 character string}
    Textline   = string [255];             {NOT A STRING}
    Varstring  = String80;                 {string for parameter passing}

  const

    Nofile     = 'TEMP';              { Default filename printed in window }
    DefRecSize = 127;                 { Default record size for temporary file}
    DefErrorFile = 'EDITERR.MSG';     { Default error message filename     }
    DefExt = '.def';                  { Extension for definition files }
    DesExt = '.des';                  { Extension for descriptor files }
    TxtExt = '.txt';                  { Extension for text files }
    IxExt = '.ix ';                   { Extension for index files }
    Line1      : Byte = 1;            { Logical line number 1 }
    Col1       : Byte = 1;            { Logical column number 1 }
    Lex1       = 1;                   { Lexical level 1 }
    Defnorows : Byte = 25;            { Default number of rows/physical screen }
    Defnocols = 80;                   { Default number of cols/physical screen }

    {These constants define the maximum database size}

    MaxCols = 16;
    MaxStreams = 8;
    MaxIndexes = 4;
```

```
  {These are constants for the buffer manager}

  MaxBufSize = 200;
  DefBufSize = 25;

  {The constants for Turbo Access are defined in main}

  Deftypahd = 500;                { Default number of chars/typeahead buf }
  Notavailable = 255;             { Used to test for input available }
  EndHdr = '\';                   { End of header indicator for tables }

  { The following is the set of ASCII control characters }

  Nul  = 0; Ctrla= 1; Ctrlb= 2; Ctrlc= 3; Ctrld= 4; Ctrle= 5;
  Ctrlf= 6; Ctrlg= 7; Ctrlh= 8; Ctrli= 9; Ctrlj=10; Ctrlk=11;
  Ctrll=12; Ctrlm=13; Ctrln=14; Ctrlo=15; Ctrlp=16; Ctrlq=17;
  Ctrlr=18; Ctrls=19; Ctrlt=20; Ctrlu=21; Ctrlv=22; Ctrlw=23;
  Ctrlx=24; Ctrly=25; Ctrlz=26; Escape=27; Del=127;

  { The following are the ASCII digits 0..9 }

  Zero = 48; One = 49; Two = 50; Three = 51; Four = 52;
  Five = 53; Six = 54; Seven = 55; Eight = 56; Nine = 57;

  { The following are flags to be used in the Linedesc record }

  Colored = $04;                  { Set: display line in Usercolor }
  Dchg    = $10;                  { Set: descriptor was changed    }
  Dins    = $20;                  { Set: descriptor was inserted   }
  Ddel    = $40;                  { Set: descriptor was deleted    }
  Tchg    = $80;                  { Set: text buffer is dirty      }
  DTchg   = $90;                  { Mask for both Dchg and Tchg    }
  Dmask   = $F0;                  { Mask for dirty bits            }
  Umask   = $0F;                  { Mask for user bits             }

  Screenadr : integer = $B800;    {Address of screen}

  { Search Character Categories }

  Alphas      : set of char = ['A'..'Z','a'..'z','0'..'9',''''];
  Separators  : set of char = [' '..'/', ':'..'@','['..'`', '{'..'}'];
  Digits      : set of char = ['0'..'9'];

type

  RegType = record
    case integer of
      1 : (AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : integer);
```

```
    2 : (AL, AH, BL, BH, CL, CH, DL, DH            : byte);
end;


Character  = record
               Ch : char;
               Color : byte
             end;

Plinedesc  = ^ Linedesc;              {real address of line descriptor}
Ptextline  = ^ Textline;              {real address of line contents}
Pwindesc   = ^ Windesc;

DataRef    = integer;                 {file address format}
FName = VarString;                    {file name format}

BatCtrlRec = record
               FromStrm : integer;    {stream when begin batch called}
               FromRef  : dataref;    {line when begin batch called}
               ToStrm   : integer;    {stream when end batch called}
               ToRef    : dataref;    {line when end batch called}
             end;

StreamDesc = record
               InUse    : boolean;    {true if stream is in use}
               DefRef   : dataref;    {address of this record in .def}
               UsrName  : FName;      {base name of this stream's files}
               NewName  : FName;      {name to call them when we close}
               RecLen   : integer;    {record size of text}
               TofRef   : dataref;    {address of first descriptor rec}
               DefHndl  : DataFile;   {handle of stream definition file}
               DesHndl  : DataFile;   {handle of line descriptor file}
               TxtHndl  : DataFile;   {handle of text file}
               CurIx    : 1..MaxIndexes;
               IxMap : array[1..MaxIndexes] of
                  record
                     Ixfn : FName;     {FileName for this index}
                     Ixcn : VarString; {Column name being indexed}
                     Opnd : boolean;   {set if this index opened}
                     Ixh  : IndexFile; {handle of this index file}
                  end;
               InTransaction : boolean;{set: between BT ET brackets}
             end;


SlotCtrlRec = record
                Dref : dataref;   {file address of descriptor}
                Dptr : Plinedesc; {memory address of descriptor}
```

```
                Lru  : integer;   {count for victim selection}
                Tref : dataref;   {file address of text line}
                Tptr : Ptextline; {memory address of text line}
                Tbak : Ptextline; {memory address of before image}
              end;

SlotCtrlAry = array[1..MaxBufSize] of SlotCtrlRec;

Pslot = ^SlotCtrlAry;

BufCtrlRec = record
               BufSize  : 1..MaxBufSize; {buffer capacity in lines}
               BufCnt   : 0..MaxBufSize; {buffer contents in lines}
               Slot     : Pslot;
             end;

Linedesc   = record
               FwdRef   : DataRef;     {Ptr to previous line in file}
               BakRef   : DataRef;     {Ptr to next line in file}
               TxtRef   : DataRef;     {Ptr to text record in file}
               TxtOfs   : integer;     {Offset to start of text in file}
               Flags    : integer;     {Define these in the const section}
               BuffLen  : integer;     {Length of text record}
               Txt      : Ptextline;   {pointer to text in memory}
             end;

Insflag = (Insert, Typeover);          {Used only in window records}

AccessType  = (rd,                     {read descriptor}
               rt,                     {read descriptor and text}
               wd,                     {write descriptor}
               wt);                    {write descriptor and text}

RepLastType = (Find,
               Replace,
               Keyword,
               Select,
               Project,
               Join,
               Sort,
               Unique,
               Closure,
               Nothing);

               {used by SearchNext to decide what kind of search to repeat}

Windesc     = record
```

```
{Links to other windows}

Fwdlink     : Pwindesc;       {Next window down}
Backlink    : Pwindesc;       {Next window up}

{Files attached to this window}

Filename    : FName;          {Name of file being edited}
Stream      : integer;        {unique text stream identifier}
Formatted   : boolean;        {Set if formatted file}
Header      : Ptextline;      {Column headers if formatted}
HdrLen      : integer;        {Size of header in bytes}

{Display attached to this window}

Firstlineno : integer;        {Physical top line number}
Lastlineno  : integer;        {Physical bottom line no}
Lmargin     : integer;        {Left margin column number}
Rmargin     : integer;        {Right margin column number}
Lineno      : integer;        {Last cursor address}
Colno       : integer;        {Last cursor address}
Clineno     : integer;        {Display only: line in file}
Topline     : DataRef;        {Ptr to first line in view}
Curline     : DataRef;        {Ptr to line cursor is On}
Leftedge    : integer;        {Leftmost displayed column}

{Transaction parameters for this window}

BTcurline   : DataRef;        {Curline at begin transaction}
BTtopline   : DataRef;        {Topline at begin transaction}
BTtofref    : DataRef;        {TofRef at begin transaction}
NTstream    : integer;        {Nested transaction stream id}

{Search parameters for this window}

SearchStr   : Varstring;      {parameters to find, replace}
ReplaceStr  : Varstring;      {parameters to replace}
OptStr      : Varstring;      {options for all searches}
QualStr     : Varstring;      {parameters to rel commands}
DestinStr   : Varstring;      {chr window # for search output}
AllOccur    : Boolean;        {true if searching all occurrences}
IgnCase     : Boolean;        {true if ignoring case in search}
WholeWord   : Boolean;        {true if match must be whole word}
SearchBack  : Boolean;        {true if searching backwards}
Global      : Boolean;        {true if finding first/last occur}
NoAsk       : Boolean;        {true if replacing w/o confirm}
FirsTime    : Boolean;        {true if search just initiated}
OutPut      : Boolean;        {true if writing output to window}
```

```
                SetScan     : Boolean;      {true if SetScanning lines}
                Deswin      : Pwindesc;     {ptr to output window}
                Deslin      : integer;      {length of source window hdr}
                Despass     : integer;      {number of search passes}
                MTpass      : boolean;      {true if no rows qualified}
                NextPass    : boolean;      {true if source to be advanced}
                RepLast     : RepLastType;  {type of search to be repeated}

                {Miscellaneous data for this window}

                WindNo    : integer;        {number of this window}
                Insertflag  : Insflag;      {Insert mode flag}
                AI          : boolean;      {Autoindent mode}

              end;

var

  {buffer attributes}

  StreamDef : array[1..MaxStreams] of Streamdesc;
  BufMap    : array[1..MaxStreams] of BufCtrlRec;
  MTStrDef  : StreamDesc;

  {physical screen attributes}

  Physcrrows : integer;    { No. lines/physical screen }
  Physcrcols : integer;    { No. cols/physical line }
  Logscrrows : integer;    { No. lines/logical screen }
  Logscrcols : integer;    { No. cols/logical line }
  Logtopscr  : integer;    { Physical line no. for logical line #1 }
  Physcrsig  : integer;    { Physical line for signals, etc. }
  Retracemode: boolean;    { Set if wait for vertical retrace is needed }
  Linelength : integer;    { Number of chars/textline }
  Cmdcol     : integer;    { Column for next cmd printed on signal }

  {color attributes}

  Bordcolor  : integer;    { Border color from init routine }
  Txtcolor   : integer;    { Text color from init routine }
  Cmdcolor   : integer;    { Command line color from init routine}
  Usercolor  : integer;    { Special display color -- user programmable }
  CmdOnColor : integer;    { Cmdcolor set by EditAppCmd }
  CmdOffColor: integer;    { Cmdcolor set by EditZapCmd }
  VertColor  : integer;    { Column divider color set by init }

  {window pointers}
```

```
    Curwin      : Pwindesc;    { Pointer to window containing cursor }
    Window1     : Pwindesc;    { Pointer to window at top of screen  }
    Winstack    : Pwindesc;    { Ptr to free list of windows }

    {task scheduling parameters}

    Rundown     : boolean;     { Determines when scheduler aborts }
    Notfound    : boolean;     { Set by search and replace cmds }
    Abortcmd    : boolean;     { Set by EditAbort to stop EditAskfor }
    Aborting    : boolean;     { Set by EditAbort to stop recursion }
    Typbufovl   : boolean;     { Used by EditPushtbf only!}
    Nextstream  : integer;     { Next stream id to be assigned }
    Intrflag    : (Nointrpt,Intrpt);       {Scheduling}
    Updcurflag  : boolean;     { Set if we must reposition cursor }
    Asking      : boolean;     { Set if EditAskFor procedure has control }
    Interactive: boolean;      { Set if EditAskFor should read via EditPolcon }
    EditChangeFlag : boolean;{Indicates changes in text}

    {display areas}

    Screen      : array [1..25, 1..80] of Character;
    Cmdlinest   : Textline;    { Command line image used by EditUpdphyscr }
    Tabsize     : integer;     { Distance between tab stops }

    {keyboard input areas}

    Circbuf     : array [0..Deftypahd] of char;
    Circin      : integer;     {Pointer to put data into the c.b.}
    Circout     : integer;     {Pointer to take data out of c.b.}
    EditUsercommandInput : integer; {count of chars pushed by UserCommand}

    {batch control area}

    BatRec : BatCtrlRec;

const

{ The following strings are the headlines of the pull-down menus. Insert     }
{ only spaces before the section. If you need more than one space after the  }
{ the whole line, change the value of "NoSpaces". The length must match the  }
{ the actual length of the string.                                            }

  st01 : string[05] = ' Scan';
  st02 : string[09] = '    Query';
  st03 : string[10] = '    Jspace';
  st04 : string[12] = '    Transact';
  st05 : string[10] = '    Window';
  st06 : string[08] = '    File';
```

```
  st07 : string[12] = '    Relation';
  st08 : string[11] = '    Engine ';
  st09 : string[01] = '';     { The headline and each submenu are terminated }
                              { by an empty String with length of "1"        }

{ The following strings are the labels for Pull-Down Menu #1- the Update Com-}
{ mands. The first entry must be padded with spaces because the length of    }
{ this string is used to represent the maximum length of a submenu's label.  }

  st11 : string[12] = 'Include     ';
  st12 : string[07] = 'eXclude';
  st13 : string[05] = 'Begin';
  st14 : string[03] = 'End';
  st15 : string[05] = 'cLear';
  st16 : string[04] = 'Read';
  st17 : string[04] = 'Next';
  st18 : string[05] = 'Prior';
  st19 : string[06] = 'Delete';
  st1A : string[04] = 'Move';
  st1B : string[04] = 'Copy';
  st1C : string[05] = 'Write';
  st1x : string[01] = '';

{ Pull-Down Menu #2 - Query Commands                                         }

  st21 : string[13] = 'Find         ';
  st22 : string[07] = 'Replace';
  st23 : string[07] = 'Keyword';
  st24 : string[06] = 'Select';
  st25 : string[07] = 'Project';
  st26 : string[04] = 'Join';
  st27 : string[05] = 'Order';
  st28 : string[04] = 'Next';
  st29 : string[05] = 'cLear';
  st2A : string[05] = 'Union';
  st2B : string[10] = 'Difference';
  st2x : string[01] = '';

{ Pull-Down Menu #3 - Index Commands                                         }

  st31 : string[12] = 'Create      ';
  st32 : string[06] = 'Remove';
  st33 : string[04] = 'Open';
  st34 : string[05] = 'cLose';
  st35 : string[05] = 'Synch';
  st36 : string[14] = 'synch All';
  st3x : string[01] = '';
```

```
{ Pull-Down Menu #4 - Transaction Commands                              }

  st41 : string[13] = 'Begin        ';
  st42 : string[06] = 'Commit';
  st43 : string[05] = 'Abort';
  st44 : string[05] = 'Enter';
  st45 : string[05] = 'Leave';
  st46 : string[05] = 'Share';
  st47 : string[09] = 'eXclusive';
  st48 : string[04] = 'scaN';
  st4x : string[01] = '';

{ Pull-Down Menu #5 - Window Commands                                   }

  st51 : string[10] = 'Select    ';
  st52 : string[04] = 'Open';
  st53 : string[05] = 'Close';
  st54 : string[04] = 'Link';
  st55 : string[05] = 'cleaR';
  st5x : string[01] = '';

{ Pull-Down Menu #6 - File Commands                                     }

  st61 : string[11] = 'Open       ';
  st62 : string[05] = 'Close';
  st63 : string[04] = 'Read';
  st64 : string[05] = 'Write';
  st65 : string[04] = 'Quit';
  st66 : string[03] = 'Top';
  st67 : string[06] = 'Bottom';
  st68 : string[05] = 'Synch';
  st6x : string[01] = '';

{ Pull-Down Menu #7 - Relation Commands                                 }

  st71 : string[13] = 'Create       ';
  st72 : string[06] = 'sWitch';
  st73 : string[05] = 'aLign';
  st74 : string[04] = 'Sort';
  st75 : string[06] = 'Add to';
  st76 : string[09] = 'Drop from';
  st77 : string[06] = 'cHange';
  st78 : string[06] = 'Unique';
  st7x : string[01] = '';

{ Pull-Down Menu #8 - Engine Commands                                   }

  st81 : string[13] = 'Closure      ';
```

```
st82 : string[05] = 'Match';
st83 : string[10] = 'Part-whole';
st84 : string[08] = 'Deviance';
st85 : string[08] = 'Temporal';
st86 : string[08] = 'Resource';
st87 : string[05] = 'Begin';
st88 : string[03] = 'End';
st89 : string[10] = 'checKpoint';
st8A : string[07] = 'reStart';
st8x : string[01] = '';


UpperLeft  = 201; { '┌' }
UpperRight = 185; { '╣' }
LowLeft    = 200; { '└' }
LowRight   = 188; { '┘' }
Horizontal = 205; { '═' }
Vertical   = 186; { '│' }
TConnect   = 203; { '╦' }
RConnect   = 206; { '╬' }
LConnect   = 204; { '╠' }
LRConnect  = 202; { '╩' }

WUpperleft = 201; { '┌' }
WUpperright= 187; { '┐' }
WLowLeft   = 200; { '└' }
WLowRight  = 188; { '┘' }
WHorizontal= 205; { '═' }
WVertical  = 186; { '│' }

CursorLeft  = ^S;
CursorRight = ^D;
CursorDown  = ^X;
CursorUp    = ^E;
PageUp      = ^R;
PageDown    = ^C;

NoSpaces   =  1;                  { Trailing spaces after the headline    }

MaxSubmenu   = 10;                { Maximum no. of submenus               }
MaxSelection = 14;                { Max no. of selections in a submenu    }
MaxSave      = 17;                { Max no. of lines to save under menu   }
MaxWide      = 30;                { Maximum width of selection            }

CursorOff = $2000;

MaxWindowX   = 80;                { Maximum columns in window             }
MaxWindowY   = 20;                { Maximum lines in window               }
```

```
Var

{ Colors for the pulldown menus                                          }
  PNormColor  : integer;              { Color of non-selected parts      }
  PLowColor   : integer;              { Color of selected parts          }
  PFrameColor : integer;              { Color of frame                   }

{ Colors for the windows                                                 }
  WNormColor  : integer;              { Color of non-selected parts      }
  WLowColor   : integer;              { Color of selected parts          }
  WFrameColor : integer;              { Color of frame                   }

  NoSubmenus   :Byte;                   { Number of submenus (max. 10)   }
  MenuData     :Array[1..MaxSubmenu] of { Definition of main pull-down menu }
                Record
                  MenuChar :Char;       { Significant Character          }
                  Position :Byte;       { Horizontal position on screen  }
                  MenuAddr :Integer;    { Offset for Address             }
                  SubStart :Integer;    { Start of submenu               }
                End;

  NoSelection  :Byte;                     { Number of entrys in submenu  }
  SelectionData:Array[1..MaxSelection] of { Definition of submenu         }
                Record
                  SubChar  :Char;       { Significant character          }
                  SubAddr  :Integer;    { Offset for Address             }
                End;
  SaveSubmenu  :array [0..MaxSave] of     { Data structure to save the   }
                array [0..MaxWide] of     { old screen under a submenu   }
                Integer;

  MenuStatus   :Byte;                   { Status of selection            }
  CurrSubmenu  :Byte;                   { Current submenu                }
  CurrSelection:Byte;                   { Current selection within submenu }
  CurrAddress   :Integer;               { Current Address in definition  }
  CurrPosition :Byte;                   { Current horizontal position    }
                                        { on screen                      }
  EndPosition  :Byte;                   { Last character of main line    }
  SubMenuPos   :Byte;                   { Horizontal start of submenu    }
  SubLength    :Byte;                   { Wide of pulldown menu          }
  MenuColor    :Integer;                { Current color                  }
  MemAdr       :Integer;                { Address in memory for next char. }
                                        { to display                     }
  Counter      :Integer;                { General purpose loop counters  }
  Counter1     :Integer;
  XPosition    :Integer;                { Basic Address of pulldown menu }
  YPosition    :Integer;
```

```
PdSaveCursor :Integer;                { Cursor mode on entry to pulldowns   }
CursorOn     :Integer;

          { Array to save the screen under a pop-up window }

WArray       : array[0..MaxwindowY,0..MaxWindowX] of integer;
```

4. BUFFER MANAGEMENT

```
{
*******************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     bufman.tt                   Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                       under Contract F30602-C-85-0190

Function:  All memory management activities including the creation and
           deletion of line descriptors and text lines, and the management
           of the data file buffers.
*******************************************************************************
}

procedure EditUpdIndex(Stream,i : integer);
{  This procedure is called to update all the index files associated with a
   record.  If the record is flagged for deletion, the keys are extracted
   from the before image and deleted from the indexes.  If the record is
   flagged for insertion, the keys are extracted from the after image and
   added to the indexes.  If the record is flagged as a change, the before
   and after image of each key is compared and, if there is a difference,
   the before image is deleted from the index and the after image is added
   to the index.
}

begin
end;

procedure EditSync(Stream,i : integer);
begin
   if BufMap[Stream].Slot^[i].Dref = 0 then exit;
   with StreamDef[Stream],BufMap[Stream] do
      with Slot^[i],Slot^[i].Dptr^ do
         case (Flags and Dmask) of
            Dchg : begin
                      Flags := Flags and not(Dmask);
                      PutRec(DesHndl,Dref,Dptr^);
                   end;
            Tchg : begin
                      EditUpdIndex(Stream,i);
                      PutRec(TxtHndl,Tref,Tptr^);
```

```
                end;
            Dins,
           DTchg : begin
                      EditUpdIndex(Stream,i);
                      Flags := Flags and not(Dmask);
                      PutRec(DesHndl,Dref,Dptr^);
                      PutRec(TxtHndl,Tref,Tptr^);
                   end;
            Ddel : begin
                      EditUpdIndex(Stream,i);
                      DeleteRec(DesHndl,Dref);
                      DeleteRec(TxtHndl,Tref);
                   end;
            end;
end;

function Edit(Stream : integer;
              Dr     : dataref;
              Intent : AccessType) : Plinedesc;

{  This function returns a real memory address of the line descriptor in
   the specified descriptor stream.  If the descriptor is not in real
   memory, it is swapped in.  If the access type is r)ead t)ext or
   w)rite t)ext, the text for the line is also swapped in and its real
   address is stored in the line descriptor.  If the access type is
   w)rite d)descriptor or w)rite t)ext, the dirty data bits are set.
   The guaranteed viability of addresses returned by this function is
   limited to the number of real lines (BufSize) in the queue for
   replacement.  It is the callers responsibility to refresh addresses
   that will be saved accross calls.
}

var i,j        : integer;
    SlotAvail  : integer;

   function Victim(Stream : integer) : integer;

   var Age,i : integer;

   begin
      Victim := BufMap[Stream].BufSize;
      Age := 0;
      with StreamDef[Stream],BufMap[Stream] do
         for i := 1 to BufSize do
            with Slot^[i],Slot^[i].Dptr^ do
               if (not InTransaction) or
                  ((InTransaction) and (Flags and Dmask = 0)) then
                     if Lru > Age then
```

```
                        begin
                           Age := Lru;
                           Victim := i;
                        end;
   end;

   function Expand(Stream : integer) : integer;

   var Size : integer;

      function NoMem(Size : integer) : boolean;

      begin
         NoMem := (MaxAvail > 0) and
                  (MaxAvail <=
                  ((2*succ(Size) + 32 + SizeOf(LineDesc) + 4096) div 16));
      end;


   begin
      Expand := 0;
      Size := StreamDef[Stream].RecLen;
      with BufMap[Stream] do
         with Slot^[succ(BufSize)] do
            if (succ(BufSize) < MaxBufSize) and (not NoMem(Size)) then
               begin
                  GetMem(Dptr,SizeOf(LineDesc));
                  GetMem(Tptr,succ(Size));
                  GetMem(Tbak,succ(Size));
                  Dref := 0;
                  Tref := 0;
                  Lru := 0;
                  Expand := succ(BufSize);
               end;
   end;

begin
   Edit := nil;
   if Dr = 0 then exit;
   with StreamDef[Stream],BufMap[Stream] do
      begin
         {look for the requested dataref in the buffer}
         {and, while we're at it, we'll increment the}
         {LRU counts for any slots that we look at and}
         {generate a count of the slots with clean data}
         i := 1;
         SlotAvail := 0;
         while (i <= BufSize) and (Dr <> Slot^[i].Dref) do
```

```
   with Slot^[i], Slot^[i].Dptr^ do
      begin
         Lru := succ(Lru);
         i := succ(i);
         if (Flags and Dmask) = 0 then SlotAvail := succ(SlotAvail);
      end;
if i > BufSize then
   begin

      {the requested dataref is not in the buffer}
      {so we have to swap it in from the disk}

      if BufCnt >= BufSize then
         begin

            {the buffer is full so we have to select}
            {a victim to swap out to the disk first}

            if InTransaction then
               begin

                  {we're in a transaction so we can't swap out}
                  {any records that have dirty bits set but we}
                  {want to keep at least enough clean slots to}
                  {satisfy screen update requirements}

                  while (SlotAvail < DefBufSize) do
                     begin
                        i := Expand(Stream);
                        if i <> 0 then
                           begin
                              BufSize := succ(BufSize);
                              SlotAvail := succ(SlotAvail);
                           end
                        else
                           begin
                              SlotAvail := DefBufSize; {break}
                           end;
                     end;
               end;

            i := Victim(Stream);
            if i = 0 then
               begin

                  {the WAL is full so the best we can do is to}
                  {clear the transaction flag so we can get the}
                  {memory to continue -- Sorry!!}
```

```
                  InTransaction := false;
                  i := Victim(Stream);
                  EditErrorMsg(84);
               end;

            {we found a legal victim so we need}
            {to sync him to the file system}

            EditSync(Stream,i);
            BufCnt := pred(BufCnt);
            Slot^[i].Dref := 0;
            Slot^[i].Tref := 0;
         end
      else
         begin

            {there's at least one empty slot in the}
            {buffer so let's find the first one and}
            {use it for the new record}

            i := 1;
            while Slot^[i].Dref <> 0 do i := succ(i);

         end;
      GetRec(DesHndl,Dr,Slot^[i].Dptr^);
      if not OK then exit;
      Slot^[i].Dref := Dr;
      Slot^[i].Tref := 0;
      BufCnt := succ(BufCnt);
   end
else
   begin

      {the requested record is already in memory and i points to it}
      {so just up the rest of the LRU counts}

      j := succ(i);
      while j <= BufSize do
         begin
            Slot^[j].Lru := succ(Slot^[j].Lru);
            j := succ(j);
         end;
   end;
with Slot^[i] do
   begin
      Lru := 0;
      if ((Intent = rt) or (Intent = wt)) and (Tref = 0) then
```

```
         begin

            {the caller has told us that he intends to}
            {do something with the text for this record}
            {so we need to bring it in from disk too}

            GetRec(TxtHndl,Dptr^.TxtRef,Tptr^);
            if not OK then
               begin
                  Dref := 0;
                  exit;
               end;
            Tref := Dptr^.TxtRef;
            Move(Tptr^,Tbak^,RecLen);
            Dptr^.Txt := Tptr;
         end;

      {set the proper dirty bits}

      with Dptr^ do
         case Intent of
            rd : ;
            rt : ;
            wd : if Flags and Dins = 0 then Flags := Flags or Dchg;
            wt : if Flags and Dins = 0 then Flags := Flags or Tchg;
         end;

      {return the Plinedesc}

      Edit := Dptr;
   end;

if (not InTransaction) and (BufSize > DefBufSize) then

   {a prior transaction has expanded the buffer}
   {beyond the default buffer size so we need to}
   {shrink it back by sync'ing all the slots above}
   {DefBufSize and releasing them to OS for reuse}

   while BufSize > DefBufSize do
      with Slot^[BufSize], Slot^[BufSize].Dptr^ do
         begin
            if Dref <> 0 then BufCnt := pred(BufCnt);
            if (Flags and Dmask) <> 0 then EditSync(Stream,BufSize);
            FreeMem(Dptr,SizeOf(Linedesc));
            FreeMem(Tptr,succ(RecLen));
            FreeMem(Tbak,succ(RecLen));
            BufSize := pred(BufSize);
```

```
                end;
      end;
end;

procedure EditDestxtdes  (Stream : integer; Desc : dataref) ;
{  This routine disposes of a line descriptor and its associated
   text.
}

var  pt : Plinedesc;

begin {EditDestxtdes}
   with StreamDef[Stream] do
      begin
         pt := Edit(Stream,Desc,wd);
         pt^.Flags := Ddel;
      end;
end; {EditDestxtdes}

function EditMaktxtdes  (Stream : integer; Ncols : integer) : dataref;
{  This routine allocates a line descriptor and a text line from Ta
   and links them together for the caller.  If the stream buffer is full, a
   victim is swapped out to make room for the new records.  The contents of
   the text line are set to spaces, and the length of the buffer is stored
   in the descriptor.  The file address of the new descriptor is returned
   if the process was successful otherwise it returns 0.
}
var errcd : integer;
    dref  : dataref;
    tref  : dataref;
    p     : Plinedesc;
    t     : Ptextline;

begin {EditMaktxtdes}
   EditMakTxtDes := 0;
   with StreamDef[Stream] do
      begin
         if pred( ((Ncols + 16) div 16) * 16 ) > RecLen then exit;
         GetMem(t,succ(RecLen));
         FillChar(t^[1],RecLen,' ');
         AddRec(TxtHndl,tref,t^);
         FreeMem(t,succ(RecLen));
         GetMem(p,SizeOf(LineDesc));
         with p^ do
            begin
               FwdRef := 0;
               BakRef := 0;
               TxtRef := tref;
```

```
                TxtOfs := 0;
                Flags := Dins;
                BuffLen := pred( ((Ncols + 16) div 16) * 16 );
                Txt := nil;
             end;
          AddRec(DesHndl,dref,p^);
          FreeMem(p,SizeOf(LineDesc));
       end;
   p := Edit(Stream,dref,wt); {This causes dref to be swapped in}
   EditMaktxtdes := dref;
end; {EditMaktxtdes}

function EditSizeline (Stream : integer;
                       p      : dataref;
                       Ncols  : integer) : boolean ;

{  This routine examines the length of the allocated line buffer for
   the text descriptor pointed to by p and adjusts the buffer length.
}

var q : Plinedesc;

begin {EditSizeline}
  EditSizeLine := false;
  q := Edit(Stream,p,wd);
  with q^ do
     begin
        if Ncols <= StreamDef[Stream].RecLen then
           begin
              if Ncols > Bufflen then
                 begin
                    BuffLen := pred( ((Ncols + 16) div 16) * 16 );
                    if Bufflen < StreamDef[Stream].RecLen then
                       begin
                          EditSizeLine := true;
                       end
                    else
                       begin
                          Bufflen := StreamDef[Stream].RecLen;
                          EditSizeLine := true;
                       end;
                 end
              else
                 begin
                    EditSizeLine := true;
                 end;
           end;
     end;
```

```
end; {EditSizeline}

function EditOpenFile(Stream : integer; Fn : FName; Wid : integer) : boolean;
{  This funciton attaches the file specified by Fn to the specified Stream.
   If the process succeeds the function returns true.
}

var SvDef  : DataFile;

begin
   EditOpenFile := false;
   with StreamDef[Stream] do
      begin
         if Fn = Nofile then
            begin
               Fn := Fn + chr(ord('0') + Stream);
               Move(MTStrDef,StreamDef[Stream],SizeOf(StreamDesc));
               MakeFile(DefHndl,Fn + DefExt,SizeOf(StreamDesc));
               if not OK then exit;
               RecLen := Wid;
               UsrName := Fn;
               NewName := '';
               AddRec(DefHndl,DefRef,StreamDef[Stream]);
               CloseFile(DefHndl);
               MakeFile(DesHndl,Fn + DesExt,SizeOf(LineDesc));
               if not OK then exit;
               MakeFile(TxtHndl,Fn + TxtExt,succ(RecLen));
               if not OK then
                  begin
                     CloseFile(DefHndl);
                     exit;
                  end;
            end
         else
            begin
               OpenFile(DefHndl,Fn + DefExt,SizeOf(StreamDesc));
               if not OK then exit;
               move(DefHndl,SvDef,SizeOf(DataFile));
               GetRec(DefHndl,DefRef,StreamDef[Stream]);
               if not OK then exit;
               move(SvDef,DefHndl,SizeOf(DataFile));
               UsrName := Fn;
               NewName := '';
               CloseFile(DefHndl);
               OpenFile(DesHndl,Fn + DesExt,SizeOf(LineDesc));
               if not OK then exit;
               OpenFile(TxtHndl,Fn + TxtExt,succ(RecLen));
               if not OK then
```

```
                    begin
                       CloseFile(DesHndl);
                       exit;
                    end;
              end;
        end;
    EditOpenFile := true;
end;

procedure EditCloseFile(Stream : integer);
{  This procedure syncs and closes the file attached to Stream.
}

var i : integer;
    f : file;

begin
    with StreamDef[Stream],BufMap[Stream] do
       begin
          for i := 1 to BufSize do
             if Slot^[i].Dref <> 0 then EditSync(Stream,i);
          CloseFile(DesHndl);
          CloseFile(TxtHndl);
          if CurIx <> 0 then CloseIndex(IxMap[CurIx].Ixh);
          OpenFile(DefHndl,UsrName + DefExt,SizeOf(StreamDesc));
          If not OK then exit;
          PutRec(DefHndl,DefRef,StreamDef[Stream]);
          CloseFile(DefHndl);
          if NewName <> '' then
             begin
                assign(f,UsrName + DefExt);
                rename(f,NewName + DefExt);
                assign(f,UsrName + DesExt);
                rename(f,NewName + DesExt);
                assign(f,UsrName + TxtExt);
                rename(f,NewName + TxtExt);
             end;
       end;
end;

function EditNewstream(Fn : FName; Wid : integer)  : integer ;
{  This function initializes a stream buffer for a file of records whose
   name is Fn.  If the buffer is successfully created, the Stream ID
   is returned otherwise the function returns 0.
}

   function NoMem(Size : integer) : boolean;
```

```
   begin
      NoMem := (MaxAvail > 0) and
               (MaxAvail <=
               ((2*succ(Size) + 32 + SizeOf(LineDesc) + 4096) div 16));
   end;

var i    : integer;
    Size : integer;

begin {EditNewstream}
   EditNewStream := 0;
   i := 1;
   while (StreamDef[i].InUse) and (i <= MaxStreams) do i := succ(i);
   if (i > MaxStreams) then exit;
   StreamDef[i].DefRef := 1;
   if not EditOpenFile(i,Fn,Wid) then exit;
   with StreamDef[i] do
      begin
         if NoMem(RecLen) then exit;
         InUse := true;
         CurIx := 0;
         Size := RecLen;
      end;
   with BufMap[i] do
      begin
         BufSize := 1;
         BufCnt := 0;
         repeat
            with Slot^[BufSize] do
               begin
                  GetMem(Dptr,SizeOf(LineDesc));
                  GetMem(Tptr,succ(Size));
                  GetMem(Tbak,succ(Size));
                  Dref := 0;
                  Tref := 0;
                  Lru := 0;
                  BufSize := succ(BufSize);
               end;
         until (NoMem(Size)) or (BufSize > DefBufSize);
      end;
   BufMap[i].BufSize := pred(BufMap[i].BufSize);
   EditNewStream := i;
end; {EditNewstream}

procedure EditDelStream(Stream : integer);
{  This procedure closes the file attached to Stream and releases the buffer
   and stream definition for reuse.
}
```

```
begin
   if Stream = 0 then exit;
   if not StreamDef[Stream].InUse then exit;
   EditCloseFile(Stream);
   StreamDef[Stream].InUse := false;
   with BufMap[Stream],StreamDef[Stream] do
      repeat
         with Slot^[BufSize] do
            begin
               FreeMem(Dptr,SizeOf(Linedesc));
               FreeMem(Tptr,succ(RecLen));
               FreeMem(Tbak,succ(RecLen));
               BufSize := pred(BufSize);
            end;
      until BufSize = 0;
end;

function EditRenStream(Stream : integer; Fn : FName) : boolean;
{  This function renames Stream as Fn.  If Fn already exists or for some
   other reason the stream can't be renamed to Fn, the function returns
   false, otherwise it returns true.  If Fn is null, the operation is
   idempotent and returns true.
}

var f : file;

   function Exist(Fn : FName) : boolean;

   var f : file;

   begin
      Assign(f,Fn);
      Reset(f);
      Exist := (IOResult = 0);
      Close(f);
   end;

begin
   EditRenStream := true;
   if Fn = '' then exit;
   with StreamDef[Stream] do
      begin
         EditRenStream := false;
         if Exist(Fn + DefExt) then exit;
         if Exist(Fn + DesExt) then exit;
         if Exist(Fn + TxtExt) then exit;
         NewName := Fn;
```

```
        EditRenStream := true
      end;
end;
```

5. RESIDENT STREAM MANAGEMENT

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     user.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  Low level stream update functions called by overlaid command
           procedures.
*****************************************************************************
}


{  The following are forward declarations to routines
   which are called from routines in this module or in the command
   modules.  This allows more flexibility in positioning toolbox
   INCLUDE files in your programs.
}
function  EditEdistat : boolean; forward;
function  EditKeypressed : boolean; forward;
function  EditGetinput : byte; forward;
procedure EditBreathe; forward;
procedure EditBackground; forward;
procedure EditUpdphyscr; forward;
procedure EditHscroll; forward;
procedure EditUpdrowasm (row : byte); forward;
function  EditWrline (var s:textline;row:byte;txtcolor:byte):boolean; forward;
procedure EditUpdwinsl (w : Pwindesc); forward;

procedure EditIncline;
{  This routine increments the current window's line number
   and then displays it on the status line.
}
begin {EditIncline}
  Curwin^.Clineno := Succ (Curwin^.Clineno);
  EditUpdwinsl (Curwin)
end; {EditIncline}

procedure EditDecline;
```

```
{  This routine decrements the current window's line number
   and then displays it on the status line.
}
begin {EditDecline}
  Curwin^.Clineno := Pred (Curwin^.Clineno);
  EditUpdwinsl (Curwin)
end; {EditDecline}

procedure EditAppcmdnam  (s : Varstring) ;
{  This routine appends a string to the command line for later display. }
var
  i : integer;

begin {EditAppcmdnam}
  if Asking then
     CmdColor := CmdOnColor
  else
     CmdColor := CmdOffColor;
  if Cmdcol = 1 then
     s := chr(Vertical) + chr(Curwin^.Windno + ord('0')) +
          chr(Vertical) + ' ' + s;
  if s <> #0 then
    for i := 1 to Byte (s[0]) do
      begin
         if pred(Cmdcol + i) > PhyScrCols then
            begin
               delete(Cmdlinest,1,1);
               Cmdlinest := Cmdlinest + ' ';
               Cmdcol := pred(Cmdcol);
            end;
         Cmdlinest [Pred (Cmdcol+i)] := s [i];
      end;
  Cmdcol := Cmdcol + Byte (s[0]);
  if Asking then GotoXY(succ(Cmdcol),PhyScrSig);
end; {EditAppcmdnam}

procedure EditZapcmdnam;
{  This routine sets the command line string to spaces. }
var  i : integer;

begin {EditZapcmdnam}
  Cmdcolor := CmdOffColor;
  Cmdlinest [0] := Chr(Physcrcols);
  Fillchar (Cmdlinest[1], Physcrcols, ' ');
  Cmdcol := 1;                              {Not black, still blanks out}
end; {EditZapcmdnam}

function EditMessage (Mno : byte) : Varstring;
```

```
{  This routine trys to read a error message from the
   file. If not possible, it produces the error no. and
   an additional message instead. The string is returned
   to the caller
}

var
  St  : varstring;
  fl  : text;
  i,c : Integer;

begin {EditMessage}
  Assign(fl,DefErrorFile);          {The error message is read from the file}
  Reset(fl);
  c := IOresult;
  if c = 0 then
  begin
    Repeat
      ReadLn(fl,St);
      Val(Copy(St,1,3),i,c);        {Check the error no.}
    until (Mno = i) or eof(fl);
    If eof(fl) then c := 255;
    Delete(St,1,3);
    Close(fl);
  end;
  If c <> 0 then
  begin                             {If something went wrong then display}
    Str(Mno,St);                    {the plain error number}
    St := '  Error ' + St + ' - Message file not found';
  end;
  EditMessage := St;
end; {EditMessage}

procedure EditErrormsg ((Msgno : byte));
{  This routine is called by any command module to append an error
   message to the command line, dump the typeahead buffer to a bit
   bucket, then wait for the user to press any key to continue. The
   routine tries to load the message from a file called 'EDITERR.MSG'.
   if it doesn't find the file, then the plain error no. is displayed.
   If you code a UserError routine to trap the error code, you can set
   UserError's argument to any byte value, enabling you to change the
   message to be displayed or to disable display of messages.
}
var
  Mno : byte;
  Ch  : byte;

begin {EditErrormsg}
```

```
  Circin := Circout;                    {Zap typeahead buffer before read}
  EditUsercommandInput := 0;            {Reset char count for EditClsinp}
  Mno := Msgno;                         {Local copy so user can change its value}
  UserError (Mno);                      {Give user first crack at the error}
  If Mno <> 0 then                      {If he sets it to 0, then don't issue msg}
  begin
    EditAppcmdnam(EditMessage(Mno) + ' - Press <Esc>');
    EditUpdphyscr;
    repeat
      while not EditKeypressed do;
      Ch := EditGetInput;
    until ch = 27;
  end;
  Circin := Circout;                    {Zap typeahead buffer again}
  Updcurflag := true;
end; {EditErrormsg}

function EditFileerror  : boolean  ;
{  This routine examines Ioresult set by the Turbo system from the last
   I/O, and if non-zero, prints a message by calling EditErrormsg with
   the appropriate code.
}
var
  Code : Byte;

begin {EditFileerror}
  Code := Ioresult;
  if Code <> 0 then EditErrormsg (Code);
  EditFileerror := Code <> 0
end; {EditFileerror}

procedure EditRealign;
{  This procedure is called anytime a text stream is modified by
   either adding lines to it or deleting lines from it. If several
   windows happen to be positioned over the same text area which
   is contracted or expanded, the line number, current line pointer,
   and Topline pointer can be made to be disharmoneous. This procedure
   assumes that Topline and Curline are well defined for each window;
   that is, if you delete text, you must make sure that you do not
   make these pointers point to reclaimed storage. If you delete
   lines in a procedure, call EditDelline; it checks this.
}
var
  p     : Pwindesc;
  q,r   : dataref;
  pt    : Plinedesc;
  SvTop : dataref;
```

```
  function TOF(SvTop : dataref; Win : Pwindesc) : dataref;

  var p,q : dataref;
      pt  : Plinedesc;

  begin
     p := SvTop;
     with Win^ do
        begin
           repeat
              q := p;
              pt := Edit(Stream,p,rd);
              p := pt^.BakRef;
           until p = 0;
           TOF := q;
        end;
  end;

begin {EditRealign}
  p := Curwin;
  repeat
    with p^ do
      begin                    {Realign this window}
        Lineno := 1;
        q := Topline;          {This should always be defined}
        SvTop := Topline;
        while q <> Curline do
          begin                {Scan until we find the current line}
            pt := Edit(Stream,q,rd);
            q := pt^.FwdRef;
            if q = 0 then
               begin
                  q := TOF(SvTop,p);
                  Topline := q;
                  Lineno := 1;
               end
            else
               begin
                  if Lineno = (Lastlineno - Firstlineno) then
                     begin
                        pt := Edit(Stream,Topline,rd);
                        Topline := pt^.FwdRef;
                     end
                  else
                     begin
                        Lineno := Succ (Lineno);
                     end;
               end;
```

```
          end;
      end;
    p := p^.Fwdlink;
  until p = Curwin;
end; {EditRealign}

function EditCrewindow ( Top : integer;
                         Len : integer;
                         Wid : integer;
                         Fn  : varstring;
                         Cr  : integer;
                         Cc  : integer)
                             : Pwindesc ;

{  This routine is called to create a window data structure. It does
   not integrate the structure into any existing data structures;
   rather, it returns a pointer to the completed window. If no memory
   can be found for the first text line in the text stream, nil is
   returned.
}
var
  p   : Pwindesc;
  j   : integer;
  pt : Plinedesc;
  Mem : Varstring;

   function NoMem(Size : integer) : boolean;

   begin
      NoMem := (MaxAvail > 0) and
               (MaxAvail <=
               ((2*succ(Size) + 32 + SizeOf(LineDesc) + 4096) div 16));
   end;


begin {EditCrewindow}
  EditCrewindow := nil;
  p := Winstack;
  Winstack := Winstack^.Fwdlink;
  with p^ do
    begin
      Fwdlink := nil;
      Backlink := nil;
      Filename := Fn;
      Insertflag := Typeover;
      AI := true;
      Firstlineno := Top;
      Lastlineno := Pred (Top + Len);
```

```
Lmargin := 1;
Rmargin := Logscrcols;
Lineno := Cr;
Colno := Cc;
Clineno := 0;
Leftedge := 1;
RepLast := nothing;
if Fn = Nofile then
   begin
      Formatted := false;
      Header := nil;
      HdrLen := 0;
      if NoMem(Wid) then
         begin
            j := MaxAvail;
            Str(j,Mem);
            EditAppcmdnam('MaxAvail = ' + Mem + ' ');
            EditErrorMsg(35);
            exit;
         end;
      Stream := EditNewStream(Fn,Wid);
      if Stream = 0 then
         begin
            EditErrorMsg(IOstatus);
            exit;
         end;
      Curline := EditMakTxtDes(Stream,16);
      StreamDef[Stream].TofRef := Curline;
   end
else
   begin
      Stream := EditNewStream(Fn,0);
      if Stream = 0 then
         begin
            EditErrorMsg(IOstatus);
            exit;
         end;
      with StreamDef[Stream] do
         begin
            pt := Edit(Stream,TofRef,rt);
            Curline := TofRef;
            with pt^ do
               begin
                  j := Bufflen;
                  while Txt^[j] = ' ' do j := pred(j);
                  if Txt^[j] = EndHdr then
                     begin
                        GetMem(Header,succ(Bufflen));
```

```
                                HdrLen := Bufflen;
                                for j := 1 to Bufflen do
                                   case Txt^[j] of
                                      '-'  : Header^[j] := chr(Horizontal);
                                      '|'  : Header^[j] := chr(TConnect);
                                      else   Header^[j] := Txt^[j];
                                   end;
                                Formatted := True;
                                EditUserPush(^Z);
                             end;
                       end;
                end;
          end;
       TopLine := Curline;
       if Curline <> 0 then EditCrewindow := p else EditErrorMsg(IOstatus);
    end;
end; {EditCrewindow}

procedure EditDelline  (p : dataref) ;
{  This routine deletes a line from a text stream and makes sure that
   it doesn't corrupt any windows doing it.
}

var
  w       : Pwindesc;
  bak,fwd : dataref;
  pt      : Plinedesc;


begin {EditDelline}

  { Check window Toplines, Curlines, and shortening window's span }

  w := Curwin;
  pt := Edit(w^.Stream,p,rd);
  repeat
    with w^ do if Stream = Curwin^.Stream then
      begin
         if (pt^.FwdRef = 0) and (pt^.BakRef = 0) then
            begin
               {just space out the line and exit}
               pt := Edit(Stream,p,wt);
               Fillchar (pt^.Txt^,pt^.BuffLen, ' ');
               exit;
            end;
         if p = Topline then
            if pt^.FwdRef <> 0 then
               Topline := pt^.FwdRef
```

```
            else
               Topline := pt^.BakRef;
         if p = Curline then
            if pt^.FwdRef <> 0 then
               Curline := pt^.FwdRef
            else
               Curline := pt^.BakRef;
      end;
    w := w^.Fwdlink
  until w = Curwin;

  with Curwin^ do
     begin
        bak := pt^.BakRef;
        fwd := pt^.FwdRef;
        if bak <> 0 then
           begin
              pt := Edit(Stream,bak,wd);
              pt^.FwdRef := fwd;
           end
        else
           begin
              StreamDef[Stream].TofRef := fwd;
           end;
        if fwd <> 0 then
           begin
              pt := Edit(Stream,fwd,wd);
              pt^.BakRef := bak;
           end;
        EditDesTxtDes(Stream,p);
     end;
end; {EditDelline}

function  EditInsbuf  (Ncols : integer) : boolean ;
{  This routine makes a text descriptor and inserts it into the text
   stream for the current window after the current line. If memory
   cannot be acquired for the line, false is returned, else true. Once
   the line is inserted, the current window's Curline pointer is moved
   to point to the new line.
}
var  p,fwd : dataref;
     pt    : Plinedesc;

begin {EditInsbuf}
  with Curwin^ do
    begin
      p := EditMaktxtdes (Stream,Ncols);       {Make new text buffer}
      if p = 0 then
```

```
          begin
             EditInsbuf := false;
             exit;
          end;
        EditInsbuf := true;
        pt := Edit(Stream,Curline,wd);
        fwd := pt^.FwdRef;
        pt^.FwdRef := p;
        pt := Edit(Stream,p,wd);
        with pt^ do
          begin
            BakRef := CurLine;
            FwdRef := fwd;
          end;
        if fwd <> 0 then
          begin
             pt := Edit(Stream,fwd,wd);
             pt^.BakRef := p;
          end;
        Curline := p;
      end;
  end; {EditInsbuf}

procedure EditPushtbf  (Ch : byte) ;
{  This routine pushes the argument onto the front of the typeahead
   buffer, unlike UserPush which appends the character to the back of
   the typeahead circular buffer.
}
begin {EditPushtbf}
  if Succ (Circin) mod Deftypahd = Circout then
    begin
      EditErrormsg (21);
      Typbufovl := true                {Tell caller to stop pushing chars}
    end
  else
    begin
      Circout := Pred (Circout+Deftypahd) mod Deftypahd;
      Circbuf [Circout] := chr (Ch);
      Typbufovl := false
    end
end; {EditPushtbf}

  function  EditScanFwd (p         : dataref;
                         Pattern   : Varstring;
                         var c     : integer;
                         var l     : integer;
                         IgnCase   : boolean;
                         WholeWord : boolean) : dataref;
```

```
{  This routine helps out all of the search commands (search and
   replace, search and do, and incremental search) by scanning
   in the current windows stream for the next occurrence of pattern.
   If the pattern is not found, nil is returned. Otherwise, the
   returned pointer points to the line containing the string, and
   "c", the column number, is positioned to the beginning of the
   string.  l is the line number relative to the first line of the
   search (0 is the first line).
}
var
  q     : dataref;
  Pcol  : integer;
  Plen  : integer;
  Match : boolean;
  PCh   : Char;
  Ch    : Char;
  pt    : Plinedesc;
  SvPat : Varstring;

begin {EditScanFwd}
  EditScanFwd := 0;           {Assume failure}
  if p = 0 then exit;         {Can be called with a nil line pointer}
  q := p;
  Match := false;
  l := 0;
  SvPat := Pattern;
  pt := Edit(Curwin^.Stream,q,rt);
  repeat
    Pattern := SvPat;
    Plen := Byte (Pattern [0]);
    Pcol := 1;
    if c + Plen > pt^.BuffLen then  {If no point continuing on this line,}
    begin                           {try next line}
      EditBreathe;                  {check for abort once a line}
      if Abortcmd then
        exit;
      q := pt^.FwdRef;
      pt := Edit(Curwin^.Stream,q,rt);
      l := succ(l);
      c := 1
    end;
    if q = 0 then exit;         {Don't search past end of stream!}
    with pt^ do
      repeat
        if IgnCase then
          begin
            PCh := Upcase (Pattern [Pcol]);
```

```
    Ch := Upcase (Txt^ [Pred(Pcol+c)]);
  end
else
  begin
    PCh := Pattern [Pcol];
    Ch := Txt^ [Pred(Pcol+c)];
  end;

  {Now we determine whether or not the pattern character matches
   the character being searched.  The syntax is very similar to
   MSDOS Command.com (i.e., '*' means any string of characters and
   '?' means any single character) except that we require that the
   wildcard characters be introduced by a '\' a la Unix (tm) so
   that (a) the wildcards can also be used literally and (b) we
   can introduce other regular expressions at some future time.}

  if Pch <> '\' then Match := (Pch = Ch);
  if (Pch = '\') and (Plen > 1) then
     begin

        {delete the '\' from the Pattern and adjust Plen and
         Pch accordingly}

        Delete(Pattern,Pcol,1);
        Plen := pred(Plen);
        Pch := Pattern[Pcol];

        case Pch of
           '?'  :  begin
                      Pch := Ch;
                      Match := true;
                   end;
           '*'  :  begin
                      Pch := Ch;
                      Match := true;

                      {look ahead one char in both strings and if
                       there is not a match then insert a '\*'
                       after Pch and adjust Plen}

                      if ((Pcol + c) < pt^.BuffLen) and
                         (Pattern[succ(Pcol)] <> Txt^[Pcol + c]) and
                         (not(Txt^[Pcol + c] in Separators)) then
                         begin
                            Insert('\*',Pattern,succ(Pcol));
                            Plen := Plen + 2;
                         end;
                   end;
```

```
                '#'  :  if Ch in Digits then
                           begin
                              Pch := Ch;
                              Match := true
                           end
                        else Match := false;
                '$'  :  begin
                           if Ch in Digits then
                              begin
                                 Pch := Ch;
                                 Match := true;

                                 {look ahead one char in both strings and if
                                  there is not a match then insert a '\$'
                                  after Pch and adjust Plen}

                                 if ((Pcol + c) < pt^.BuffLen)
                                                  and
                                    (Pattern[succ(Pcol)] <> Txt^[Pcol + c])
                                                  and
                                    (not(Txt^[Pcol + c] in Separators)) then
                                    begin
                                       Insert('\$',Pattern,succ(Pcol));
                                       Plen := Plen + 2;
                                    end;
                              end
                           else
                              Match := false;
                        end;
                '\'  :  if Ch = '\' then Match := true else Match := false;
                else Match := (Pch = Ch);
             end;
          end;
       if (Pch = '\') and (Plen < 1) then Match := false;
       Pcol := Succ(Pcol);

  until (Pcol > Plen) or (not Match);

{if we Matched the pattern, terminate; else try next col/line}

if Match then
  if not WholeWord then
  begin
    EditScanFwd := q;
    exit;
  end else
    if ((c = 1) or (pt^.Txt^[Pred(c)] in Separators)) and
       ((c = Succ(pt^.BuffLen - Plen)) or
```

```
              (pt^.Txt^[c+Plen] in Separators)) then
          begin
            EditScanFwd := q;
            exit;
          end;
      c := Succ(c)                    {try one column over}
    until false;     {all exits are from middle of loop}
  end; {EditScanFwd}

function EditPos(s : Varstring; p : Ptextline; l : integer)  :  integer;
{  This function is just like the Turbo POS function except it operates on
   dynamic strings on the heap.
}

var Pcol : integer;
    Plen : integer;
    c    : integer;
    Match: boolean;

begin
   EditPos := 0;
   Plen := Length(s);
   c := 1;
   Match := false;
   repeat
      if c + Plen > l then exit;
      Pcol := 1;
      repeat
         Match := ( UpCase(s[Pcol]) = UpCase(p^[pred(Pcol + c)]) );
         Pcol := succ(Pcol);
      until (Pcol > Plen) or (not Match);
      if Match then
         begin
            EditPos := c;
            exit;
         end;
      c := succ(c);
   until false; {all exits from middle of loop}
end;

procedure EditUpcase (var s : Varstring);
{  This routine converts every lowercase letter to an uppercase
   letter in the argument string.
}
var
  i  : Byte;
begin {EditUpcase}
  for i := 1 to Byte (s[0]) do s[i] := UpCase(s[i]);
```

```
end; {EditUpcase}

procedure EditColorLine(ColorOn : boolean);
{  This routine sets/unsets the Colored flag in the current line so that
   EditUpdphyscr will display it in Usercolor instead of Txtcolor
   and the update routines can find it.
}

var pt : Plinedesc;
    n  : string[3];

begin {EditColorLine}
  with Curwin^ do
     begin
        pt := Edit(Stream,CurLine,wd);
        if ColorOn then
           pt^.Flags := pt^.Flags or Colored
        else
           pt^.Flags := pt^.Flags and not(Colored);
     end;
end; {EditColorLine}

procedure EditColorFile(ColorOn : boolean; start,finish : dataref);
{  This routine sets/unsets the Colored flag on every line from start to
   finish in the current window so that they will be displayed in Usercolor
   instead of Txtcolor and the update routines can find them.
}

var  pt   : Plinedesc;
     last : dataref;

begin {EditColorFile}
   with Curwin^ do
      begin
         repeat
            pt := Edit(Stream,start,wd);
            if ColorOn then
               pt^.Flags := pt^.Flags or Colored
            else
               pt^.Flags := pt^.Flags and not(Colored);
            last := start;
            start := pt^.FwdRef;
         until (last = finish) or (start = 0);
      end;
end; {EditColorFile}
```

6. SCREEN MANAGEMENT

```
{
*******************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     screen.tt                   Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  Routines to update the screen.  The update sequence is always
           interruptable and follows the sequence:
           1) update the current line in the current window
           2) update the status line of the current window
           3) update the the contents of all windows
           4) update the command line
*******************************************************************************
}

                  { LOW-LEVEL SCREEN UPDATE ROUTINES }
                  { USED AS PROVIDED BY BORLAND, INC.}

{
procedure MoveToScreen(Var Source,Dest; Length: Integer);
procedure MoveFromScreen(Var Source,Dest; Length: Integer);
procedure EditHscroll;
procedure EditUpdrowasm {(Row:byte)};
}

function EditWrline {(var s : Textline;
                      Row : byte;
                      Txtcolor : byte) : boolean};
{  This routine copies the contents of s into the screen array indexed
   by the specified row number. All of the attribute bytes in the row
   are set to the Txtcolor argument except column verticals which are
   set to VertColor.
}
var
  Counter : byte;
  Adress : integer;
  Change : boolean;
  Newvalue : integer;
  Wx,Wy : byte;
```

```
    Nv : array [1..2] of byte absolute Newvalue;
    Ch : char;

begin
    Change := false;
    Adress := ofs (Screen [Row, 1]);
    for Counter := 1 to Physcrcols do
    begin
      Nv [1] := ord (s [Counter]);
      if Nv[1] = Vertical then
         Nv[2] := VertColor
      else
         Nv [2] := Txtcolor;
      if Newvalue <> Memw [seg (Screen) : Adress] then
      begin
        Memw [seg (Screen) : Adress] := Newvalue;
        Change := true;
      end;
      Adress := Adress + 2;
    end;
    EditWrline := Change;
end;

  procedure EditUpdwinsl {(w : Pwindesc)} ;
  {  This routine updates the status line for the specified window. }
  var
    St   : Varstring;
    Whdr : Textline;
    p    : Pwindesc;
    wn   : Byte;
    fcol : integer;

  begin {EditUpdwinsl}

    p := Window1;
    wn := 1;
    while p <> w do
      Begin
        p := p^.Fwdlink;
        wn := Succ (wn)
      end;
    UserStatusline(wn,w);
    if wn = 0 then exit;
    St := w^.Filename;
    if St = Nofile then St := St + chr(ord('0') + w^.Stream);
    Fillchar (Whdr[1], Physcrcols, chr(Horizontal));
    fcol := (Physcrcols div 2) - (Length(St) div 2);
    Move (St[1], Whdr [fcol], Length (St));
```

```
  Whdr[2] := chr(w^.WindNo + ord('0'));
  if w^.LeftEdge = 1 then Whdr[1] := chr(Vertical)
     else Whdr[1] := chr(UpperRight);
  Whdr[3] := chr(LConnect);
  if EditWrline (Whdr, w^.Firstlineno, Bordcolor) then
     EditUpdrowasm (w^.Firstlineno);
end; {EditUpdwinsl}

procedure EditUpdwindow  (p : Pwindesc) ;
{  This routine updates the status line, then updates every
   line in the text region by calling EditWrline and EditUpdrowasm.
}
var
  t        : Textline;
  q        : dataref;
  r        : integer;
  l        : integer;
  Thecolor : integer;
  i,j      : integer;
  ch       : char;
  pt       : Plinedesc;

begin {EditUpdwindow}
  with p^ do
    begin
      q := Topline;
      r := Succ (Firstlineno);
      repeat
        If EditKeypressed then exit;
        pt := Edit(Stream,q,rt);
        if q <> 0 then with pt^ do
          begin
            l := Succ (BuffLen - Leftedge);
            if l > Logscrcols then l := Logscrcols;
            if (Flags and Colored) <> 0 then
              Thecolor := Usercolor
            else
              Thecolor := Txtcolor;
            FillChar(t[1],Physcrcols,' ');
            if l > 0 then Move (Txt^ [Leftedge], t [1], l);
            if Formatted then
               begin
                  i := 1;
                  ch := Header^[pred(i) + LeftEdge];
                  while (ch <> EndHdr) and (i <= Physcrcols) do
                     begin
                        if ch = chr(TConnect) then t[i] := chr(Vertical);
                        i := succ(i);
```

```
                        ch := Header^[pred(i) + LeftEdge];
                      end;
                  end;
              if EditWrline (t, r, Thecolor) then
                EditUpdrowasm (r);
              q := FwdRef;
            end
          else
            begin
              Fillchar (t [1], Logscrcols, ' ');
              if EditWrline (t, r, Txtcolor) then
                EditUpdrowasm (r)
            end;
          r := Succ (r)
        until r > Lastlineno
      end;
    EditUpdwinsl (p);
end; {EditUpdwindow}

procedure EditUpdphyscr;
{  This routine calls Updwindow for each window in the display,
   effectively updating the entire screen.
}
var
  p        : Pwindesc;
  t        : Textline;
  l        : integer;
  r        : integer;
  Thecolor : integer;
  i        : integer;
  pt       : Plinedesc;
  ch       : char;

begin {EditUpdphyscr}

  { Update the line we're editing next }

  with Curwin^ do
    begin
      pt := Edit(Stream,Curline,rt);
      l := Succ (pt^.BuffLen - Leftedge);
      if l > Logscrcols then l := Logscrcols;
      if (pt^.Flags and Colored) <> 0 then
        Thecolor := Usercolor
      else
        Thecolor := Txtcolor;
      r := Firstlineno + Lineno;          {For speed}
      Fillchar (t [1], Logscrcols, ' ');
```

```
      if l > 0 then Move (pt^.Txt^ [Leftedge], t [1], l);
      if Formatted then
         begin
            i := 1;
            ch := Header^[pred(i) + LeftEdge];
            while (ch <> EndHdr) and (i <= Physcrcols) do
               begin
                  if ch = chr(TConnect) then t[i] := chr(Vertical);
                  i := succ(i);
                  ch := Header^[pred(i) + LeftEdge];
               end;
         end;
      if EditWrline (t, r, Thecolor) then EditUpdrowasm (r)
    end;

  { Update the command line next }

  if EditWrline (Cmdlinest, Physcrsig, Cmdcolor) then
    EditUpdrowasm (Physcrsig);

  { if we've got things to do elsewhere, don't update any more of the screen }

  if EditEdistat then exit;

  { Update the rest of the screen window by window }

  p := Curwin;
  repeat
    EditUpdwindow (p);
    p := p^.Fwdlink
  until EditEdistat or (p = Curwin);
end; {EditUpdphyscr}


                  { Pulldown Menu System specific screen routines }
                  { used as provided by Borland, Inc.             }

{
procedure SetMemAddress(Col,Line:byte);
procedure WriteChar(ch:byte);
procedure WriteString(Var st);
function ReadChar : char;
function GetCursorMode: integer;
procedure SetCursorMode(Mode : integer);
procedure SaveScreen(Var Adr; Num:byte);
procedure RestoreScreen(var Adr; Num:byte);
}
```

7. CURSOR AND WINDOW MAINTENANCE

```
{
*******************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     fastcom.tt                  Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  These are all the routines that move the cursor and take care of
           aligning the window as a result of the movement.
*******************************************************************************
}

procedure EditNewLine;
{  This routine moves the contents of the current line to the right
   of the cursor down to a new line and positions to column 1 of that
   line, if in insert mode.  If not in insert mode, the cursor is
   positioned to column 1 of the next line, without text movement.
}
var
  pt1,pt2 : Plinedesc;
  i : integer;
  c : integer;
  L : integer;

   procedure EditAutoIndentLine;
   {  This routine inserts spaces at the beginning of a line so
      that it lines up with the first nonblank character of the
      line above.
   }
   var
     p   : dataref;
     i,j : integer;
     bak : dataref;
     pt  : Plinedesc;

   begin {EditAutoIndentLine}
   with Curwin^ do
      begin
         pt := Edit(Stream,Curline,rd);
         bak := pt^.BakRef;
```

```
        pt := Edit(Stream,bak,rt);
        with pt^ do
           begin
              i := 1;
              while (i < BuffLen) and (Txt^ [i] = ' ') do i := Succ (i);
              if i = BuffLen then i := 1;
              Colno := i;
           end;
        pt := Edit(Stream,Curline,wt);
        with pt^ do
           begin
              j := BuffLen;
              if not EditSizeLine(Stream,Curline,Pred(i) + j) then
                 begin
                    EditErrormsg (41);
                    EditRealign;
                    exit
                 end;
              if i > 1 then
                 begin
                    move(Txt^ [1], Txt^ [i], j);
                    fillchar(Txt^[1], Pred(i), ' ');
                 end;
           end;
      end;
   end; {EditAutoIndentLine}


begin {EditNewLine}
  EditChangeFlag := true;
  with Curwin^ do
    begin
      if Insertflag = Insert then
        begin                    {Creates new line and moves down}
          if not EditInsBuf(16) then
             begin
                EditErrormsg (35);
                exit;
             end;
          EditIncline;
          if Lineno = (Lastlineno-Firstlineno) then
             begin
                pt1 := Edit(Stream,Topline,rd);
                Topline := pt1^.FwdRef;
             end
          else
            Lineno := Succ (Lineno);
          Leftedge := 1;
```

```
    { Now split the line: text on line to be split to right of cursor
      goes to the new line, then pad the ends of the old and the new
      with spaces }

    pt2 := Edit(Stream,Curline,wt);
    pt1 := Edit(Stream,pt2^.BakRef,wt);
    c := pt1^.BuffLen;
    while (c > 1) and (pt1^.Txt^ [c] = ' ') do c := Pred (c);
    L := Succ (pt1^.BuffLen - Colno);
    if L < 1 then L := 1;
    if not EditSizeLine(Stream,Curline,L) then
       begin
          EditErrorMsg(35);
          exit;
       end;
    if c >= Colno then
      begin
        Move (pt1^.Txt^ [Colno], pt2^.Txt^ [1], L);
        Fillchar (pt1^.Txt^ [Colno], L, ' ')
      end;

    if AI then
      EditAutoIndentLine
    else                          {if no autoindent mode}
      Colno := Lmargin;
  end
else
  begin                           {Typeover mode}
    pt1 := Edit(Stream,Curline,rd);
    if pt1^.FwdRef = 0 then
      begin                       {Create line if at end of file}
        if not EditInsBuf(16) then;
      end
    else
      Curline := pt1^.FwdRef;
    EditIncline;
    if Lineno = (Lastlineno - Firstlineno) then
       begin
          pt1 := Edit(Stream,Topline,rd);
          Topline := pt1^.FwdRef;
       end
    else
      Lineno := Succ (Lineno);
    Leftedge := 1;

   if AI then
      begin                           {Autoindent mode}
```

```
                pt2 := Edit(Stream,Curline,rd);
                pt1 := Edit(Stream,pt2^.BakRef,rt);
                i := 1;
                while (i < pt1^.BuffLen) and (pt1^.Txt^ [i] = ' ') do
                   i := Succ (i);
                if i = pt1^.BuffLen then i := 1;
                Colno := i;

              end
            else                              {If no autoindent mode}
              Colno := 1
          end
      end;
    EditRealign             {Realign other windows; we inserted stuff}
  end; {EditNewLine}

¦ procedure EditLeftChar;
¦ {  This routine moves the cursor left one character. If the cursor is
¦    in column one, it is positioned immediately after the first non-blank
¦    character on the previous line, if one exists.
¦ }
¦
¦ var pt : Plinedesc;
¦
¦ begin {EditLeftChar}
¦   with Curwin^ do
¦     if Colno > 1 then
¦       begin
¦         Colno := Pred (Colno);
¦         if Colno < Leftedge then
¦           Leftedge := Colno
¦       end;
¦ end; {EditLeftChar}
¦
¦ procedure EditRightChar;
¦ {  This routine advances the cursor one character position. }
¦ begin {EditRightChar}
¦   with Curwin^ do
¦     if Colno < Pred (StreamDef[Stream].RecLen) then
¦       Colno := Succ (Colno)
¦ end; {EditRightChar}

  procedure EditScrollUp;
  {  This routine scrolls the current window up one line. }

  var pt : Plinedesc;

  begin {EditScrollUp}
```

```
  with Curwin^ do
    begin
      pt := Edit(Stream,Topline,rd);
      if pt^.BakRef <> 0 then
        begin
          Topline := pt^.BakRef;
          if Lineno = (Lastlineno - Firstlineno) then
            begin
              EditDecline;
              pt := Edit(Stream,Curline,rd);
              Curline := pt^.BakRef
            end
          else
            Lineno := Succ (Lineno)
        end
    end
end; {EditScrollUp}

procedure EditUpLine;
{  This routine moves the cursor up one line in the window, scrolling
   it if necessary.
}

var pt : Plinedesc;

begin {EditUpLine}
  with Curwin^ do
    begin
      pt := Edit(Stream,Curline,rd);
      if pt^.BakRef <> 0 then
        begin
          EditDecline;
          Curline := pt^.BakRef;
          if Lineno = 1 then
            begin
               pt := Edit(Stream,Topline,rd);
               Topline := pt^.BakRef;
            end
          else
            Lineno := Pred (Lineno)
        end
    end
end; {EditUpLine}

procedure EditScrollDown;
{  This routine scrolls the screen down one line. }

var pt : Plinedesc;
```

```
begin {EditScrollDown}
  with Curwin^ do
    begin
      pt := Edit(Stream,Topline,rd);
      if pt^.FwdRef <> 0 then
        begin
          Topline := pt^.FwdRef;
          if Lineno = 1 then
            begin
              EditIncline;
              pt := Edit(Stream,Curline,rd);
              Curline := pt^.FwdRef
            end
          else
            Lineno := Pred (Lineno)
        end
    end
end; {EditScrollDown}

procedure EditDownLine;
{  This routine moves the cursor down one line, scrolling the window
   if necessary.
}

var pt : Plinedesc;

begin {EditDownLine}
  with Curwin^ do
    begin
      pt := Edit(Stream,Curline,rd);
      if pt^.FwdRef <> 0 then
        begin
          EditIncline;
          Curline := pt^.FwdRef;
          if Lineno = (Lastlineno - Firstlineno) then
             begin
                pt := Edit(Stream,Topline,rd);
                Topline := pt^.FwdRef;
             end
          else
            Lineno := Succ (Lineno)
        end
    end
end; {EditDownLine}


procedure EditDeleteRightChar;
```

```
{  This routine deletes the character underneath the cursor. }

var  i, j : integer;
     pt   : Plinedesc;

begin {EditDeleteRightChar}
  EditChangeFlag := true;
  pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with Curwin^, pt^ do
    begin
      if Colno >= Bufflen then
        if not EditSizeLine(Stream,Curline,succ(Colno)) then
          begin
             EditErrormsg (41);
             exit
          end;
      i := BuffLen;
      while (i > 1) and (Txt^ [i] = ' ') do i := Pred (i);
      for j := Colno to Pred (BuffLen) do
         Txt^ [j] := Txt^ [Succ (j)];
      Txt^ [BuffLen] := ' ';
    end
end; {EditDeleteRightChar}

procedure EditTab;
{  This routine processes the tabs by moving the cursor to character position
   of the next word in the prior line.
}
var
  c : integer;
  p : Plinedesc;

begin {EditTab}
  with Curwin^ do
    begin
      p := Edit(Stream,Curline,rd);
      if p^.BakRef <> 0 then
         begin
            p := Edit(Stream,p^.BakRef,rt);
            c := Colno;
            while (p^.Txt^[c] <> ' ') and (c < p^.Bufflen) do c := succ(c);
            while (p^.Txt^[c] = ' ') and (c < p^.Bufflen) do c := succ(c);
            if c <= StreamDef[Stream].RecLen then Colno := c;
         end;
    end;
end; {EditTab}

procedure EditBackTab;
```

```
{  This routine processes the back tabs (^b) by moving the cursor
   to the first character position of the previous word in the
   prior line.
}
var
  c : integer;
  p : Plinedesc;

begin {EditBackTab}
  with Curwin^ do
    begin
      p := Edit(Stream,Curline,rd);
      if p^.BakRef <> 0 then
         begin
            p := Edit(Stream,p^.BakRef,rt);
            c := pred(Colno);
            while (p^.Txt^[c] = ' ') and (c > 1) do c := pred(c);
            while (p^.Txt^[c] <> ' ') and (c > 1) do c := pred(c);
            c := succ(c);
            if c <= StreamDef[Stream].RecLen then Colno := c;
         end;
    end;
end; {EditBackTab}

procedure EditDeleteLeftChar;
{  This routine deletes the character to the left of the cursor. if
   the cursor is in column one, nothing happens.
}
var  i : integer;
     pt: Plinedesc;

begin {EditDeleteLeftChar}
  EditChangeFlag := true;
  pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with Curwin^, pt^ do
    begin
      if Colno = 1 then
        begin
        end
      else if Colno > BuffLen then
        Colno := Pred (Colno)      {We're out in space, so nothing to del}
      else
        begin
           for i := Pred (Colno) to Pred (BuffLen) do
              Txt^ [i] := Txt^ [Succ (i)];
           Txt^ [BuffLen] := ' ';
           Colno := Pred (Colno);
        end;
```

```
    end
end; {EditDeleteLeftChar}

{commands removed from qcmd overlay area because they are called by other}
{commands in qcmd start here}

procedure EditEndLine;
{  This routine positions the cursor to the right of the last non-
   blank character in the line.
}

var pt : Plinedesc;

begin
  pt := Edit(Curwin^.Stream,Curwin^.Curline,rt);
  with Curwin^, pt^ do
  begin
    Colno := BuffLen;
    while (Txt^ [Colno] = ' ') and (Colno > 1) do
      Colno := Pred (Colno);
    if Colno < BuffLen then
      if Txt^ [Colno] <> ' ' then
    Colno := Succ (Colno);              {Position after last char}
  end;
end;

procedure EditAlignLast;
{  This routine puts the current line as low in the window as possible
   (the top line of the window must always contain a line of the file).
}
var
  Pagesize: integer;
  pt      : Plinedesc;

begin {EditAlignLast}
  with Curwin^ do begin
    Lineno := 1;
    Pagesize := Lastlineno - FirstLineno;
    Topline := Curline;
    pt := Edit(Stream,Topline,rd);
    while (Lineno < Pagesize) and (pt^.BakRef <> 0) do
    begin
      Topline := pt^.BakRef;
      pt := Edit(Stream,Topline,rd);
      Lineno := Succ (Lineno)
    end;
  end;
end; {EditAlignLast}
```

```
procedure EditWindowTopFile;
{  This routine positions the cursor at the top of the text stream in
   the current window, making the first line in the stream be the first
   line in the window.
}
var  p : dataref;
     pt: Plinedesc;

begin {EditWindowTopFile}
  with Curwin^ do
    begin
      p := Topline;
      pt := Edit(Stream,p,rd);
      while pt^.BakRef <> 0 do
         begin
            p := pt^.BakRef;
            pt := Edit(Stream,pt^.BakRef,rd);
         end;
      Topline := p;
      Curline := p;
      Lineno := 1;
      Colno := 1;
      Leftedge := 1;
      if Formatted then EditScrollDown;
    end
end; {EditWindowTopFile}

procedure EditWindowBottomFile;
{  This routine positions the window over the text stream so that the
   last line in the stream is on the last line in the window. The
   cursor is placed on the last line, to the right of the last non-
   blank character on the line.
}
var  p : dataref;
     i : integer;
     pt: Plinedesc;

begin {EditWindowBottomFile}
  with Curwin^ do
    begin
      p := Topline;
      pt := Edit(Stream,p,rd);
      while pt^.FwdRef <> 0 do
         begin
            p := pt^.FwdRef;
            pt := Edit(Stream,pt^.FwdRef,rd);
         end;
```

```
        Curline := p;
        Colno := 1;
        Leftedge := 1;
        EditAlignLast;
        EditEndLine;
      end
  end; {EditWindowBottomFile}

| procedure EditDeleteLine;
| {  This routine processes the delete line command by deleting the
|    current line from the current window. If there is only one line
|    in the text stream, it is filled with spaces.
| }
|
| begin {EditDeleteLine}
|   EditChangeFlag := true;
|   EditDelline(Curwin^.Curline);
|   Curwin^.Colno := 1;
|   EditRealign;                          {Realign other windows};
| end; {EditDeleteLine}

| procedure EditWindowGoto (Wno : byte);
| {  This routine moves the cursor to the specified window. }
| var
|   i : integer;
|   w : Pwindesc;
|
| begin {EditWindowGoto}
|   if Wno < 1 then exit;
|
|   w := window1^.Backlink;
|   for i := 1 to Wno do
|     w := w^.Fwdlink;
|   Curwin := w;
|   Updcurflag := true;
|
| end; {EditWindowGoto}

| procedure EditAskfor  (var s : Varstring) ;
| {  This routine places the cursor on the command line (at the end) and
|    waits for the user to type a string. He terminates the string with
|    a carriage return. Control characters print with a ^ symbol, and
|    take up twice as much room as printable characters. The characters
|    typed are then passed back to the caller.
| }
| const
|   Quote = '"';
|
```

```
var
  Firstcol  : integer;
  r         : integer;
  t         : Varstring;
  Flags     : array [0..Defnocols] of boolean;
  Ch        : byte;
  i         : integer;
  Count     : integer;
  v         : integer;

  function EditGetch : byte;
    begin {EditGetch of EditAskfor}
      if Interactive then
        begin
          if EditWrline (Cmdlinest,Physcrsig, Cmdcolor) then
            EditUpdrowasm (Physcrsig);
          while not EditKeypressed do ;
          EditGetch := EditGetinput
        end
      else if EditKeypressed then
        EditGetch := EditGetinput
      else
        EditGetch := Notavailable
    end; {EditGetch of EditAskfor}

begin {EditAskfor}
  if Abortcmd then exit;              {If Cpabort has requested we stop}
  t := '';
  for i := 0 to Physcrcols do
    Flags [i] := false;               {Set flag in line position if ctrlchar}
  Asking := true;                     {Let Updphyscr know we're on sig lin}
  Firstcol := Cmdcol;                 {This is where we start writing}
  Cmdcolor := CmdOnColor;             {Hilite command line to get user attention}
  Count := 0;
  v := 0;
  Updcurflag := true;
  while v <> Ctrlm do
    begin
      EditBackground;                 {Keep updating screen, etc.}
      if Abortcmd then
        begin                         {Cpabort has told us to stop}
          s := '';
          Updcurflag := true;
          Asking := false;
          EditZapcmdnam;
          exit
        end;
      Ch := EditGetch;
```

```
      if Ch <> Notavailable then
        begin                          {Found a char}
          if Abortcmd then
            begin
              s := '';
              Updcurflag := true;
              Asking := false;
              EditZapcmdnam;
              exit
            end;
          v := Ch;
          if (v = Ctrlh) or (v = Del) then
            begin                      {We must back up}
              if Cmdcol <> Firstcol then
                begin                  {Erase prev char and back up}
                  Delete (t, Byte (t [0]), 1);
                  Cmdcol := Pred (Cmdcol);
                  Cmdlinest [Cmdcol] := ' ';
                  if Flags [Count] then
                    begin              {Get rid of ^ too}
                      Flags [Count] := false;
                      Cmdcol := Pred (Cmdcol);
                      Cmdlinest [Cmdcol] := ' '
                    end;
                  Count := Pred (Count);
                  Updcurflag := true
                end
            end
          else if Ch = 13 then
            begin                      {Drop out of loop}
            end
          else if (Ch > 0) and (Ch < 32) then
            begin                      {Control char}
              if (Cmdcol + 2) > Physcrcols then {scroll left 2}
                begin
                   delete(Cmdlinest,1,2);
                   Cmdlinest := Cmdlinest + '  ';
                   Cmdcol := Cmdcol - 2;
                end;
              t := t + chr (Ch);
              Updcurflag := true;
              Cmdlinest [Cmdcol] := '^';
              Cmdlinest [Cmdcol+1] := chr (Ch+64);
              Count := Succ (Count);
              Flags [Count] := true;
              Cmdcol := Cmdcol + 2
            end
          else
```

```
            begin                    {Add this one to the buffer}
              if Cmdcol > Physcrcols then {scroll left 1}
                begin
                   delete(Cmdlinest,1,1);
                   Cmdlinest := Cmdlinest + ' ';
                   Cmdcol := pred(Cmdcol);
                end;
              t := t + chr (Ch);
              Updcurflag := true;
              Cmdlinest [Cmdcol] := chr (Ch);
              Count := Succ (Count);
              Cmdcol := Succ (Cmdcol)
            end
        end
    end;
  s := t;
  Updcurflag := true;
end; {EditAskfor}

procedure EditPrctxt (ch : byte);
{  This routine inserts the character specified as the argument into
   the text of the current window at the current cursor position. If
   insert mode is turned on, all the characters to the right of the cursor
   are shifted over by one column to make room for the character.
}
var
  i       : integer;
  r       : integer;
  Cno     : integer;
  t       : Textline;
  Thecolor: integer;
  Endcol  : integer;
  p       : dataref;
  l       : integer;
  pt      : Plinedesc;
  c       : char;

begin {EditPrctxt}
  if ch = NotAvailable then exit;
  if Curwin^.Colno >= StreamDef[Curwin^.Stream].RecLen then
     begin
        EditErrorMsg(41);
        exit;
     end;
  EditChangeFlag := true;
  pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with pt^ do
     if (Flags and Colored) <> 0 then
```

```
      Thecolor := Usercolor
    else
      Thecolor := Txtcolor;

with Curwin^ do
  begin
    r := Firstlineno + Lineno;
    if Colno-Leftedge < 0 then
      Leftedge := Colno;
    if (Colno > Leftedge + Logscrcols - 2) then
      EditHscroll;

    { If we're in insert mode, we shift everyone. If at eol, do nothing }

    with pt^ do
       begin
          if Insertflag = Insert then
             begin
                l := Succ (Colno);
                if Txt^ [BuffLen] <> ' ' then
                   if l < Succ (BuffLen) then l := Succ (BuffLen);
                if BuffLen < l then
                   if not EditSizeLine(Stream,Curline,l) then
                      begin
                         EditErrormsg (41);
                         exit
                      end;
                { shift text right }
                Move (Txt^[Colno],Txt^[Succ(Colno)],BuffLen-Colno);
             end
          else
             begin
                if not EditSizeLine(Stream,Curline,Succ (Colno)) then
                   begin
                      EditErrormsg (41);
                      exit
                   end;
             end;

          { Update the line we're on and fix the cursor position }

          Txt^ [Colno] := chr (ch);
          l := Succ (BuffLen - Leftedge);
          if l > Logscrcols then l := Logscrcols;
          Fillchar (t [1], Logscrcols, ' ');
          Move (Txt^ [Leftedge], t [1], l);
          if Formatted then
             begin
```

```
                    i := 1;
                    c := Header^[pred(i) + LeftEdge];
                    while (c <> EndHdr) and (i <= Physcrcols) do
                       begin
                          if c = chr(TConnect) then t[i] := chr(Vertical);
                          i := succ(i);
                          c := Header^[pred(i) + LeftEdge];
                       end;
                 end;
              if EditWrline (t, r, Thecolor)
                 then EditUpdrowasm (r);
              Colno := Succ (Colno);
              GotoXY (Succ (Colno-Leftedge) , r);
              Updcurflag := true;
           end;
     end;
  end; {EditPrctxt}

function EditCtrlChar:Byte;
{  This routine waits for the second character of Ctrl-K, -O and
   -Q commands and displays this charcter in the first line.
}
var ch:Byte;
    s :Varstring;
    c :Char;

begin
  repeat
    EditUpdphyscr;                          {Keep trying to update screen}
    if Abortcmd then                        {If EditAbort has called us}
      begin
      EditCtrlChar := 255;
      exit;
      end;
  until EditKeypressed;
  Ch := EditGetInput;
  if  ((Ch >= $41) and (Ch <= $5A)) or    { Alpha character -- A..Z or a..z  }
      ((Ch >= $61) and (Ch <= $7A)) then  { is made into a control character }
    EditCtrlchar := Ch and $1F
  else
    EditCtrlChar := Ch;
  if ch <= 31 then c := chr(ch + 64) else c := chr(ch); {ctrl ch to upcase ch}
  s := c + ' ';                             { and put it into a display string }
  EditAppCmdnam(s);                         { that goes to the command line    }
  EditUpdPhyScr;
end;
```

8. DEBUG

```
{
*******************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     debug.tt                    Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                          under Contract F30602-C-85-0190

Function:    This include file is for debugging overlays in area 1.  It is
included from tt.pas as the first procedure/function in area 1 so that the
F)ind Runtime Error facility of the Turbo Pascal compiler can correctly
locate the error.
}
```

9. INITIALIZATION

```
{
*****************************************************************************
Program:  TT                          Version 2.0          Date: 20 Nov 87
File:     init.tt                     Version 2.00         Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  Initialization of all global variables and global structured
           typed constants used as variables.
*****************************************************************************
}

overlay procedure EditInitialize;
{  This routine is called by the initialization sequence in the
   editor program prior to calling the scheduler to initialize
   all data structures in the editor.
}
var
  Regs      : Regtype;
  p         : Pwindesc;
  r         : integer;
  c         : integer;
  i         : integer;
  j         : integer;
  k         : integer;
  x         : integer;
  n         : integer;
  Nlines    : integer;
  Lptr      : Plinedesc;
  Lp        : Plinedesc;
  UserStr   : Varstring;
  Num       : integer;
  Rnum      : real;
  Flag      : char;
  Ok        : boolean;
  ErrCd     : integer;

begin {EditInitialize}

  {initialize task scheduling parameters}
```

```
Asking := false;                    {EditAskFor does not have control}
Interactive := false;               {used by EditAskFor}
Notfound := false;                  {Set by search and replace cmds}
Abortcmd := false;                  {Set by EditAbort to stop EditAskfor}
Aborting := false;                  {Set by EditAbort to stop recursion}
Updcurflag := true;                 {Indicator to background to upd cur.}
Rundown := false;                   {Set when editor must exit}
Intrflag := Intrpt;                 {Enable input interrupts during screen upd}

{initialize keyboard input area}

Circin := 0;                        {In ptr for circular buffer}
Circout := 0;                       {Indicate buffer is empty (in=out)}
Cmdcol := 1;                        {Next cursor position on signal line}
EditUsercommandInput := 0;          {Number of chars user has pushed so far}

{initialize color attributes}

Bordcolor := 14;                    {Bright on black}
Txtcolor := 12;                     {Medium on black}
Usercolor := 15;                    {Very bright on black}
CmdOnColor := 15;                   {Very bright on black}
CmdOffColor:= 12;                   {Medium on black}
Cmdcolor := CmdOffColor;
VertColor := 14;                    {bright on black}

{ initialize block control record }

with BatRec do
   begin
      FromStrm := -1;
      FromRef := 0;
      ToStrm := -1;
      ToRef := 0;
   end;

{ Initialize screen array and other stuff }

for r := 1 to Defnorows do
  for c := 1 to Defnocols do
    with Screen [r,c] do
      begin
        Ch := chr (0);              {Have the editor clean up the screen}
        Color := Txtcolor
      end;

{ Determine screen type for screen updating procedure, EditUpdrowasm }
```

```
Regs.AX := $0F00;                    {See turbo 3.0 manual, pg 214}
Intr ($010, Regs);                   {BIOS INT 10H call to get screen type}
Retracemode := (Regs.AX and $00FF) <> 7;
If Retracemode then
begin
  Screenadr := $B800;                {Address of color screen}
  PNormColor  := $3000;              {Color of non-selected parts}
  PLowColor   := $7800;              {Color of selected parts    }
  PFrameColor := $0300;              {Color of frame             }

  WNormColor  := $7000;              {Color of non-selected parts}
  WLowColor   := $7000;              {Color of selected parts    }
  WFrameColor := $0300;              {Color of frame             }
end else
begin
  Screenadr := $B000;                {Address of monochrome screen}
  PNormColor := $0F00;               {As above}
  PLowColor  := $7000;
  PFrameColor := $1000;

  WNormColor := $0F00;
  WLowColor  := $7000;
  WFrameColor := $1000;
end;

{initialize physical screen attributes}

Physcrrows := Defnorows;             {Editor screen shape}
Physcrcols := Defnocols;
Logtopscr  := 5;                     {Windows don't use first 4 lines of screen}
Logscrrows := Physcrrows - Logtopscr + 1;
Logscrcols := Defnocols;
Physcrsig  := 1;                     {Line number for command line}
Cmdcol     := 1;                     {Column number for start of command line}
Tabsize := 8;                        {Distance between tab stops}

{ Allocate window structures }

Winstack   := nil;                   {Free list of windows}
for i := 1 to Succ (Defnorows div 2) do
  begin                              {Make a whole bunch of windows}
    new (p);                         {This should be the only call to new}
    p^.Fwdlink := Winstack;          {Push a window}
    Winstack := p
  end;
```

```
{initialize stream buffers}

for i := 1 to MaxStreams  do   {Initialize text stream definitions}
   begin
      with StreamDef[i] do
         begin
            InUse  := false;
            InTransaction := false;
         end;
      with BufMap[i] do new(Slot);
   end;

{create a reasonable empty StreamDesc record}

with MTStrDef do
   begin
      InUse   := false;
      DefRef  := 1;
      UsrName := Nofile;
      NewName := '';
      RecLen  := DefRecSize;
      TofRef  := 1;
      with DefHndl do
         begin
            FirstFree := 1;
            NumberFree := MaxInt;
            Int1 := 0;
            Int2 := 0;
            NumRec := 0;
         end;
      with DesHndl do
         begin
            FirstFree := 1;
            NumberFree := MaxInt;
            Int1 := 0;
            Int2 := 0;
            NumRec := 0;
         end;
      with TxtHndl do
         begin
            FirstFree := 1;
            NumberFree := MaxInt;
            Int1 := 0;
            Int2 := 0;
            NumRec := 0;
         end;
      CurIx   := 0;
      for i := 1 to MaxIndexes do with IxMap[i] do
```

```
            begin
               Ixfn := 'NOINDEX';
               Ixcn := 'NOCOLUMN';
               Opnd := false;
               with Ixh do
                  begin
                     AllowDuplKeys := true;
                     KeyL := MaxKeyLen;
                     RR := 0;
                     PP := 0;
                     for i := 1 to MaxHeight do with Path[i] do
                        begin
                           PageRef := 0;
                           ItemArrIndex := 0;
                        end;
                  end;
            end;
         InTransaction := false;
      end;


   {create a window and set up the screen}

   Curwin := EditCrewindow  (Logtopscr,    {Physical top of window}
                             Logscrrows,   {Length of window}
                             DefRecSize,   {Width of window}
                             Nofile,       {Editing no file}
                             Line1,        {Setup cursor positions}
                             Col1);

   Window1 := Curwin;                      {Define top window}
   Window1^.Fwdlink := Window1;
   Window1^.Backlink := Window1;
   Window1^.WindNo := 1;

   ClrScr;
   EditZapCmdnam;                          {Initialize command line string}
   EditUpdPhyscr;                          {show what we've done}

end;{EditInitialize}

overlay procedure EditCopyright;

type Cstr = string[41];

const

   c1 : Cstr = '[----------------------------------------]';
```

```
   c2 : Cstr = '¦                                          ¦';
   c3 : Cstr = '¦                    T T                   ¦';
   c4 : Cstr = '¦                                          ¦';
   c5 : Cstr = '¦       Rome Air Development Center        ¦';
   c6 : Cstr = '¦       Contract:  F30602-C-85-0190        ¦';
   c7 : Cstr = '¦                                          ¦';
   c8 : Cstr = '{------------------------------------------}';

   procedure DispStr(c : Cstr);

   var i : integer;

   begin
      i := 1;
      for i := 1 to Length(c) do
         case c[i] of
            '[' : write(chr(WUpperLeft));
            ']' : write(chr(WUpperRight));
            '{' : write(chr(WLowLeft));
            '}' : write(chr(WLowRight));
            '-' : write(chr(Horizontal));
            '¦' : write(chr(Vertical));
            else  write(c[i]);
         end;
      writeln;
   end;

begin
   ClrScr;
   GotoXY(20,8);  DispStr(c1);
   GotoXY(20,9);  DispStr(c2);
   GotoXY(20,10); DispStr(c3);
   GotoXY(20,11); DispStr(c4);
   GotoXY(20,12); DispStr(c5);
   GotoXY(20,13); DispStr(c6);
   GotoXY(20,14); DispStr(c7);
   GotoXY(20,15); DispStr(c8);
end;
```

10. CTRL-K COMMANDS

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     kcmd.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  These routines do the text manipulation for all the Ctrl-K
           commands.  All of these routines are overlaid in area 1 and
           thus cannot call any other routines in area 1.
*****************************************************************************
overlay procedure EditReatxtfil (Fn : Varstring);
{  This routine opens the file specified and copies it line by
   line into the current window's text stream.
}
const
  Bufsize = 512;

var
  Error      : boolean;
  Nulln      : boolean;
  Endoffile  : boolean;            {Set by read routine}
  Endofline  : boolean;
  Ch         : char;
  Nrecsread  : integer;            {Bytes read by read routine}
  Infile     : file;
  Colnosave  : integer;
  Linenosave : integer;
  i          : integer;
  Pointer    : integer;            {Next byte to read in buffer}
  Topsave    : dataref;
  Textsave   : dataref;
  Filnam     : String80;
  Buffer     : array [1..Bufsize] of char;
  x          : real;               {Number of bytes to read}
  j,k        : integer;
  pt         : plinedesc;

begin {Reatxtfil}
```

```
Assign(Infile,Fn);
Reset (Infile,1);
if EditFileerror then exit;
x := longfilesize(InFile);        {Get number of bytes to read}
if EditFileerror then exit;
with Curwin^ do
begin
  Nulln := true;
  pt := Edit(Stream,Curline,rt);
  with pt^ do
     begin
        for i := 1 to BuffLen do
           if Txt^ [i] <> ' ' then Nulln := false;
        if (BakRef = 0) and (FwdRef = 0) and Nulln then
           begin
           end
        else if not EditInsbuf (1) then  {Inserts line}
           begin
           end;
     end;
  Topsave := Topline;           {Save cursor position}
  Textsave := Curline;
  Colnosave := Colno;
  Linenosave := Lineno;
  pt := Edit(Stream,Curline,wt);
  Error := false;
  Pointer := Succ (Bufsize);        {Force read on the first time}
  Nrecsread := 0;
  Endoffile := false;
  repeat
    Error := Error or Abortcmd;
    Endofline := false;
    repeat                          { Get next char in line }
      if Pointer > Nrecsread then
      begin                         { Need to load another buffer full }
        if x > BufSize then
        begin
          Blockread (Infile, Buffer, Bufsize, Nrecsread);
          x := x - Nrecsread;   {Number of bytes left to read}
        end
        else
          Blockread (Infile, Buffer, trunc(x), Nrecsread);
        Error := Error or EditFileerror;
        Pointer := 1
      end;
      if Nrecsread = 0 then
        Ch := chr (Ctrlz)
      else
```

```
  begin
    Ch := Buffer [Pointer];
    Pointer := Succ (Pointer)
  end;
  case ord (Ch) of
    Ctrlm : begin end;
    Ctrlj : begin
              if not EditSizeLine(Stream,Curline,Colno) then;
              if not EditInsbuf (1) then
                 begin
                   EditErrormsg (40);
                   Error := true
                 end
              else
                 begin
                   Colno := 1;
                   pt := Edit(Stream,Curline,wt);
                 end;
            end;
    Ctrlz : begin
              Endofline := true;
              if not EditSizeLine(Stream,Curline,Colno) then;
              Endoffile := true
            end;
  else
  begin
     if not EditSizeLine(Stream,Curline,succ(Colno)) then
        begin
           EditErrormsg(40);
           Error := true;
        end
     else
        begin
           pt^.Txt^ [Colno] := Ch;
           Colno := Succ (Colno);
        end;
  end;
end; {case}
until Error or Endofline;
if (not Endoffile) and (not Error) then {Don't add line if not needed}
  if not EditInsbuf (1) then
     begin                    {Out of memory in middle of file}
        EditErrormsg (40);
        Error := true
     end
  else
     begin
        pt := Edit(Stream,Curline,wt);
```

```
            end;
    until Endoffile or Error;
    Close (Infile);
    Topline := Topsave;
    Curline := Textsave;
    Colno := Colnosave;
    Lineno := Linenosave;
    pt := Edit(Stream,Curline,rt);
    with pt^ do
       begin
          if BakRef = 0 then
             begin
                j := Bufflen;
                while Txt^[j] = ' ' do j := pred(j);
                if Txt^[j] = EndHdr then
                   begin
                      GetMem(Header,succ(Bufflen));
                      HdrLen := Bufflen;
                      for j := 1 to Bufflen do
                         case Txt^[j] of
                            '-'  : Header^[j] := chr(Horizontal);
                            '|'  : Header^[j] := chr(TConnect);
                            else   Header^[j] := Txt^[j];
                         end;
                      Formatted := True;
                      EditUserPush(^Z);
                   end;
             end;
       end;
    end;
  end; {EditReatxtfil}

overlay function EditWriteToFile(Fname  : VarString;
                         Start,
                         Finish : dataref) : boolean;

{  This routine writes a contiguous series of lines to a text file.
   It takes two line descriptors as pointers to the beginning and end
   of the text stream, and traverses the linked list writing the text
   to disk.
}
var
  Outfile : integer;              { represents the File handle }
  c, i    : integer;
  p       : dataref;
  Crlf, WrapCrlf : array [1..2] of char;
  Fn      : array[0..65] of char;
  Buffer  : ^byte;                { define a buffer that points to the heap }
```

```
  Regs     : RegType;
  pt       : Plinedesc;

begin { EditWriteToFile }
  EditWriteToFile := false;
  if length(Fname) < 1 then exit;
  if (Start = 0) then
  begin
    EditErrormsg (26);
    exit
  end;
  Crlf [1] := chr (Ctrlm);
  Crlf [2] := chr (Ctrlj);
  for i := 1 to Length (Fname) do   { copy filename into array }
    Fn [i-1] := Fname [i];
  Fn [Length (Fname)] := chr (0);   { set the byte after the filename to $0 }
  with Regs do
  begin
    AX := $3C00;                    { MS-DOS function $3C - Create file }
    DS := Seg (Fn [0]);             { indicate location of Filename }
    DX := Ofs (Fn [0]);
    CX := 0;                        { set file attribute read/write }
    MsDos (Regs);                   { Do the function call }
    if (Flags and 1) = 1 then       { If carry set, then there is an error  }
    begin
      EditErrormsg (AX);            { Display the error message }
      exit;
    end;
    Outfile := AX;                  { file handle identifier }
  end;
  p := Start;                       { line descriptor to start of block }
  repeat
    pt := Edit(Curwin^.Stream,p,rt);
    with pt^ do
       begin
          c := BuffLen;             { Delete trailing blanks of line }
          while (c > 1) and (Txt^[c] = ' ') do
             c := pred(c);
          with Regs do
             begin
                AX := $4000;        { MS-DOS function $40 - Write to file }
                BX := Outfile;      { pass file handle identifier }
                CX := c;            { number of Bytes to write }
                DS := Seg (Txt^[1]);{ indicate address of buffer }
                DX := Ofs (Txt^[1]);
                MsDos (Regs);       { Do the write }
                if ((Flags and 1) = 1) then  {carry set means error }
                   begin
```

```
                    EditErrormsg (6);
                    AX := $3E00;
                    BX := Outfile;
                    MsDOS (Regs);
                    exit;
                 end;
              if (AX <> c) then   { If AX not same as c, disk full }
                 begin
                    EditErrormsg (39);
                    AX := $3E00;
                    BX := Outfile;
                    MsDOS (Regs);
                    exit;
                 end;
           end;
        if p = Finish then
           p := 0
        else
           p := FwdRef;
        if p <> 0 then
           with Regs do
              begin
                 AX := $4000;     { MS-DOS function $40 - Write to file }
                 BX := Outfile;   { pass file handle identifier }
                 CX := 2;         { number of Bytes to write }
                 DS := Seg (Crlf [1]);{ address of buffer }
                 DX := Ofs (Crlf [1]);
                 MsDos (Regs);               { Do the write }
                 if (Flags and 1) = 1 then p := 0;
              end;
           if Abortcmd then p := 0;
     end;
until p = 0;
Crlf [1] := chr (Ctrlz);
Crlf [2] := chr (Ctrlz);
with Regs do
  begin
    AX := $4000;                  { MS-DOS function $40 - Write to file }
    BX := Outfile;                { pass file handle identifier }
    CX := 2;                      { number of Bytes to write }
    DS := Seg (Crlf [1]);         { indicate address of buffer }
    DX := Ofs (Crlf [1]);
    MsDos (Regs);                 { Do the write }
    AX := $3E00;                  { MS-DOS function $3E - Close file }
    BX := Outfile;                { pass file handle identifier }
    MsDos (Regs);                 { Do the close }
  end;
EditWriteToFile := true;
```

```
end; { EditWriteToFile }


  overlay procedure EditDefineTab (Size : integer);
  {  This routine is passed a new distance to be used between tab stops.
     It places the number in "Tabsize".
  }
  begin {EditDefineTab}
    if (Size > 1) and (Size < StreamDef[Curwin^.Stream].RecLen) then
      Tabsize := Size
  end; {EditDefineTab}

  overlay procedure EditExit;
  {  This command causes all open files including temporary files to be
     closed and then sets Rundown to true to terminate the scheduler and,
     consequently, the system terminates.
  }

  var p : Pwindesc;

  begin {EditExit}
    p := Window1;
    repeat
       EditDelStream(p^.Stream);
       p := p^.FwdLink;
    until p^.FwdLink = Window1;
    Rundown := true;
  end; {EditExit}

  overlay procedure EditBatBegin;
  {  This command saves the location of the current line in the current window
     in the batch control record for inspection by EditBatEnd.
  }
  begin
     with BatRec do
        begin
           FromStrm := Curwin^.Stream;
           FromRef := Curwin^.Curline;
        end;
  end;

  overlay procedure EditBatEnd;
  {  This command inspects the batch control record and if the current line in
     the current window is consistent (i.e., in the same stream and later in
     sequence) it calls EditColorFile to set the batch flags in all the lines
     from BatRec.fromref to BatRec.toref inclusively.
  }
```

```
var   pt    : Plinedesc;
      p     : dataref;

begin
   with BatRec,Curwin^ do
      begin
         if FromStrm <> Stream then exit;
         if FromRef = Curline then
            begin
               EditColorLine(true);
               exit;
            end;
         p := Curline;
         pt := Edit(Stream,p,rd);
         repeat
            p := pt^.BakRef;
            pt := Edit(Stream,p,rd);
         until (p = FromRef) or (p = 0);
         if p = FromRef then EditColorFile(true,p,Curline);
         FromStrm := -1;
         FromRef := 0;
      end;
end;

overlay procedure EditBatInclude;
{  This command includes the current line in the batch.
}

begin
   EditColorLine(true);
end;

overlay procedure EditBatExclude;
{  This command excludes the current line from the batch.
}

begin
   EditColorLine(false);
end;

overlay procedure EditBatPrior;
{  This command positons the cursor at the previous line in the batch.
}

var  pt    : Plinedesc;
     p,q   : dataref;

begin
```

```
   with Curwin^ do
      begin
         p := Curline;
         pt := Edit(Stream,Curline,rd);
         p := pt^.BakRef;
         repeat
            pt := Edit(Stream,p,rd);
            q := p;
            p := pt^.BakRef;
         until (p = 0) or ((pt^.Flags and Colored) <> 0);
         if p <> 0 then if q <> 0 then
            begin
               Curline := q;
               EditRealign;
            end;
      end;
end;

overlay procedure EditBatNext;
{  This command positions the cursor at the next line in the batch.
}

var  pt   : Plinedesc;
     p,q  : dataref;

begin
   with Curwin^ do
      begin
         p := Curline;
         pt := Edit(Stream,Curline,rd);
         p := pt^.FwdRef;
         repeat
            pt := Edit(Stream,p,rd);
            q := p;
            p := pt^.FwdRef;
         until (p = 0) or ((pt^.Flags and Colored) <> 0);
         if p <> 0 then if q <> 0 then
            begin
               Curline := q;
               EditRealign;
            end;
      end;
end;

overlay procedure EditBatDelete;
{  This command causes all lines in the batch of the current window to
   be deleted.
}
```

```
var  pt  : Plinedesc;
     p,q : dataref;

begin
   with Curwin^ do
      begin
         p := Topline;
         repeat
            pt := Edit(Stream,p,rd);
            q := p;
            p := pt^.BakRef;
         until p = 0;
         repeat
            pt := Edit(Stream,q,rd);
            p := q;
            q := pt^.FwdRef;
            if ((pt^.Flags and Colored) <> 0) then
               begin
                  pt := Edit(Stream,p,rt);
                  EditDelLine(p);
               end;
         until q = 0;
      end;
   EditRealign;
end;

overlay procedure EditBatClear;
{  This command clears the batch bits from all lines in the stream of the
   current window.
}

var   pt   : Plinedesc;
      p,q  : dataref;

begin
   with Curwin^ do
      begin
         p := Topline;
         repeat
            pt := Edit(Stream,p,rd);
            q := p;
            p := pt^.BakRef;
         until p = 0;
         EditColorFile(false,q,0);
      end;
end;
```

```
overlay procedure EditBatMove;
{  This command causes all lines in the batch of the current window to
   be deleted from their current positions and added after the line the
   cursor is on.
}

var  pt,pt1 : Plinedesc;
     p,q : dataref;
     fwd : dataref;
     sv  : dataref;
     cur : dataref;

begin
   with Curwin^ do
      begin
         sv := Curline;
         p := Topline;
         repeat
            pt := Edit(Stream,p,rd);
            q := p;
            p := pt^.BakRef;
         until p = 0;
         repeat
            pt := Edit(Stream,q,rd);
            p := q;
            q := pt^.FwdRef;
            if ((pt^.Flags and Colored) <> 0) then
               begin
                  if not EditInsBuf(pt^.BuffLen) then;
                  pt := Edit(Stream,p,rt);
                  pt1 := Edit(Stream,Curline,wt);
                  move(pt^.Txt^[1],pt1^.Txt^[1],pt^.BuffLen);
                  pt1^.Flags := pt1^.Flags and not(Colored);{don't recurse}
                  cur := Curline;
                  EditDelline(p);
                  Curline := cur;
               end;
         until q = 0;
         Curline := sv;
         EditRealign;
      end;
end;

overlay procedure EditBatCopy;
{  This command causes all lines in the batch of the current window to
   be copied from their current positions and added after the line the
   cursor is on.
}
```

```
var  pt,pt1   : Plinedesc;
     p,q      : dataref;
     sv       : dataref;
     i        : integer;

begin
   with Curwin^ do
      begin
         sv := Curline;
         p := Topline;
         repeat
            pt := Edit(Stream,p,rd);
            q := p;
            p := pt^.BakRef;
         until p = 0;
         repeat
            pt := Edit(Stream,q,rd);
            p := q;
            q := pt^.FwdRef;
            if ((pt^.Flags and Colored) <> 0) then
               begin
                  if not EditInsBuf(pt^.BuffLen) then;
                  pt := Edit(Stream,p,rt);
                  pt1 := Edit(Stream,Curline,wt);
                  move(pt^.Txt^[1],pt1^.Txt^[1],pt^.BuffLen);
                  pt1^.Flags := pt1^.Flags and not(Colored);{don't recurse}
               end;
         until q = 0;
         Curline := sv;
         EditRealign;
      end;
end;

overlay procedure EditBatRead(Swin : Pwindesc);
{  This command causes all lines in the batch of the specified window to
   be copied from their current positions and added after the line the
   cursor is on in the current window.
}

var  pt,pt1   : Plinedesc;
     p,q      : dataref;

begin
   if StreamDef[Curwin^.Stream].RecLen < StreamDef[Swin^.Stream].RecLen
      then exit;
   p := Swin^.Topline;
   repeat
```

```
      pt := Edit(Swin^.Stream,p,rd);
      q := p;
      p := pt^.BakRef;
   until p = 0;
   repeat
      pt := Edit(Swin^.Stream,q,rd);
      p := q;
      q := pt^.FwdRef;
      if ((pt^.Flags and Colored) <> 0) then
         begin
            if not EditInsBuf(pt^.BuffLen) then;
            pt := Edit(Swin^.Stream,p,rt);
            pt1 := Edit(Curwin^.Stream,Curwin^.Curline,wt);
            move(pt^.Txt^[1],pt1^.Txt^[1],pt^.BuffLen);
         end;
   until q = 0;
   EditRealign;
end;

overlay procedure EditBatWrite(Twin : Pwindesc);
{  This command causes all lines in the batch of the current window to
   be copied from their current positions and added after the line the
   cursor is on in the specified window.
}

var  pt,pt1   : Plinedesc;
     p,q      : dataref;
     svcw     : Pwindesc;

begin
   if StreamDef[Twin^.Stream].RecLen < StreamDef[Curwin^.Stream].RecLen
      then exit;
   p := Curwin^.Topline;
   repeat
      pt := Edit(Curwin^.Stream,p,rd);
      q := p;
      p := pt^.BakRef;
   until p = 0;
   repeat
      pt := Edit(Curwin^.Stream,q,rd);
      p := q;
      q := pt^.FwdRef;
      if ((pt^.Flags and Colored) <> 0) then
         begin
            svcw := Curwin;
            Curwin := Twin;
            if not EditInsBuf(pt^.BuffLen) then;
            Curwin := svcw;
```

```
            pt := Edit(Curwin^.Stream,p,rt);
            pt1 := Edit(Twin^.Stream,Twin^.Curline,wt);
            move(pt^.Txt^[1],pt1^.Txt^[1],pt^.BuffLen);
         end;
   until q = 0;
   EditRealign;
end;
```

11. CTRL-O COMMANDS

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     ocmd.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                     under Contract F30602-C-85-0190

Function:  This routines perform the text manipulation for the Ctrl-O
           commands.  All of these routines are overlaid in area 1 and
           therefore cannot call any other routines in area 1.
*****************************************************************************
}

overlay procedure EditChangeCase;
{  This routine checks the character at the current cursor position.
   If it is an alphabetic character, it changes the case from upper
   to lower or vice versa. If the character is not alphabetic, no action
   is performed.
}
var
  Ch : char;
  pt : Plinedesc;

begin {EditChangeCase}
  EditChangeFlag := true;
  pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with Curwin^,pt^ do
    begin
      Ch := Txt^ [Colno];
      if (Ch >= 'A') AND (Ch <= 'Z') then
        Ch := Chr (ord (Ch) + 32)
      else if (Ch >= 'a') and (Ch <= 'z') then
        Ch := Chr (ord (Ch) - 32);
      Txt^ [Colno] := Ch
    end
end; {EditChangeCase}

overlay procedure EditCenterLine;
{  This routine centers all text on the current line as a single entity. }
```

```
var
  r    : integer;
  l    : integer;
  i    : integer;
  Tlen : integer;
  Llen : integer;
  Disp : integer;
  pt   : Plinedesc;

begin {EditCenterLine}

  EditChangeFlag := true;
  pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with Curwin^,pt^ do
    begin
      l := 1;
      while (l < BuffLen) and (Txt^ [l] = ' ') do l := Succ (l);
      r := BuffLen;
      while (r >= 1) and (Txt^ [r] = ' ') do r := Pred (r);
      if r = 0 then Exit;
      Tlen := Succ (r - l);
      Llen := Rmargin;
      Disp := Succ ((Llen - Tlen) div 2);
      if not EditSizeline (Stream,Curline, Succ (Disp + Tlen)) then
        begin
          EditErrormsg (41);
          exit
        end;
      Move (Txt^ [l], Txt^ [1], Tlen);
      Fillchar (Txt^ [Succ (Tlen)], Pred (BuffLen-Tlen), ' ');
      Move (Txt^ [1], Txt^ [Disp], Tlen);
      Fillchar (Txt^ [1], Pred (Disp), ' ')
    end
end; {EditCenterLine}

  overlay procedure EditGotoColumn;
  {  This routine works only on formatted files and advances the cursor
     to the next relational column, adjusting LeftEdge as necessary to
     insure that the entire column is within the window.
  }

  var   i,j    : integer;
        p      : plinedesc;

     procedure SkipFwd;

     var
       c : integer;
```

```
    p : Plinedesc;

  begin
    with Curwin^ do
      begin
        p := Edit(Stream,Curline,rd);
        if p^.BakRef <> 0 then
           begin
              p := Edit(Stream,p^.BakRef,rt);
              c := Colno;
              while (p^.Txt^[c] = ' ') and
                    (c < p^.Bufflen) and
                    (Header^[c] <> chr(TConnect)) do c := succ(c);
              if Header^[c] = chr(TConnect) then c := Colno;
              if c <= StreamDef[Stream].RecLen then Colno := c;
           end;
      end;
  end;


begin  {EditGotoColumn}
   if not Curwin^.Formatted then exit;
   with Curwin^ do
      begin

         {move cursor to first char position of next column}

         i := 0;
         while (Header^[Colno + i] <> chr(TConnect)) and
               (Colno + i < HdrLen)
               do i := succ(i);

         {if this is the end of the relation go back to first column}

         if (Header^[Colno + succ(i)] = EndHdr) or
            (Colno + i >= HdrLen) then
            begin
               Colno := 1;
               LeftEdge := 1;
               exit;
            end;

         {position one past the TConnect}

         Colno := Colno + succ(i);

         {see if the whole column is displayed in the window}
```

```
        i := 0;
        while (Header^[Colno + i] <> chr(TConnect)) and
              (Colno + i < Hdrlen)
              do i := succ(i);
        if Colno + i - pred(LeftEdge) > PhyScrCols then

        {it isn't so we have to scroll the screen to the left}

           begin
              LeftEdge := succ(Colno + i - PhyScrCols);
              EditRealign;
           end;

        {now advance the cursor to the first non-blank position
         of the previous line IF auto-indent mode is set}

        p := Edit(Stream,Curline,rt);
        if (p^.txt^[Colno] = ' ') and (AI) then SkipFwd;

      end;{with Curwin^}
end;  {EditGotoColumn}


overlay procedure EditBackColumn;
{  This routine works only on formatted files and moves the cursor
   to the prior relational column, adjusting LeftEdge as necessary to
   insure that the entire column is within the window.
}

var   i,j    : integer;
      p      : plinedesc;

   procedure SkipFwd;

   var
     c : integer;
     p : Plinedesc;

   begin
     with Curwin^ do
       begin
         p := Edit(Stream,Curline,rd);
         if p^.BakRef <> 0 then
            begin
               p := Edit(Stream,p^.BakRef,rt);
               c := Colno;
               while (p^.Txt^[c] = ' ') and
                     (c < p^.Bufflen) and
```

```
                    (Header^[c] <> chr(TConnect)) do c := succ(c);
                if Header^[c] = chr(TConnect) then c := Colno;
                if c <= StreamDef[Stream].RecLen then Colno := c;
             end;
       end;
   end;


begin  {EditBackColumn}
   if not Curwin^.Formatted then exit;
   with Curwin^ do
      begin

         {find char position of current column delimiter}

         i := 0;
         while (Header^[Colno + i] <> chr(TConnect)) and
               (Colno + i > 1)
               do i := pred(i);

         {skip past column delimiter}

         i := pred(i);

         {now find char position of previous column delimiter}

         while (Header^[Colno + i] <> chr(TConnect)) and
               (Colno + i > 1)
               do i := pred(i);

         {position the cursor on the first character of the column}

         Colno := Colno + succ(i);

         {make sure that the whole column is displayed in the window}

         if Colno < LeftEdge then
            begin
               LeftEdge := pred(Colno);
               EditRealign;
            end;

         {now advance the cursor to the first non-blank position
          of the previous line IF auto-indent mode is set}

         p := Edit(Stream,Curline,rt);
         if (p^.txt^[Colno] = ' ') and (AI) then SkipFwd;
```

```
      end;{with Curwin^}

end;  {EditBackColumn}

overlay function EditWindowCreate (Size : byte;
                                   Win : byte;
                                   Wid : integer;
                                   Fn  : FName) : Pwindesc;

{  This routine creates a window of the specified size, taking the lines
   from the window specified by Win and returns a pointer to it.  The new
   window is attached to Fn.
}
var  i       : integer;
     p, q    : Pwindesc;
     svcurwin: Pwindesc;
     pt      : Plinedesc;

begin {EditWindowCreate}
  EditWindowCreate := nil;
  if Size >= 3 then
    begin
      if Win < 1 then exit;
      p := Window1;
      i := 1;
      while (i < Win) and (p^.Fwdlink <> Window1) do
        begin
          p := p^.Fwdlink;
          i := Succ (i);
        end;
      if i <> Win then
         begin
            EditErrorMsg(42);
            exit;
         end;

      { Make a new window structure }

      q := EditCrewindow (Succ (p^.Lastlineno-Size),
                          Size,
                          Wid,
                          Fn,
                          Line1,
                          Col1);

      if q = nil then exit;
      if p^.Lastlineno-p^.Firstlineno - Size >= 1 then
```

```
        begin                          {If window can be compressed}
          p^.Lastlineno := p^.Lastlineno - Size;

          { we may be positioned outside the window's area now }

          with p^ do
             while Lineno > (Lastlineno - Firstlineno) do
                begin                  { Fix up the pointers}
                   Lineno := Pred (Lineno);
                   pt := Edit(Stream,Curline,rd);
                   Curline := pt^.BakRef;
                end;

          q^.Backlink := p;
          q^.Fwdlink := p^.Fwdlink;
          p^.Fwdlink^.Backlink := q;
          p^.Fwdlink := q;
        end
      else
        begin
          EditErrorMsg(22);
          q^.FwdLink := Winstack;
          Winstack := q;
          EditDelstream(q^.Stream);
          q := nil;
        end;
    end;

    {regenerate all the window numbers}

    p := Window1;
    i := 1;
    Repeat
      p^.WindNo := i;
      i := succ(i);
      p := p^.Fwdlink
    until p = Window1;
    EditWindowCreate := q;
end; {EditWindowCreate}

overlay procedure EditWindowDelete (Wno : byte);
{  This routine deletes the window specified by Wno. }
var  Thechar : char;
     i       : integer;
     p       : Pwindesc;
     w       : Pwindesc;
     Found   : boolean;
     fl      : integer;
```

```
begin {EditWindowDelete}
  if Wno < 1 then
    exit;
  i := 1;
  p := Window1;
  while (i < Wno) and (p^.FwdLink <> Window1) do
  begin
    p := p^.Fwdlink;
    i := Succ (i)
  end;
  if i <> Wno then exit;
  fl := p^.Firstlineno;
  if p <> p^.Fwdlink then
  begin
    if p = Window1 then
    begin                                        {Window below gets the lines}
      Window1 := Window1^.Fwdlink;
      if Curwin = p then
        Curwin := Window1;
      Window1^.Backlink := p^.Backlink;
      p^.Backlink^.Fwdlink := Window1;
      Window1^.Firstlineno := P^.Firstlineno;
      EditRealign                               {Fixes Lineno for all windows}
    end
    else
    begin                                        {Window above gets the lines}
      if Curwin = p then
        Curwin := p^.Backlink;
      p^.Backlink^.Fwdlink := p^.Fwdlink;
      p^.Fwdlink^.Backlink := p^.Backlink;
      p^.Backlink^.Lastlineno := p^.Lastlineno;
      EditRealign;                              {Fixes Lineno for all windows};
    end;

    { Check other windows-- they may point to the same text stream }
    { And, while we're at it, generate new window numbers for all  }
    { the windows since they've changed as a result of the delete. }

    w := Window1;
    i := 1;
    Found := false;
    Repeat
      if w^.Stream = p^.Stream then
        Found := true;
      w^.WindNo := i;
      i := succ(i);
      w := w^.Fwdlink
```

```
    until w = Window1;

    { If no other object references the text stream, it may be deleted }

    if not Found then EditDelStream(p^.Stream);
    if (not(Found)) and (p^.Formatted) then FreeMem(p^.Header,succ(p^.HdrLen));
    p^.Fwdlink := Winstack;                          {Push window onto free list}
    Winstack := p
  end;
  EditUpdrowasm(fl);
end; {EditWindowDelete}
```

12. CTRL-Q COMMANDS

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     qcmd.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                      under Contract F30602-C-85-0190

Function:  These routines perform the text manipulation for the Ctrl-Q
           commands.  All these routines are overlaid in area 1 and
           therefore cannot call any other routines in area 1.
*****************************************************************************
}

overlay procedure EditBeginningLine;
{  This routine positions the cursor to column 1 of the current line }
begin
  Curwin^.ColNo := 1;
end;

overlay procedure EditDeleteTextRight;
{  This routine deletes the character at the current cursor position,
   and all text to the right of it on the same line.
}
var
  i  : integer;
  pt : Plinedesc;

begin {EditDeleteTextRight}
  pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with Curwin^,pt^ do
  begin
    if Colno <= BuffLen then
    begin
      for i := Colno to BuffLen do
        Txt^ [i] := ' ';
      if not EditSizeline (Stream, Curline, Colno) then
      begin
        EditErrormsg (41);
        exit;
```

```
        end
      end
    end
end; {EditDeleteTextRight}

overlay procedure EditWindowDeleteText(Fn : FName);
{  This routine deletes all lines in the text stream of the current
   window, and starts it off wtih a new stream attached to Fn.
   If the stream is not in use by any other window it is deleted before
   the new stream is opened.
}
var
  p             : dataref;
  q             : dataref;
  w             : Pwindesc;
  InUse         : boolean;
  pt            : Plinedesc;
  j             : integer;

begin {EditWindowDeleteText}
  EditChangeFlag := true;
  InUse := false;
  w := Curwin^.Fwdlink;
  while w <> Curwin do
    begin
       if w^.Stream = Curwin^.Stream then InUse := true;
       w := w^.Fwdlink
    end;
  with Curwin^ do
    begin
      if not InUse then EditDelStream(Stream);
      Filename := Fn;
      Insertflag := Insert;
      AI := false;
      Lmargin := 1;
      Rmargin := Logscrcols;
      Clineno := 0;
      Leftedge := 1;
      RepLast := nothing;
      if (Formatted) and not (InUse) then FreeMem(Header,succ(HdrLen));
      Lineno := 1;
      Colno := 1;
      if Fn = Nofile then
         begin
            Formatted := false;
            Header := nil;
            HdrLen := 0;
            Stream := EditNewStream(Fn,DefRecSize);
```

```
            if Stream = 0 then exit;
            Curline := EditMakTxtDes(Stream,16);
         end
      else
         begin
            Stream := EditNewStream(Fn,0);
            if Stream = 0 then exit;
            with StreamDef[Stream] do
               begin
                  pt := Edit(Stream,TofRef,rt);
                  Curline := TofRef;
                  with pt^ do
                     begin
                        j := Bufflen;
                        while Txt^[j] = ' ' do j := pred(j);
                        if Txt^[j] = EndHdr then
                           begin
                              GetMem(Header,succ(Bufflen));
                              HdrLen := Bufflen;
                              for j := 1 to Bufflen do
                                 case Txt^[j] of
                                    '-'  : Header^[j] := chr(Horizontal);
                                    '|'  : Header^[j] := chr(TConnect);
                                    else   Header^[j] := Txt^[j];
                                 end;
                              Formatted := True;
                              EditUserPush(^Z);
                           end;
                     end;
               end;
         end;
       TopLine := Curline;
    end;
end; {EditWindowDeleteText}

overlay procedure EditToggleAutoindent;
{  This routine processes the toggle autoindent mode command, just
   inverting the AI flag in the current window.
}
begin {EditToggleAutoindent}
  with Curwin^ do AI := not AI
end; {EditToggleAutoindent}

overlay procedure EditWindowLink (Wto : byte; Wfrom : byte);
{  Two windows are made to point to the same text stream by assigning
   the same stream identifier to the pointed window, and copying the
   pointers of the window pointed to into the window doing the pointing.
}
```

```
var
  p           : dataref;
  Pto, Pfrom  : Pwindesc;
  Pw          : Pwindesc;
  i           : integer;
  Others      : boolean;

begin {EditWindowLink}
  if Wto < 1 then exit;
  if Wfrom < 1 then exit;

  Pto := window1;
  i := 1;
  while i < Wto do
    begin
      Pto := Pto^.Fwdlink;
      i := Succ (i)
    end;

  Pfrom := window1;
  i := 1;
  while i < Wfrom do
    begin
      Pfrom := Pfrom^.Fwdlink;
      i := Succ (i)
    end;

  { If source = destination, do nothing. }

  if Pfrom^.Stream = Pto^.Stream then exit;

  { Clean out source window's text if no other windows point to it.
    Failure to do this would result in a dangling text stream.
    Doing this when other windows point to the old text stream
    would make them point to the free linedesc list. }

  Others := false;
  Pw := Pfrom^.Fwdlink;
  while Pw <> Pfrom do
    begin
      if Pw^.Stream = Pfrom^.Stream then Others := true;
      Pw := Pw^.Fwdlink
    end;

  if not Others then EditDelStream(Pfrom^.Stream);

  { Match streams }
```

```
    Pfrom^.Stream := Pto^.Stream;

    { Now for the equate }

    for i := 0 to 80 do
      Pfrom^.Filename [i] := Pto^.Filename [i];
    Pfrom^.Topline := Pto^.Topline;
    Pfrom^.Curline := Pto^.Topline;
    Pfrom^.lmargin := Pto^.lmargin;
    Pfrom^.rmargin := Pto^.rmargin;
    Pfrom^.Leftedge := Pto^.Leftedge;
    Pfrom^.Lineno := Pto^.Lineno;
    Pfrom^.Colno := Pto^.Colno;
    if (Pfrom^.Formatted) and not(Others) then
       FreeMem(Pfrom^.Header,succ(Pfrom^.HdrLen));
    Pfrom^.Formatted := Pto^.Formatted;
    Pfrom^.Header := Pto^.Header;
    Pfrom^.HdrLen := Pto^.HdrLen;
    Pfrom^.RepLast := nothing;
    EditRealign;
  end; {EditWindowLink}

¦ overlay procedure EditWindowUp;
¦ {  This routine moves the cursor up one window, with wraparound to
¦    the bottom of the screen.
¦ }
¦ begin {EditWindowUp}
¦   Curwin := Curwin^.Backlink;
¦   Updcurflag := true
¦ end; {EditWindowUp}
¦
¦ overlay procedure EditWindowDown;
¦ {  this routine moves the cursor down one window, with wraparound to
¦    the top window if necessary.
¦ }
¦ begin {EditWindowDown}
¦   Curwin := Curwin^.Fwdlink;
¦   Updcurflag := true                {Update cursor position}
¦ end; {EditWindowDown}

  overlay procedure EditKeyword;
  {  This routine finds a conjunction of keywords in the current text stream.
     Each keyword is found separately and the search succeeds only if all the
     keywords were found on the same line.
  }

  const Delimiter : char = '&';       {delimits qualifier expressions}
```

```
type  target = record
                  ref : dataref;
                  num : integer;
               end;

      colmap = record
                  value: Varstring;
               end;

      targetarray = array[1..MaxCols] of target;
      columnarray = array[1..MaxCols] of colmap;


var
  c,i         : integer;
  p           : dataref;
  s           : Varstring;
  qual        : Varstring;                        {qualifier workstring}
  colset      : columnarray;
  numcols     : 0..MaxCols;                       {number of columns}
  ssi,ksi     : integer;
  targetset   : targetarray;
  SrchLine    : dataref;
  SrchCol     : integer;
  SrchLno     : integer;
  j,k,m,n     : integer;
  SrchSet     : integer;
  Searching   : boolean;
  pt          : Plinedesc;

begin  {EditKeyword}

  {parse the column names and values}

with Curwin^ do
 begin
  if qualstr = '' then exit;
  if Pos(Delimiter,qualstr) <> 0 then
     begin
        qual := '';
        for numcols := 1 to MaxCols do
            colset[numcols].value := '';
        numcols := 1;
        ssi := 1;
        while ssi <= Length(qualstr) do
           begin
              while (qualstr[ssi] <> Delimiter) and
                    (ssi <= Length(qualstr)) do
```

```
            begin
               qual := qual + qualstr[ssi]; {build a qualifier expression}
               ssi := ssi + 1;
            end;
         colset[numcols].value:= qual;
         numcols := numcols + 1;
         if numcols > MaxCols then
            begin
               if not AbortCmd then EditErrorMsg(52);
               exit;
            end;
         ssi := ssi + 1;                    {skip the delimiter}
         qual := '';
      end;
   numcols := numcols - 1;                  {set to last entry in colset}
   for ksi := 1 to numcols do
       if colset[ksi].value= '' then
          begin
             if not AbortCmd then EditErrorMsg(57);
             exit;
          end;
end
else
   begin
      colset[1].value:= qualstr;
      numcols := 1;
   end;

Notfound := false;                  {this is a GLOBAL so we do with it}
                                    {whatever the original code did but}
                                    {we won't use it for anything in this}
                                    {routine}

Searching := true;                  {this is our local control variable}
Replast := Keyword;                 {For Ctrl-L command}
EditUpdphyscr;

{ Parse the option string }

Global := false;
IgnCase := false;
WholeWord := false;

if Pos ('F', Optstr) <> 0 then
  Global := true;
if Pos ('I', Optstr) <> 0 then
  IgnCase := true;
if Pos ('W', Optstr) <> 0 then
```

```
    WholeWord := true;
end;{with Curwin^}

  { Now the search begins }

  with Curwin^ do
    begin
       if Global then
          begin                          { Go to top/bottom of file }
             EditWindowTopFile;
             EditRealign;
          end;

    while Searching do
        begin
           if numcols > 1 then
              for k := 1 to numcols do
                 begin
                    targetset[k].ref := 0;
                    targetset[k].num := 0;
                 end;
           for ksi := 1 to numcols do
              begin
                 SrchLine := Curline;
                 SrchCol := Colno;
                 s := colset[ksi].value;
                 SrchLine := EditScanFwd (SrchLine,
                                          s,
                                          SrchCol,
                                          SrchLno,
                                          IgnCase,
                                          WholeWord);
                 targetset[ksi].ref := SrchLine;
                 targetset[ksi].num := SrchLno;

                 {this is the FIRST occurrence of each key so if we get}
                 {a SrchLine of nil then there are no occurrences so we}
                 {may as well quit now}

                 if SrchLine = 0 then
                    begin
                       NotFound := true;
                       Searching := false;
                       if not AbortCmd then EditErrorMsg(38);
                       Exit;
                    end;

              end;
```

```
p := targetset[1].ref;
if (numcols > 1) and (p <> 0) then
   begin
      SrchSet := targetset[1].num;
      for k := 2 to numcols do
         if SrchSet <> targetset[k].num then SrchSet := -1;
      if not(SrchSet<0) then p := targetset[1].ref else p := 0;
   end;
if p <> 0 then
   begin
      Topline := p;
      Curline := p;
      Colno := SrchCol + Length(s);
      if Colno < 1 then Colno := 1;
      Lineno := 1;
      Notfound := false;
      Searching := false;
   end
else
   begin
      if numcols > 1 then
         begin
            SrchLno := -1;
            for k := 1 to numcols do

            {find the largest lineno for which there is a non-nil}
            {pointer in the targetset - we have to make sure the }
            {pointer is non-nil because the lineno will be set to}
            {the file length if the search failed}

               if (targetset[k].ref <> 0) and
                  (targetset[k].num > SrchLno) then
                  begin
                     SrchLno := targetset[k].num;
                     p := targetset[k].ref;
                  end;

            {if we found one then we set Curline to it }
            {and repeat the process again}

            if not (SrchLno < 0) then
               begin
                  NotFound := false;
                  Searching := true;
                  Topline := p;
                  Curline := p;
                  Colno := 1;
                  Lineno := 1;
```

```
                        EditRealign;
                     end
                  else
                     begin
                        NotFound := true;
                        Searching := false;
                        if not AbortCmd then EditErrormsg (38);
                     end;
               end
            else
               begin
                  Notfound := true;
                  Searching := false;
                  if not AbortCmd then EditErrormsg (38);
               end;{if numkeys > 1}
         end;{if p <> 0}
      end;{while Searching}
   end;{with Curwin^}
end; {EditKeyword}

overlay procedure EditFind;
{  This routine finds a pattern in the current text stream, beginning
   at the current cursor location.
}
var
  c       : integer;
  i       : integer;
  p       : dataref;
  Ntimes  : integer;
  St : Varstring;
  s  : Varstring;
  l  : integer;

begin  {EditFind}
with Curwin^ do
 begin
  if Searchstr = '' then exit;
  s := Searchstr;
  Notfound := false;
  Replast := Find;                    {For Ctrl-L command}
  EditUpdphyscr;

  { Parse the option string }

  Global := false;
  IgnCase := false;
  WholeWord := false;
```

```
  if Pos ('F', Optstr) <> 0 then
    Global := true;
  if Pos ('I', Optstr) <> 0 then
    IgnCase := true;
  if Pos ('W', Optstr) <> 0 then
    WholeWord := true;

  c := 0;
  for i := 1 to Length(Optstr) do
    if (Optstr[i] in ['0'..'9']) and (c = 0) then c := i;

  Ntimes := 1;              {assume finding 1st occurrence}
  if c > 0 then             {numbers in option string, extract them}
    begin
      i := c;
      while (i <= Length (Optstr)) and (Optstr [i] in ['0'..'9']) do
        i := Succ (i);
      St := Copy (Optstr, c, i-c);
      Val (St, Ntimes, i);
      {convert error or ignore request to search zero times}
      if (i <> 0) or (Ntimes = 0) then Ntimes := 1;
    end;
end;{with Curwin^}

  { Search for the pattern }

  with Curwin^ do
    begin
      if Global then
        begin                          { Go to top/bottom of file }
          EditWindowTopFile;
          EditRealign;
        end;

      i := 1;
      while (i <= Ntimes) and not Notfound do
        begin
          i := Succ (i);
          c := Colno;
          p := EditScanFwd (Curline, s, c, 1, IgnCase, WholeWord);
          if p <> 0 then
            begin
              Topline := p;
              Curline := p;
              Colno := c + Length (s);
              if Colno < 1 then
                Colno := 1;
              Lineno := 1;
```

```
              Notfound := false;
              EditRealign
            end
          else
            begin
              Notfound := true;
              if not AbortCmd then
                EditErrormsg (38);      {Pattern not found}
            end
        end
    end
end; {EditFind}

overlay procedure EditReplace;
{  This routine finds a pattern in the current text stream, replacing
   it if a match is found.
}
var
  p        : dataref;
  t        : Textline;
  c        : integer;
  l        : integer;
  r        : integer;
  i        : integer;
  j        : integer;
  Ch       : byte;
  Ntimes   : integer;
  St       : Varstring;
  s        : Varstring;
  Count    : integer;
  replaced: boolean;
  line     : integer;
  pt       : Plinedesc;

  procedure Replacestring;
    var
      S1fwa : integer;
      S2fwa : integer;
      S1len : integer;
      pt    : Plinedesc;
      i     : integer;
      toobig: boolean;

    begin {Replacestring of EditReplace}
      EditChangeFlag := true;
      pt := Edit(Curwin^.Stream,Curwin^.Curline,wt);
      with Curwin^,pt^ do
        begin
```

```
          if l > r then
            begin                {replacement string < find string}
              S1fwa := Colno + l;
              S2fwa := Colno + r;
              S1len := Succ (BuffLen - S1fwa);
              move (Txt^ [S1fwa], Txt^ [S2fwa], S1len);
              fillchar (Txt^ [Succ (BuffLen-l+r)], l-r, ' ')
            end
          else if r > l then
            begin                {replacement string > find string}

              i := BuffLen;
              toobig := false;
              while (i >= BuffLen - (r -1)) and (toobig = false) do
                 begin
                    if Txt^[i] <> ' ' then toobig := true;
                    i := pred(i);
                 end;
              if toobig then
                 if not EditSizeLine(Stream,Curline,(i + 2 + r - 1)) then
                    begin
                       EditErrormsg (41);
                       Global := false;      {Stop searching}
                       exit;
                    end;

              S1fwa := Colno + l;
              S2fwa := Colno + r;
              S1len := Succ (BuffLen - S2fwa);
              move (Txt^ [S1fwa], Txt^ [S2fwa], S1len);
            end;
          move (ReplaceStr [1], Txt^ [Colno], r)
        end
    end; {Replacestring of EditReplace}

begin  {EditReplace}
with Curwin^ do
 begin
  l := Byte (SearchStr[0]);
  if l < 1 then exit;
  s := Searchstr;
  Replast := Replace;                  {For Ctrl-L}
  r := Byte (ReplaceStr[0]);

  NoAsk := false;
  Global := false;
  IgnCase := false;
  WholeWord := false;
```

```
  if Pos ('G', Optstr) <> 0 then
    Global := true;
  if Pos ('I', Optstr) <> 0 then
    IgnCase := true;
  if Pos ('W', Optstr) <> 0 then
    WholeWord := true;
  if Pos ('D', Optstr) <> 0 then
    NoAsk := true;

  c := 1;
  while (c <= Length (Optstr)) and not (Optstr [c] in ['0'..'9']) do
    c := Succ (c);

  if not (Optstr [c] in ['0'..'9']) then
    Ntimes := 1
  else
    begin
      i := c;
      while (i <= Length (Optstr)) and (Optstr [i] in ['0'..'9']) do
        i := Succ (i);
      St := Copy (Optstr, c, i-c);
      Val (St, Ntimes, i);
    end;
end;{with Curwin^}

  { search for the pattern }

  Notfound := true;              {reset if even one occurance is found}
  with Curwin^ do
    begin
      if Global then
        begin                          { Go to top/bottom of file }
          EditWindowTopFile;
          Ntimes := Maxint;        {Don't terminate prematurely}
          EditRealign;
        end;

      Count := 1;                  {Count up to Ntimes}
      repeat                       {until not global will do it at least once}
        Count := Succ (Count);
        Global := (Count <= Ntimes);
        c := Colno;
        p := EditScanFwd (Curline, s, c, line, IgnCase, WholeWord);
        if p <> 0 then
          begin
            Topline := p;
            Curline := p;
```

```
        Colno := c;
        Lineno := 1;
        Notfound := false;     {found at least one occurance of the pattern}

        {optional preview before replacement}

        replaced := false;

        if not NoAsk then
          begin
            EditRealign;
            ch := 0;
            EditUpdphyscr;
            with Curwin^ do
              GotoXY (Colno-Leftedge+1,Firstlineno+Lineno);
            EditUpdphyscr;
            UserReplace(ch);
            if ch = 0 then
              begin
                EditAppcmdnam ('<Replace ? (y,n)>:');
                while (not EditKeypressed) and (not Abortcmd) do
                  begin
                    EditUpdphyscr;
                    with Curwin^ do
                      GotoXY (Colno-Leftedge+1,Firstlineno+Lineno);
                  end;
                if Abortcmd then exit;
                Ch := EditGetinput;
              end;
            if (ch = Ctrlu) or (ch = 27) then exit;
            if UpCase(chr (Ch)) = 'Y' then
            begin
              Replacestring;
              replaced := true;
            end;
            EditUpdphyscr;
          end
        else
          begin                          {replace without asking}
            if not EditKeypressed then
              begin
                EditUpdphyscr;
                with Curwin^ do
                  GotoXY (Colno-Leftedge+1,Firstlineno+Lineno);
              end;
            if Abortcmd then exit;
            Replacestring;
            replaced := true;
```

```
              if not EditKeypressed then EditUpdphyscr;
            end;

          {advance over replace string to prevent left recursive sch/rep}

          pt := Edit(Stream,Curline,rd);
          with pt^ do
          begin
            if replaced then
              Colno  := Colno + r
            else
              Colno  := Colno + 1;
            if (Colno > BuffLen) and (FwdRef = 0) then
            begin
              Global := false;
              Colno := BuffLen   {don't stay off the screen}
            end else
              if Colno > BuffLen then
              begin
                Curline := FwdRef;
                Colno := 1
              end;
          end;
        end
      else
        Global := false
    until not Global;
  end;

  if Notfound then EditErrormsg (38); {Pattern not found}
  Circin := Circout;                  {Zap typeahead}
end;  {EditReplace}
```

13. RELATIONAL OPERATORS

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     relcmd.tt                   Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                      by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  These are all the routines that perform the relational operators.
           They are all overlaid in area 1 and thus cannot call any other
           routines in area 1.
*****************************************************************************
}

overlay procedure EditSelect;
{  This routine finds a conjunction of keywords in the current text stream.
   Each keyword is found separately and the search succeeds only if all the
   keywords were found on the same line and each keyword was found within
   the boundaries of the specified column.
}

const Delimiter : char = '&';          {delimits qualifier expressions}
      RelOp     : char = '=';          {delimits column name from value}
      AddLine   : boolean = true;
      SameLine  : boolean = false;
      MaxHits     = 1;                 {max no. of occurrences of a value that}
                                       {will be looked at during one iteration}
type  target = record
                  ref : dataref;
                  num : integer;
               end;

      colmap = record
                  name : Varstring;
                  value: Varstring;
                  bcol : integer;
                  ecol : integer;
               end;

      targetarray = array[1..MaxCols] of target;
```

```
    columnarray = array[1..MaxCols] of colmap;


var
  c,i         : integer;
  p           : dataref;
  s           : Varstring;
  qual        : Varstring;                    {qualifier workstring}
  colset      : columnarray;
  numcols     : 0..MaxCols;                   {number of columns}
  ssi,ksi     : integer;
  targetset   : targetarray;
  SrchLine    : dataref;
  SrchCol     : integer;
  SrchLno     : integer;
  j,k,m,n     : integer;
  SrchSet     : integer;
  Searching   : boolean;
  EndCol      : char;
  Destwin     : Pwindesc;
  row         : Ptextline;
  rowlen      : integer;
  NullRow     : boolean;
  pt          : Plinedesc;

  procedure ValWin;
  {  This procedure checks that the window specified in DestinStr exists
     and is empty and has no file attached.  If these tests are passed
     then Destwin is set to point to the window otherwise Destwin is set
     to nil.
  }

  var  p     : Pwindesc;
       w     : integer;
       pt    : Plinedesc;

  begin
     Destwin := nil;
     if not(Curwin^.DestinStr[1] in ['1'..'9']) then exit;
     p := Window1;
     w := 1;
     while (p^.FwdLink <> Window1) and
           (chr(w + ord('0')) <> Curwin^.DestinStr[1]) do
         begin
            p := p^.FwdLink;
            w := succ(w);
         end;
     if chr(w + ord('0')) <> Curwin^.DestinStr[1] then exit;
```

```
   pt := Edit(p^.Stream,p^.Curline,rd);
   if pt^.BakRef <> 0 then exit;
   if pt^.FwdRef <> 0 then exit;
   if p^.Filename <> NoFile then exit;
   Destwin := p;
end;

function BldHdr : boolean;
{  This function builds a header for the destination window and returns
   true if there were no errors.
}

var i : integer;
    d : Plinedesc;
    c : Plinedesc;
    w : Pwindesc;

begin
   BldHdr := false;
   if StreamDef[Destwin^.Stream].RecLen < StreamDef[Curwin^.Stream].RecLen
      then exit;
   BldHdr := true;
   GetMem(Destwin^.Header,succ(Curwin^.HdrLen));
   Destwin^.HdrLen := Curwin^.HdrLen;
   for i := 1 to Curwin^.HdrLen do
      Destwin^.Header^[i] := Curwin^.Header^[i];
   Destwin^.Formatted := true;
end;

procedure GetRow(p : dataref;row : Ptextline);
{  This procedure goes into the text line specified by p and
   extracts the values into the buffer specified by row.
}

var  c    : integer;
     pt   : Plinedesc;
     e    : integer;

begin
   Fillchar(row^[1], rowlen, ' ');
   NullRow := true;
   c := 1;
   e := EditPos(EndHdr,Curwin^.Header,Curwin^.HdrLen);
   pt := Edit(Curwin^.Stream,p,rt);
   with pt^ do
      begin
         while (c <= BuffLen) and (c < e) do
            begin
```

```
                row^[c] := Txt^[c];
                if Txt^[c] <> ' ' then NullRow := false;
                c := succ(c);
             end;
      end;
end;

function AddRow(NewLine : boolean; row : Ptextline) : boolean;
{  This procedure adds the contents of the row buffer as a text
   line in the destination window.  If NewLine is true the contents
   are added as a new line otherwise they are added to the current
   line of the destination window.  It returns false on error.
}

var   c          : integer;
      SvCurwin   : Pwindesc;
      pt         : Plinedesc;

begin
   AddRow := true;
   SvCurwin := Curwin;
   Curwin := Destwin;
   with Curwin^ do
      begin
         if NewLine then if not(EditInsbuf(rowlen)) then
            begin
               AddRow := false;
               Curwin := SvCurwin;
               exit;
            end;
         Colno := 1;
         pt := Edit(Stream,Curline,wt);
         with pt^ do for c := 1 to rowlen do
            begin
               if not EditSizeLine(Stream,Curline,succ(Colno)) then
                  begin
                     AddRow := false;
                     Curwin := SvCurwin;
                     exit;
                  end
               else
                  begin
                     Txt^[Colno] := row^[c];
                     Colno := succ(Colno);
                  end;
            end;
      end;
   EditRealign;
```

```
      Curwin := SvCurwin;
   end;

begin  {EditSelect}

  EndCol := chr(TConnect);

  {EditSelect only works with formatted files}

  if not Curwin^.Formatted then
     begin
        if not AbortCmd then EditErrorMsg(61);
        exit;
     end;

  {parse the column names and values}

with Curwin^ do
 begin
  if qualstr = '' then exit;
  if Pos(Delimiter,qualstr) <> 0 then
     begin
        qual := '';
        for numcols := 1 to MaxCols do
           begin
              colset[numcols].name := '';
              colset[numcols].value := '';
              colset[numcols].bcol := 0;
              colset[numcols].ecol := 0;
           end;
        numcols := 1;
        ssi := 1;
        while ssi <= Length(qualstr) do
           begin
              while (qualstr[ssi] <> Delimiter) and
                    (ssi <= Length(qualstr)) do
                 begin
                    qual := qual + qualstr[ssi]; {build a qualifier expression}
                    ssi := ssi + 1;
                 end;
              c := pos(RelOp,qual);              {find the position of relop}
              if c = 0 then
                 begin
                    if not AbortCmd then EditErrorMsg(51);
                    exit;
                 end;
              colset[numcols].name := copy(qual,1,c-1);
              colset[numcols].value:= copy(qual,c+1,Length(qual)-c);
```

```
            numcols := numcols + 1;
            if numcols > MaxCols then
               begin
                  if not AbortCmd then EditErrorMsg(53);
                  exit;
               end;
            ssi := ssi + 1;                    {skip the delimiter}
            qual := '';
         end;
      numcols := numcols - 1;                  {set to last entry in colset}
      for ksi := 1 to numcols do
         begin
            if colset[ksi].name = '' then
               begin
                  if not AbortCmd then EditErrorMsg(54);
                  exit;
               end;
            if colset[ksi].value= '' then
               begin
                  if not AbortCmd then EditErrorMsg(57);
                  exit;
               end;
         end;
   end
else
   begin
      qual := qualstr;
      c := pos(RelOp,qual);              {find the position of relop}
      if c = 0 then
         begin
            if not AbortCmd then EditErrorMsg(51);
            exit;
         end;
      colset[1].name := copy(qual,1,c-1);
      colset[1].value:= copy(qual,c+1,Length(qual)-c);
      numcols := 1;
   end;

Notfound := false;                   {this is a GLOBAL so we do with it}
                                     {whatever the original code did but}
                                     {we won't use it for anything in this}
                                     {routine}

Searching := true;                   {this is our local control variable}
Replast := Select;                   {For Ctrl-L command}
EditUpdphyscr;

{ Parse the option string }
```

```
  Global := false;
  IgnCase := false;
  WholeWord := false;
  AllOccur := false;
  OutPut := false;
  SetScan := false;

  if Pos ('F', Optstr) <> 0 then
    Global := true;
  if Pos ('I', Optstr) <> 0 then
    IgnCase := true;
  if Pos ('W', Optstr) <> 0 then
    WholeWord := true;
  if Pos ('A', Optstr) <> 0 then
    AllOccur := true;
  if Pos ('O', Optstr) <> 0 then
    OutPut := true;
  if Pos ('S',Optstr) <> 0 then
    SetScan := true;
end;{with Curwin^}

  {parse the file header to find the column positions}

  for ksi := 1 to numcols do
     begin
        c := EditPos(colset[ksi].name,Curwin^.Header,Curwin^.HdrLen);
        if c = 0 then
           begin
              if not AbortCmd then EditErrorMsg(54);
              exit;
           end;
        while (c > 0) and (Curwin^.Header^[c] <> EndCol) do
           c := pred(c);
        c := succ(c);
        colset[ksi].bcol := c;
        while (c <= Curwin^.Hdrlen) and
              (Curwin^.Header^[c] <> EndCol) do c := succ(c);
        colset[ksi].ecol := pred(c);
     end;

  {if we're doing OutPut and this is the FirsTime, we have to validate the  }
  {the destination window, build the header and put it in the window struct }
  {and as the first line of the window, initialize the window and line ptrs }
  {in our curiwn structure, and then turn off FirsTime so we don't do this  }
  {again by mistake.}

  if (Curwin^.OutPut) and (Curwin^.FirsTime) then
```

```
   begin

      {validate the window}

      ValWin;
      if Destwin = nil then
         begin
            EditErrorMsg(44);
            exit;
         end;

      {build the header}

      if not(BldHdr) then
         begin
            EditErrorMsg(35);
            exit;
         end;

      {add the header definition record to the file}

      GetMem(row,succ(Destwin^.HdrLen));
      rowlen := Destwin^.HdrLen;
      for i := 1 to Destwin^.HdrLen do
         begin
            case ord(Destwin^.Header^[i]) of
               Horizontal : row^[i] := '-';
               TConnect   : row^[i] := '|';
               else         row^[i] := Destwin^.Header^[i];
            end;
         end;
      row^[i] := EndHdr;
      if not(Addrow(SameLine,row)) then;
      Curwin^.FirsTime := false;
      Curwin^.Deswin := Destwin;
      FreeMem(row,succ(rowlen));
   end;

{ Now the search begins }

with Curwin^ do
  begin
     if Global then
        begin
           EditWindowTopFile;
           EditRealign;
        end;
```

```
while Searching do
    begin
       if numcols > 1 then
          for k := 1 to numcols do
             begin
                targetset[k].ref := 0;
                targetset[k].num := 0;
             end;
       for ksi := 1 to numcols do
          begin
             SrchLine := Curline;
             SrchCol := Colno;
             s := colset[ksi].value;
             repeat
                SrchLine := EditScanFwd (SrchLine,
                                         s,
                                         SrchCol,
                                         SrchLno,
                                         IgnCase,
                                         WholeWord);
                targetset[ksi].ref := SrchLine;
                targetset[ksi].num := targetset[ksi].num + SrchLno;
                SrchCol:=succ(SrchCol);


             until ((SrchCol >= colset[ksi].bcol) and
                   (SrchCol <= colset[ksi].ecol)) or
                   (SrchLine = 0);


             {this is the FIRST occurrence of each key so if we get}
             {a SrchLine of nil then there are no occurrences so we}
             {may as well quit now}

             if SrchLine = 0 then
                begin
                   NotFound := true;
                   Searching := false;
                   if AllOccur then
                      begin
                         EditWindowTopFile;
                         EditRealign;
                         Exit;
                      end
                   else
                      begin
                         if not AbortCmd then EditErrorMsg(38);
                         Exit;
```

```
                end;
            end;

    end;
p := targetset[1].ref;
if (numcols > 1) and (p <> 0) then
    begin
        SrchSet := targetset[1].num;
        for k := 2 to numcols do
            if SrchSet <> targetset[k].num then SrchSet := -1;
        if not(SrchSet<0) then p := targetset[1].ref else p := 0;
    end;
if p <> 0 then
    begin
        Topline := p;
        Curline := p;
        Colno := SrchCol;
        if Colno < 1 then Colno := 1;
        Lineno := 1;
        Notfound := false;
        Searching := false;
        if OutPut then
            begin
                Destwin := Deswin;
                pt := Edit(Stream,Curline,rd);
                rowlen := pt^.Bufflen;
                GetMem(row,succ(rowlen));
                GetRow(Curline,row);
                if not NullRow then
                    if not(AddRow(AddLine,row)) then;
                FreeMem(row,succ(rowlen));
            end;
        if SetScan then EditColorLine(true);
        if AllOccur then
            begin
                EditPushtbf(ord(^L));
                EditPushtbf(WindNo + ord('0'));
                EditPushtbf(ord(^O));
            end;
        EditRealign;
    end
else
    begin
        if numcols > 1 then
            begin
                SrchLno := -1;
                for k := 1 to numcols do
```

```
      {find the largest lineno for which there is a non-nil}
      {pointer in the targetset - we have to make sure the }
      {pointer is non-nil because the lineno will be set to}
      {the file length if the search failed}

         if (targetset[k].ref <> 0) and
            (targetset[k].num > SrchLno) then
            begin
               SrchLno := targetset[k].num;
               p := targetset[k].ref;
            end;

      {if we found one then we set Curline to it }
      {and repeat the process again}

      if not (SrchLno < 0) then
         begin
            NotFound := false;
            Searching := true;
            Topline := p;
            Curline := p;
            Colno := 1;
            Lineno := 1;
            EditRealign;
         end
      else
         begin
            NotFound := true;
            Searching := false;
            if AllOccur then
               begin
                  EditWindowTopFile;
                  EditRealign;
               end
            else
               begin
                  if not AbortCmd then EditErrormsg (38);
               end;
         end;
   end
else
   begin
      Notfound := true;
      Searching := false;
      if AllOccur then
         begin
            EditWindowTopFile;
            EditRealign;
```

```
                        end
                     else
                        begin
                           if not AbortCmd then EditErrormsg (38);
                        end;
                  end;{if numkeys > 1}
            end;{if p <> nil}
         end;{while Searching}
      end;{with Curwin^}
end; {EditSelect}

overlay procedure EditProject;

{  This procedure performs the relational project operator.  It operates
   on Curwin which must be formatted and puts the results in the window
   specified in DestinStr of Curwin.
}

const
      AddLine : boolean = true;
      SameLine: boolean = false;

type  maprec = record
                  cb : integer;
                  ce : integer;
               end;

var   p           : dataref;          {points to line in curwin}
      Destwin     : Pwindesc;         {pointer to destination window}
      Row         : Ptextline;        {buffer in which new row is built}
      rowlen      : integer;
      numcols     : integer;          {number of columns in projection}
      i           : integer;
      NullRow     : boolean;
      pt          : Plinedesc;

      map         : array[1..MaxCols] of maprec;
                                      {for each column being projected, this}
                                      {array holds the beginning and ending}
                                      {txtcols for curwin and destwin}

   procedure ValWin;
   {  This procedure checks that the window specified in DestinStr exists
      and is empty and has no file attached.  If these tests are passed
      then Destwin is set to point to the window otherwise Destwin is set
      to nil.
   }
```

```
var  p     : Pwindesc;
     w     : integer;
     pt    : Plinedesc;

begin
   Destwin := nil;
   if not(Curwin^.DestinStr[1] in ['1'..'9']) then exit;
   p := Window1;
   w := 1;
   while (p^.FwdLink <> Window1) and
         (chr(w + ord('0')) <> Curwin^.DestinStr[1]) do
       begin
          p := p^.FwdLink;
          w := succ(w);
       end;
   if chr(w + ord('0')) <> Curwin^.DestinStr[1] then exit;
   pt := Edit(p^.Stream,p^.Curline,rd);
   if pt^.BakRef <> 0 then exit;
   if pt^.FwdRef <> 0 then exit;
   if p^.Filename <> NoFile then exit;
   Destwin := p;
end;

function BldHdr : boolean;
{  This procedure extracts data from the curwin header for the columns
   specified in QualStr and creates a new header from it and puts the
   new header in the Destwin structure.  While it is parsing the curwin
   header and building the destwin header it also builds the array of
   txtcol numbers (begin, end) for use by GetRow and AddRow.  Any errors
   cause the function to return false.
}

const  sep    : char = ',';

var    name   : Varstring;
       n      : integer;
       hdr    : Ptextline;
       h      : integer;
       c      : integer;
       q      : integer;


begin
   BldHdr := true;
   q := 1;
   numcols := 1;
   GetMem(hdr,succ(Curwin^.HdrLen));
   h := 1;
```

```
   with Curwin^ do
      begin
         while q <= Length(QualStr) do
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldHdr := false;
                     FreeMem(hdr,succ(HdrLen));
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1) do c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               n := map[numcols].cb;
               while n <= c do
                  begin
                     hdr^[h] := Header^[n];
                     n := succ(n);
                     h := succ(h);
                  end;
               numcols := succ(numcols);
               q := succ(q);
            end;
         numcols := pred(numcols);
         hdr^[h] := EndHdr;
      end;
   if StreamDef[Destwin^.Stream].RecLen < succ(h) then
      begin
         BldHdr := false;
         exit;
      end;
   GetMem(Destwin^.Header,succ(h));
   Move(hdr^[1],Destwin^.Header^[1],h);
   FreeMem(hdr,succ(Curwin^.HdrLen));
   Destwin^.HdrLen := h;
   Destwin^.Formatted := true;
end;
```

```
procedure GetRow(p : dataref;row : Ptextline);
{  This procedure goes into the text line of the current window and
   extracts the values of the specified columns into the row buffer.
}

var  c    : integer;
     valu : Varstring;
     col  : integer;
     r    : integer;
     pt   : Plinedesc;

begin
   NullRow := true;
   r := 1;
   pt := Edit(Curwin^.Stream,p,rt);
   with pt^ do
      begin
         for col := 1 to numcols do
            begin
               valu := '';
               for c := map[col].cb to map[col].ce do
                  if c <= BuffLen then
                     begin
                        valu := valu + Txt^[c];
                        if Txt^[c] <> ' ' then NullRow := false;
                     end;
               valu := valu + ' ';
               for c := 1 to Length(valu) do
                  begin
                     row^[r] := valu[c];
                     r := succ(r);
                  end;
            end;
      end;
end;

function AddRow(NewLine : boolean; row : Ptextline) : boolean;
{  This procedure adds the contents of the row buffer as a new text
   line in the destination window.  It returns false on error.
}

var   c          : integer;
      SvCurwin   : Pwindesc;
      pt         : Plinedesc;

begin
   AddRow := true;
   SvCurwin := Curwin;
```

```
      Curwin := Destwin;
      with Curwin^ do
         begin
            if NewLine then if not(EditInsbuf(rowlen)) then
               begin
                  AddRow := false;
                  Curwin := SvCurwin;
                  exit;
               end;
            Colno := 1;
            pt := Edit(Stream,Curline,wt);
            with pt^ do for c := 1 to rowlen do
               begin
                  if not EditSizeLine(Stream,Curline,succ(Colno)) then
                     begin
                        AddRow := false;
                        Curwin := SvCurwin;
                        exit;
                     end
                  else
                     begin
                        Txt^[Colno] := row^[c];
                        Colno := succ(Colno);
                     end;
               end;
         end;
      EditRealign;
      Curwin := SvCurwin;
   end;

begin{EditProject}
   if not(Curwin^.Formatted) then
      begin
         EditErrorMsg(61);
         exit;
      end;
   ValWin;
   if Destwin = nil then
      begin
         EditErrorMsg(44);
         exit;
      end;
   if not(BldHdr) then
      begin
         EditErrorMsg(35);
         exit;
      end;
   GetMem(row,succ(Destwin^.Hdrlen));
```

```
   rowlen := Destwin^.Hdrlen;
   for i := 1 to Destwin^.Hdrlen do
      begin
         case ord(Destwin^.Header^[i]) of
            Horizontal : row^[i] := '-';
            TConnect   : row^[i] := '|';
            else         row^[i] := Destwin^.Header^[i];
         end;
      end;
   row^[i] := EndHdr;
   if not(Addrow(SameLine,row)) then;
   Curwin^.Replast := Project;
   pt := Edit(Curwin^.Stream,Curwin^.Topline,rd);
   while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,rd);
   p := pt^.FwdRef;
   repeat
      If AbortCmd then
         begin
            FreeMem(row,succ(rowlen));
            exit;                  {if EditCpAbort told us to quit}
         end;
      FillChar(row^[1],rowlen,' ');
      GetRow(p,row);
      if not NullRow then
         if not(AddRow(AddLine,row)) then;
      pt := Edit(Curwin^.Stream,p,rd);
      p := pt^.FwdRef;
   until p = 0;
   FreeMem(row,succ(rowlen));
   Curwin := Destwin;
   EditWindowTopFile;
   EditRealign;
end;{EditProject}

overlay procedure EditJoin;
{  This routine finds the first(next,all) tuple in Curwin that matches
   Curline in a specified window according to a specified criteria.
}

const Delimiter : char = '&';       {delimits qualifier expressions}
      RelOp     : char = '=';       {delimits column name from value}
      WinPrefix : char = '.';       {used to separate window.column}
      AddLine   : boolean = true;
      SameLine  : boolean = false;
      MaxHits      = 1;             {max no. of occurrences of a value that}
                                    {will be looked at during one iteration}
type  target = record
                  ref : dataref;
```

```
                num : integer;
              end;

    colmap = record
                swin : byte;       {source window number}
                sname: Varstring;  {source column name}
                twin : byte;       {target window number}
                tname: Varstring;  {target column name}
                tvalue:Varstring;  {value from curline.swin.sname}
                sbcol: integer;    {beginning column in source window}
                secol: integer;    {ending column in source window}
                tbcol: integer;    {beginning column in target window}
                tecol: integer;    {ending column in target window}
              end;

    targetarray = array[1..MaxCols] of target;
    columnarray = array[1..MaxCols] of colmap;


var
  c,i         : integer;
  p           : dataref;
  s           : Varstring;
  qual        : Varstring;                    {qualifier workstring}
  hdr         : Ptextline;                    {file header workstring}
  sw          : Pwindesc;                     {points to source window}
  colset      : columnarray;
  numcols     : 0..MaxCols;                   {number of columns}
  ssi,ksi     : integer;
  targetset   : targetarray;
  SrchLine    : dataref;
  SrchCol     : integer;
  SrchLno     : integer;
  j,k,m,n     : integer;
  SrchSet     : integer;
  Searching   : boolean;
  ch          : byte;
  EndCol      : char;
  Destwin     : Pwindesc;
  row         : Ptextline;
  rowlen      : integer;
  srln        : integer;
  NullRow     : boolean;
  pt          : Plinedesc;

   procedure ValWin;
   {  This procedure checks that the window specified in DestinStr exists
      and is empty and has no file attached.  If these tests are passed
```

```
   then Destwin is set to point to the window otherwise Destwin is set
   to nil.
}

var  p     : Pwindesc;
     w     : integer;
     pt    : Plinedesc;

begin
   Destwin := nil;
   if not(Curwin^.DestinStr[1] in ['1'..'9']) then exit;
   p := Window1;
   w := 1;
   while (p^.FwdLink <> Window1) and
         (chr(w + ord('0')) <> Curwin^.DestinStr[1]) do
       begin
          p := p^.FwdLink;
          w := succ(w);
       end;
   if chr(w + ord('0')) <> Curwin^.DestinStr[1] then exit;
   pt := Edit(p^.Stream,p^.Curline,rd);
   if pt^.BakRef <> 0 then exit;
   if pt^.FwdRef <> 0 then exit;
   if p^.Filename <> NoFile then exit;
   Destwin := p;
end;

function BldHdr : boolean;
{  This procedure builds a header for the destination window.  This is the
   concatenation of the source window header and the current window header.
   It returns true unless there was an error.
}

var i,j  : integer;

begin
   BldHdr := true;
   i := Curwin^.Hdrlen + sw^.Hdrlen;
   i := pred(((i + 16) div 16) * 16);
   if i > MaxDataRecSize then
      begin
         BldHdr := false;
         exit;
      end;
   if i > StreamDef[Destwin^.Stream].RecLen then with Destwin^ do
      begin
         EditDelStream(Stream);
         Stream := EditNewStream(Nofile,i);
```

```
         if Stream = 0 then
            begin
               BldHdr := false;
               Stream := EditNewStream(Nofile,DefRecSize);
               exit;
            end;
         Curline := EditMakTxtDes(Stream,16);
         Topline := Curline;
      end;
   GetMem(Destwin^.Header,succ(i));
   Destwin^.Hdrlen := i;
   i := 1;
   while sw^.Header^[i] <> EndHdr do
      begin
         Destwin^.Header^[i] := sw^.Header^[i];
         i := succ(i);
      end;
   Curwin^.Deslin := pred(i);
   j := 1;
   while Curwin^.Header^[j] <> EndHdr do
      begin
         Destwin^.Header^[i] := Curwin^.Header^[j];
         i := succ(i);
         j := succ(j);
      end;
   Destwin^.Header^[i] := EndHdr;
   Destwin^.Formatted := true;
end;

procedure GetRow(sp,cp : dataref;row : Ptextline);
{  This procedure goes into the text lines of the source window and
   and the current window and concatenates them and puts the result
   in the row buffer.
}

var  c,r    : integer;
     pt     : Plinedesc;

begin
   NullRow := true;
   FillChar(row^[1],rowlen,' ');
   c := 1;
   r := 1;
   pt := Edit(sw^.Stream,sp,rt);
   with pt^ do
      begin
         while c <= BuffLen do
            begin
```

```
                row^[r] := Txt^[c];
                if Txt^[c] <> ' ' then NullRow := false;
                c := succ(c);
                r := succ(r);
             end;
      end;
   while r <= Curwin^.Deslin do
      begin
         row^[r] := ' ';
         r := succ(r);
      end;
   c := 1;
   pt := Edit(Curwin^.Stream,cp,rt);
   with pt^ do
      begin
         while c <= BuffLen do
            begin
               row^[r] := Txt^[c];
               if Txt^[c] <> ' ' then NullRow := false;
               c := succ(c);
               r := succ(r);
            end;
      end;
end;

function AddRow(NewLine : boolean; row : Ptextline) : boolean;
{  This procedure adds the contents of the row buffer as a new text
   line in the destination window.  It returns false on error.
}

var   c          : integer;
      SvCurwin   : Pwindesc;
      pt         : Plinedesc;

begin
   AddRow := true;
   SvCurwin := Curwin;
   Curwin := Destwin;
   with Curwin^ do
      begin
         if NewLine then if not(EditInsBuf(rowlen)) then
            begin
               AddRow := false;
               Curwin := SvCurwin;
               exit;
            end;
         Colno := 1;
         pt := Edit(Stream,Curline,wt);
```

```
            with pt^ do for c := 1 to rowlen do
               begin
                  if not EditSizeLine(Stream,Curline,succ(Colno)) then
                     begin
                        AddRow := false;
                        Curwin := SvCurwin;
                        exit;
                     end
                  else
                     begin
                        Txt^[Colno] := row^[c];
                        Colno := succ(Colno);
                     end;
               end;
         end;
      EditRealign;
      Curwin := SvCurwin;
   end;


begin  {EditJoin}

  EndCol := chr(TConnect);

  {EditJoin only works with formatted files}

  if not Curwin^.Formatted then
     begin
        if not AbortCmd then EditErrorMsg(61);
        exit;
     end;

  {parse the windows and column names}

with Curwin^ do
 begin
  if qualstr = '' then exit;
  qual := '';
  for numcols := 1 to MaxCols do
     begin
        colset[numcols].swin := ord('0');
        colset[numcols].sname := '';
        colset[numcols].twin := ord('0');
        colset[numcols].tname := '';
        colset[numcols].tvalue := '';
        colset[numcols].sbcol := 0;
        colset[numcols].secol := 0;
        colset[numcols].tbcol := 0;
```

```
      colset[numcols].tecol := 0;
   end;
numcols := 1;
ssi := 1;
while ssi <= Length(qualstr) do
   begin
      while (qualstr[ssi] <> Delimiter) and
            (ssi <= Length(qualstr)) do
          begin
             qual := qual + qualstr[ssi]; {build a qualifier expression}
             ssi := ssi + 1;
          end;
      c := pos(RelOp,qual);              {find the position of relop}
      if c = 0 then
         begin
            if not AbortCmd then EditErrorMsg(51);
            exit;
         end;
      colset[numcols].swin := ord(qual[1]);
      colset[numcols].sname := copy(qual,3,c-3);
      colset[numcols].twin := ord(qual[c + 1]);
      colset[numcols].tname:= copy(qual,c+3,Length(qual)-c-2);
      numcols := numcols + 1;
      if numcols > MaxCols then
         begin
            if not AbortCmd then EditErrorMsg(53);
            exit;
         end;
      ssi := ssi + 1;                    {skip the delimiter}
      qual := '';
   end;
numcols := numcols - 1;               {set to last entry in colset}
for ksi := 1 to numcols do            {check for parse errors}
   begin
      if colset[ksi].sname = '' then
         begin
            if not AbortCmd then EditErrorMsg(55);
            exit;
         end;
      if colset[ksi].tname= '' then
         begin
            if not AbortCmd then EditErrorMsg(54);
            exit;
         end;
      if not (chr(colset[ksi].swin) in ['1'..'9']) then
         begin
            if not AbortCmd then EditErrorMsg(56);
            exit;
```

```
            end;
         if not (chr(colset[ksi].twin) in ['1'..'9']) then
            begin
               if not AbortCmd then EditErrorMsg(63);
               exit;
            end;
      end;

   Notfound := false;                  {this is a GLOBAL so we do with it}
                                       {whatever the original code did but}
                                       {we won't use it for anything in this}
                                       {routine}

   Searching := true;                  {this is our local control variable}
   Replast := Join;                    {For Ctrl-L command}
   EditUpdphyscr;

   { Parse the option string }

   Global := false;
   IgnCase := false;
   WholeWord := false;
   AllOccur := false;
   OutPut := false;
   SetScan := false;

   if Pos ('F', Optstr) <> 0 then
     Global := true;
   if Pos ('I', Optstr) <> 0 then
     IgnCase := true;
   if Pos ('W', Optstr) <> 0 then
     WholeWord := true;
   if Pos ('A', Optstr) <> 0 then
     AllOccur := true;
   if Pos ('O', Optstr) <> 0 then
     OutPut := true;
   if Pos ('S',Optstr) <> 0 then
     SetScan := true;
  end;{with Curwin^}

   {parse the file header of Curwin to find the column positions in target win}

   for ksi := 1 to numcols do
      begin
         c := EditPos(colset[ksi].tname,Curwin^.Header,Curwin^.HdrLen);
         if c = 0 then
            begin
               if not AbortCmd then EditErrorMsg(54);
```

```
            exit;
         end;
      while (c > 0) and
            (Curwin^.Header^[c] <> EndCol) do
            c := pred(c);
      c := succ(c);
      colset[ksi].tbcol := c;
      while (c <= Curwin^.Hdrlen) and
            (Curwin^.Header^[c] <> EndCol) do
            c := succ(c);
      colset[ksi].tecol := pred(c);
   end;

{parse the headers of each source window to find the column positions and}
{while were there, go into curline of each window and get the value in that}
{column and stuff it in colset[].tvalue}

for ksi := 1 to numcols do
   begin

      {find the source window pointer}

      sw := Window1;
      i := ord('1');
      while i <> colset[ksi].swin do
         begin
            sw := sw^.Fwdlink;
            i := succ(i);
            if sw = Window1 then
               begin
                  if not AbortCmd then EditErrorMsg(56);
                  exit;
               end;
         end;

      {sw now points to the source window}

      if not sw^.Formatted then
         begin
            if not AbortCmd then EditErrorMsg(62);
            exit;
         end;
      c := EditPos(colset[ksi].sname,sw^.Header,sw^.Hdrlen);
      if c = 0 then
         begin
            if not AbortCmd then EditErrorMsg(55);
            exit;
         end;
```

```
      while (c > 0) and (sw^.Header^[c] <> EndCol) do c := pred(c);
      c := succ(c);
      colset[ksi].sbcol := c;
      while (c <= sw^.Hdrlen) and
            (sw^.Header^[c] <> EndCol)
            do c := succ(c);
      colset[ksi].secol := pred(c);

      {sbcol,secol now contain the column positions for Curline in sw}

      colset[ksi].tvalue := '';
      i := colset[ksi].sbcol;
      j := colset[ksi].secol;
      pt := Edit(sw^.Stream,sw^.Curline,rt);
      with pt^ do
         begin

      {we have to check if this is the last column because not all lines}
      {are as long as the header and anything beyond BuffLen of Curline }
      {is garbage or another line}

            if j > Bufflen then j := Bufflen;

            while Txt^[i] = ' ' do i := succ(i);{strip leading blanks}
            while Txt^[j] = ' ' do j := pred(j);{strip trailing blanks}
            while i <= j do
               begin
                  colset[ksi].tvalue := colset[ksi].tvalue + Txt^[i];
                  i := succ(i);
               end;
         end;

      {colset.tvalue now has the attribute value from the source window}

      if colset[ksi].tvalue = '' then
         begin
            if not AbortCmd then EditErrorMsg(58);
            exit;
         end;
   end;

{if we're doing OutPut and this is the FirsTime, we have to validate the  }
{destination window, build the header, and put it in the window structure }
{and as the first line of the window, initialize the window and line ptrs }
{in our curwin structure, and then turn off FirsTime so we don't do this  }
{again by mistake.}

if (Curwin^.OutPut) and (Curwin^.FirsTime) then
```

```
begin

   {validate the window}

   ValWin;
   if Destwin = nil then
      begin
         EditErrorMsg(44);
         exit;
      end;

   {build the header}

   if not(BldHdr) then
      begin
         EditErrorMsg(35);
         exit;
      end;

   {add the header definition record to the file}

   GetMem(row,succ(Destwin^.Hdrlen));
   rowlen := Destwin^.Hdrlen;
   i := 1;
   while Destwin^.Header^[i] <> EndHdr do
      begin
         case ord(Destwin^.Header^[i]) of
            Horizontal : row^[i] := '-';
            TConnect   : row^[i] := '|';
            else         row^[i] := Destwin^.Header^[i];
         end;
         i := succ(i);
      end;
   row^[i] := EndHdr;
   if not(Addrow(SameLine,row)) then;
   FreeMem(row,succ(rowlen));
   Curwin^.FirsTime := false;
   Curwin^.Deswin := Destwin;
end;

{ Now the search begins.  From here on out the process is identical to }
{ EditSelect except where we optionally write the result to another window}
{ or a file because we have to concat the source window columns.  The reason}
{ we don't just call EditSelect is that we would have to de-parse colset to}
{ create qualstr and this would make Ctrl-L go to EditSelect instead of}
{ EditJoin.  Besides, this is overlay code so it doesn't cost us any space.}
{ (In fact, we couldn't call EditSelect because it is in this overlay area.)}
```

```
with Curwin^ do
  begin
     if Global then
        begin                          { Go to top/bottom of file }
           EditWindowTopFile;
           EditRealign;
        end;

  while Searching do
      begin
         if numcols > 1 then
            for k := 1 to numcols do
               begin
                  targetset[k].ref := 0;
                  targetset[k].num := 0;
               end;
         for ksi := 1 to numcols do
            begin
               SrchLine := Curline;
               SrchCol  := Colno;
               s := colset[ksi].tvalue;
               repeat
                  SrchLine := EditScanFwd (SrchLine,
                                           s,
                                           SrchCol,
                                           SrchLno,
                                           IgnCase,
                                           WholeWord);
                  targetset[ksi].ref := SrchLine;
                  targetset[ksi].num := targetset[ksi].num + SrchLno;
                  SrchCol:=succ(SrchCol);


               until ((SrchCol >= colset[ksi].tbcol) and
                     (SrchCol <= colset[ksi].tecol)) or
                     (SrchLine = 0);


               {this is the FIRST occurrence of each key so if we get}
               {a SrchLine of nil then there are no occurrences so we}
               {may as well quit now}

               if SrchLine = 0 then
                  begin
                     NotFound := true;
                     Searching := false;
                     if AllOccur then
                        begin
```

```
                    EditWindowTopFile;
                    EditRealign;
                    Exit;
                 end
              else
                 begin
                    if not AbortCmd then EditErrorMsg(38);
                    Exit;
                 end;
           end;

     end;
p := targetset[1].ref;
if (numcols > 1) and (p <> 0) then
     begin
        SrchSet := targetset[1].num;
        for k := 2 to numcols do
           if SrchSet <> targetset[k].num then SrchSet := -1;
        if not(SrchSet<0) then p := targetset[1].ref else p := 0;
     end;
if p <> 0 then
     begin
        Topline := p;
        Curline := p;
        Colno := SrchCol;
        if Colno < 1 then Colno := 1;
        Lineno := 1;
        Notfound := false;
        Searching := false;
        if OutPut then
           begin
              Destwin := Deswin;
              GetMem(row,succ(Destwin^.Hdrlen));
              rowlen := Destwin^.Hdrlen;
              GetRow(sw^.Curline,Curline,row);
              if not NullRow then
                 if not (AddRow(AddLine,row)) then;
              FreeMem(row,succ(rowlen));
           end;
        if SetScan then EditColorLine(true);
        if AllOccur then
           begin
              EditPushtbf(ord(^L));
              EditPushtbf(WindNo + ord('0'));
              EditPushtbf(ord(^O));
           end;
        EditRealign;
     end
```

```
else
   begin
      if numcols > 1 then
         begin
            SrchLno := -1;
            for k := 1 to numcols do

            {find the largest lineno for which there is a non-nil}
            {pointer in the targetset - we have to make sure the }
            {pointer is non-nil because the lineno will be set to}
            {the file length if the search failed}

               if (targetset[k].ref <> 0) and
                  (targetset[k].num > SrchLno) then
                  begin
                     SrchLno := targetset[k].num;
                     p := targetset[k].ref;
                  end;

            {if we found one then we set Curline to it }
            {and repeat the process again}

            if not (SrchLno < 0) then
               begin
                  NotFound := false;
                  Searching := true;
                  Topline := p;
                  Curline := p;
                  Colno := 1;
                  Lineno := 1;
                  EditRealign;
               end
            else
               begin
                  NotFound := true;
                  Searching := false;
                  if AllOccur then
                     begin
                        EditWindowTopFile;
                        EditRealign;
                     end
                  else
                     begin
                        if not AbortCmd then EditErrormsg (38);
                     end;
               end;
         end
      else
```

```
                    begin
                       Notfound := true;
                       Searching := false;
                       if AllOccur then
                          begin
                             EditWindowTopFile;
                             EditRealign;
                          end
                       else
                          begin
                             if not AbortCmd then EditErrormsg (38);
                          end;
                    end;{if numkeys > 1}
                end;{if p <> nil}
            end;{while Searching}
        end;{with Curwin^}
end; {EditJoin}

overlay procedure EditQueryClear;
{  This command clears the active query specification if any in the current
   window.
}

begin
   with Curwin^ do RepLast := nothing;
end;
```

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     sortcmd.tt                  Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                      by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  These routines implement sorting and removing duplicate lines.
*****************************************************************************
}

overlay procedure EditOrder;
{  This procedure sorts the rows of the table according to the contents of
   the specified columns, in the major..minor order in which they are given.
}

type  maprec = record
                 cb : integer;
                 ce : integer;
               end;

const

      Ixn : String[14] = '{Sort}tt.sys';

var   map          : array[1..MaxCols] of maprec;
                                    {for each column being sorted, this}
                                    {array holds the beginning and ending}
                                    {txtcols}
      done         : boolean;
      numcols      : 1..MaxCols;
      Ascending    : boolean;
      p            : dataref;
      pt           : Plinedesc;
      SortKey      : VarString;
      IxRef        : dataref;
      Ixf          : IndexFile;
      IxFilVar     : file;
      i            : integer;
```

```
function BldMap : boolean;

{  This procedure extracts data from the curwin header for the columns
   specified in QualStr and builds the map of column positions.
}

const  sep    : char = ',';

var    name   : Varstring;
       c      : integer;
       q      : integer;


begin
   BldMap := true;
   q := 1;
   numcols := 1;
   with Curwin^ do
      begin
         while q <= Length(QualStr) do
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldMap := false;
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1)
                     do c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               numcols := succ(numcols);
               q := succ(q);
            end;
         numcols := pred(numcols);
      end;
end;
```

```
function Pass1(pl : dataref) : boolean;

{  This function passes the descriptor file starting at pl, finds the
   corresponding text, extracts the key according to the column map, and
   adds the key to a Toolbox index.
}

var plin1,plin2 : dataref;
    pt          : Plinedesc;
    col         : integer;
    ib,ie       : integer;
    num         : string[4];
    EndPass     : boolean;

begin
   Pass1 := false;
   if pl = 0 then exit;
   pt := Edit(Curwin^.Stream,pl,rt);
   plin2 := pt^.FwdRef;
   if plin2 = 0 then exit;
   EndPass := false;
   repeat
      if Plin2 = 0 then EndPass := true;
      SortKey := '';
      for col := 1 to numcols do
         begin
            ib := Map[col].cb;
            ie := Map[col].ce;
            with pt^ do
               begin
                  if BuffLen < ie then ie := BuffLen;
                  while ib <= ie do
                     begin
                        SortKey := SortKey + Txt^[ib];
                        ib := succ(ib);
                     end;
               end;
         end;
      EditUpCase(SortKey);

      {this is trick code - we need to know the buffer length of each text
       line during pass2 but we don't want to have to read the .txt file
       so what we do is tack the buffer length on to the end of the key
       so we can get it directly from the index.}

       {Pad the SortKey with spaces to MaxKeyLen - 4 and then convert
        the buffer length to a 4 character string and concatenate it
        to the end of the key.}
```

```
        while Length(SortKey) >= (MaxKeyLen - 4) do
           Delete(SortKey,Length(SortKey),1);
        while Length(SortKey) < (MaxKeyLen - 4) do SortKey := SortKey + ' ';
        Str(pt^.BuffLen,num);
        SortKey := SortKey + num;

       AddKey(Ixf,pt^.TxtRef,SortKey);
       if not OK then exit;

       pt := Edit(Curwin^.Stream,plin2,rt);
       if plin2 <> 0 then plin2 := pt^.FwdRef;
    until EndPass;
    Pass1 := true;
end;

function Pass2(pl : dataref) : boolean;

{  This procedure passes both the descriptor file from pl and the index in
   parallel and substitutes the txt^ returned from the index for the
   old txt^ thereby sorting the table.
}

var plin1,plin2 : dataref;
    pt          : Plinedesc;
    num         : string[4];
    code        : integer;
    EndPass     : boolean;

begin
   Pass2 := false;
   if pl = 0 then exit;
   pt := Edit(Curwin^.Stream,pl,wd);
   plin2 := pt^.FwdRef;
   if plin2 = 0 then exit;
   ClearKey(Ixf);
   EndPass := false;
   repeat
      if Plin2 = 0 then EndPass := true;
      NextKey(Ixf,IxRef,SortKey);
      if not OK then exit;

      {The rest of the trick - we now strip the last 4 characters of the
       SortKey and convert them to an integer and stuff them in the BuffLen
       field of the line descriptor}

      num := Copy(SortKey,MaxKeyLen - 3,4);
      Val(num,pt^.BuffLen,code);
```

```
         {Now change the text reference.}

         pt^.TxtRef := IxRef;

         pt := Edit(Curwin^.Stream,plin2,wd);
         if plin2 <> 0 then plin2 := pt^.FwdRef;
      until EndPass;
      Pass2 := true;
   end;

begin{EditOrder}
   if not Curwin^.Formatted then
      begin
         EditErrorMsg(61);
         exit;
      end;
   if not BldMap then
      begin
         EditErrorMsg(54);
         exit;
      end;
   pt := Edit(Curwin^.Stream,Curwin^.Topline,rd);
   while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,rd);
   if pt^.FwdRef = 0 then exit;
   p := pt^.FwdRef; {skip header line}
   Curwin^.Replast := Sort;

   {Create an index file}

   MakeIndex(Ixf,Ixn,MaxKeyLen,1);
   CloseIndex(Ixf);
   OpenIndex(Ixf,Ixn,MaxKeyLen,1);

   {Build the index}

   if not Pass1(p) then EditErrorMsg(70);

   {Read the index back}

   if not Pass2(p) then EditErrorMsg(71);

   {Delete the index}

   CloseIndex(Ixf);
   Assign(IxFilVar,Ixn);
   Erase(IxFilVar);
```

```
   {Somehow, get the text buffers re-synced from the disk.  This isn't really
    tricky code but it's dirty because it messes with variables that only the
    buffer manager is supposed to see.  But commit and abort transaction do
    the same thing which is to set all the buffer slots to zero which forces
    the buffer manager to re-read them from the disk even if they were already
    in memory.}

    with Curwin^, BufMap[Curwin^.Stream], StreamDef[Curwin^.Stream] do
       begin
          for i := 1 to BufSize do
             begin
                with slot^[i], slot^[i].Dptr^ do
                   begin
                      if (Flags and Dmask) <> 0 then EditSync(Stream,i);
                      Dref := 0;
                      Tref := 0;
                      BufCnt := Pred(BufCnt);
                   end;
             end;
       end;


   EditWindowTopFile;
   EditRealign;
end;{EditOrder}


overlay procedure EditUnique;
{  This procedure sequentially scans the rows in a stream and if two
   adjacent rows have the same values in the specified columns it removes
   the second row via EditDelline.
}

type  maprec = record
                  cb : integer;
                  ce : integer;
               end;

var   map         : array[1..MaxCols] of maprec;
                                    {for each column being sorted, this}
                                    {array holds the beginning and ending}
                                    {txtcols}
      done        : boolean;
      numcols     : 1..MaxCols;
      p           : dataref;
      pt          : Plinedesc;

   function BldMap : boolean;
```

```
{  This procedure extracts data from the curwin header for the columns
   specified in QualStr and builds the map of column positions.
}

const  sep     : char = ',';

var    name    : Varstring;
       c       : integer;
       q       : integer;


begin
   BldMap := true;
   q := 1;
   numcols := 1;
   with Curwin^ do
      begin
         while q <= Length(QualStr) do
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldMap := false;
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1) do
                     c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               numcols := succ(numcols);
               q := succ(q);
            end;
         numcols := pred(numcols);
      end;
end;

function Pass(p : dataref) : boolean;
{  This function performs one pass of the stream starting at p comparing
   adjacent rows according to the column numbers in Map.  If two rows are
```

```
   identical it deletes the second row and returns true otherwise it
   returns false.
}

var pl1,pl2 : dataref;
    pt      : Plinedesc;

   function EqualP(pl1,pl2 : dataref) : boolean;
   {  This function compares the lines pointed to by pl1 and pl2 according
      to the column numbers in Map and returns false if pl2 <> pl1.
   }

   var
        col          : integer;
        ib,ie,jb,je  : integer;
        val1,val2    : Varstring;
        pt           : Plinedesc;

   begin
      EqualP := true;
      if (pl1 = 0) or (pl2 = 0) then
         begin
            EqualP := false;
            exit;
         end;
      for col := 1 to numcols do
         begin
            val1 := '';
            val2 := '';
            ib := Map[col].cb;
            jb := ib;
            ie := Map[col].ce;
            je := ie;
            pt := Edit(Curwin^.Stream,pl1,rt);
            with pt^ do
               begin
                  if BuffLen < ie then ie := BuffLen;
                  while Txt^[ib] = ' ' do ib := succ(ib);
                  while Txt^[ie] = ' ' do ie := pred(ie);
                  while ib <= ie do
                     begin
                        val1 := val1 + Txt^[ib];
                        ib := succ(ib);
                     end;
               end;
            pt := Edit(Curwin^.Stream,pl2,rt);
            with pt^ do
               begin
```

```
                  if BuffLen < je then je := Bufflen;
                  while Txt^[jb] = ' ' do jb := succ(jb);
                  while Txt^[je] = ' ' do je := pred(je);
                  while jb <= je do
                     begin
                        val2 := val2 + Txt^[jb];
                        jb := succ(jb);
                     end;
               end;
            EditUpCase(val1);
            EditUpCase(val2);
            if val1 <> val2 then
               begin
                  EqualP := false;
                  exit;
               end;
         end;
   end;

begin
   Pass := true;
   if p = 0 then exit;
   pl1 := p;
   pt := Edit(Curwin^.Stream,p,rd);
   pl2 := pt^.FwdRef;
   if pl2 = 0 then exit;
   repeat
      EditBreathe;
      if AbortCmd then exit;
      Curwin^.Curline := pl2;
      if EqualP(pl1,pl2) then
         begin
            Pass := false;
            EditDeleteLine;
            EditRealign;
            pt := Edit(Curwin^.Stream,pl1,rd);
            if pl1 <> 0 then pl2 := pt^.FwdRef else pl2 := 0;
         end
      else
         begin
            pl1 := pl2;
            pt := Edit(Curwin^.Stream,pl2,rd);
            if pl2 <> 0 then pl2 := pt^.FwdRef;
         end;
   until pl2 = 0;
end;

begin{EditUnique}
```

```
   if not Curwin^.Formatted then
      begin
         EditErrorMsg(61);
         exit;
      end;
   if not BldMap then
      begin
         EditErrorMsg(54);
         exit;
      end;
   pt := Edit(Curwin^.Stream,Curwin^.Topline,rd);
   while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,rd);
   p := pt^.FwdRef;{never include the header}
   if p = 0 then exit;
   Curwin^.Replast := Unique;
   if Pass(p) then;
   EditWindowTopFile;
   EditRealign;
end;{EditUnique}

overlay procedure EditColAlign;

{  This procedure passes all the rows of the table and justifies the content
   of specified columns either Right of Left.
}

type  maprec = record
                  cb : integer;
                  ce : integer;
               end;

var   map         : array[1..MaxCols] of maprec;
                                      {for each column being sorted, this}
                                      {array holds the beginning and ending}
                                      {txtcols}
      numcols     : 1..MaxCols;
      p           : dataref;
      pt          : Plinedesc;
      i           : integer;
      rtjust      : boolean;

   function BldMap : boolean;

   {  This procedure extracts data from the curwin header for the columns
      specified in QualStr and builds the map of column positions.
   }

   const  sep    : char = ',';
```

```
var     name    : Varstring;
        c       : integer;
        q       : integer;


begin
   BldMap := true;
   q := 1;
   numcols := 1;
   with Curwin^ do
      begin
         while q <= Length(QualStr) do
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldMap := false;
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1)
                     do c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               numcols := succ(numcols);
               q := succ(q);
            end;
         numcols := pred(numcols);
      end;
end;

function Pass(pl : dataref) : boolean;

{  This function passes the descriptor file starting at pl, finds the
   corresponding text, and makes the requested change to each record.
}

var plin1,plin2 : dataref;
    pt          : Plinedesc;
```

```
 col          : integer;
 ib,ie        : integer;
 num          : string[4];
 EndPass      : boolean;

procedure MakeChange(pt : Plinedesc);

{  This procedure changes the text of the line pointed to by pt as
   follows:
}

var

   col     : integer;
   ib,ie   : integer;
   i       : integer;
   nul     : boolean;

begin
   for col := 1 to numcols do
      begin
         ib := Map[col].cb;
         ie := Map[col].ce;
         with pt^ do
            begin
               if BuffLen < ie then ie := BuffLen;
               nul := true;
               for i := ib to ie do if txt^[i] <> ' ' then nul := false;
               if nul then exit;
               if rtjust then
                  begin
                     while txt^[ie] = ' ' do
                        begin
                           for i := ie downto ib do
                              begin
                                 if (i > ib) then
                                    txt^[i] := txt^[i-1]
                                 else
                                    txt^[i] := ' ';
                              end;
                        end;
                  end
               else
                  begin
                     while txt^[ib] = ' ' do
                        begin
                           for i := ib to ie do
                              begin
```

```
                                        if (i < ie) then
                                           txt^[i] := txt^[i+1]
                                        else
                                           txt^[i] := ' ';
                                     end;
                                end;
                           end;
                      end;
                 end;
           end;

   begin
      Pass := false;
      if pl = 0 then exit;
      pt := Edit(Curwin^.Stream,pl,wt);
      plin2 := pt^.FwdRef;
      if plin2 = 0 then exit;
      EndPass := false;
      repeat
         if Plin2 = 0 then EndPass := true;
         MakeChange(pt);
         pt := Edit(Curwin^.Stream,plin2,wt);
         if plin2 <> 0 then plin2 := pt^.FwdRef;
      until EndPass;
      Pass := true;
   end;

begin{EditColAlign}
   if not Curwin^.Formatted then
      begin
         EditErrorMsg(61);
         exit;
      end;
   if not BldMap then
      begin
         EditErrorMsg(54);
         exit;
      end;
   if Curwin^.OptStr = 'R' then rtjust := true else rtjust := false;
   pt := Edit(Curwin^.Stream,Curwin^.Topline,rd);
   while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,rd);
   if pt^.FwdRef = 0 then exit;
   p := pt^.FwdRef; {skip header line}
   Curwin^.Replast := Nothing;

   {pass the file making the appropriate changes}
```

```
   if not Pass(p) then EditErrorMsg(75);

   EditWindowTopFile;
   EditRealign;
end;{EditColAlign}

overlay procedure EditColAdd;

{  This procedure passes all the rows of the table and adds a new column(s)
   after the column specified.  It also modifies the header line and forces
   a file close and open so the changes will be available to the user.  If
   the new column causes the table to exceed RecLen the command will abort.
}

type  maprec = record
                  cb : integer;
                  ce : integer;
               end;

var   map         : array[1..MaxCols] of maprec;
                                    {for each column being sorted, this}
                                    {array holds the beginning and ending}
                                    {txtcols}
      numcols     : 1..MaxCols;
      p           : dataref;
      pt          : Plinedesc;
      i           : integer;

   function BldMap : boolean;

   {  This procedure extracts data from the curwin header for the columns
      specified in QualStr and builds the map of column positions.
   }

   const  sep    : char = ',';

   var    name   : Varstring;
          c      : integer;
          q      : integer;


   begin
      BldMap := true;
      q := 1;
      numcols := 1;
      with Curwin^ do
         begin
            while q <= Length(QualStr) do
```

```
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldMap := false;
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1)
                     do c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               numcols := succ(numcols);
               q := succ(q);
            end;
         numcols := pred(numcols);
         if numcols > 1 then
            begin
               BldMap := False;
               EditErrorMsg(73);
            end;
      end;
end;


function BldHdr(pt : Plinedesc) : boolean;
{  This procedure builds a header for the destination window.  This is the
   concatenation of the source window header and the current window header.
   It returns true unless there was an error.
}

var i,j  : integer;
    newh : Ptextline;
    newl : integer;
    ib,ie: integer;

begin
   BldHdr := true;
   newl := Curwin^.Hdrlen + Length(Curwin^.OptStr);
   if newl > StreamDef[Curwin^.Stream].RecLen then
```

```
      begin
         BldHdr := false;
         exit;
      end;
   GetMem(newh,succ(newl));
   for i := 1 to Curwin^.HdrLen do newh^[i] := Curwin^.Header^[i];
   for i := succ(Curwin^.HdrLen) to newl do newh^[i] := ' ';
   ib := succ(Map[1].ce);
   ie := ib + Length(Curwin^.OptStr);
   for i := 1 to Length(Curwin^.OptStr) do
      begin
         j := newl;
         while j > ib do
            begin
               newh^[j] := newh^[j-1];
               pt^.txt^[j] := pt^.txt^[j-1];
               j := pred(j);
            end;
      end;
   j := 1;
   for i := succ(ib) to ie do
      begin
         pt^.txt^[i] := Curwin^.OptStr[j];
         case Curwin^.OptStr[j] of
            '-'  : newh^[i] := chr(Horizontal);
            '|'  : newh^[i] := chr(TConnect);
            else   newh^[i] := Curwin^.OptStr[j];
         end;
         j := succ(j);
      end;
   FreeMem(Curwin^.Header,succ(Curwin^.HdrLen));
   Curwin^.Header := newh;
   Curwin^.HdrLen := newl;
   pt^.BuffLen := succ(newl);
end;


function Pass(pl : dataref) : boolean;

{  This function passes the descriptor file starting at pl, finds the
   corresponding text, and makes the requested change to each record.
}

var plin1,plin2 : dataref;
    pt          : Plinedesc;
    col         : integer;
    ib,ie       : integer;
    num         : string[4];
```

```
      EndPass     : boolean;

    procedure MakeChange(pt : Plinedesc);

    {  This procedure changes the text of the line pointed to by pt as
       follows:
    }

    var

       col     : integer;
       ib,ie   : integer;
       i,j     : integer;

    begin
       ib := succ(Map[1].ce);
       ie := ib + Length(Curwin^.OptStr);
       for i := 1 to Length(Curwin^.OptStr) do
          begin
             j := Curwin^.HdrLen;
             while j > ib do
                begin
                   pt^.txt^[j] := pt^.txt^[j-1];
                   j := pred(j);
                end;
          end;
       for i := succ(ib) to ie do pt^.txt^[i] := ' ';
       pt^.BuffLen := pt^.BuffLen + Length(Curwin^.OptStr);
    end;


  begin
     Pass := false;
     if pl = 0 then exit;
     pt := Edit(Curwin^.Stream,pl,wt);
     plin2 := pt^.FwdRef;
     if Plin2 = 0 then exit;
     EndPass := false;
     repeat
        if Plin2 = 0 then EndPass := true;
        MakeChange(pt);
        pt := Edit(Curwin^.Stream,plin2,wt);
        if plin2 <> 0 then plin2 := pt^.FwdRef;
     until EndPass;
     Pass := true;
  end;

begin{EditColAdd}
```

```
   if not Curwin^.Formatted then
      begin
         EditErrorMsg(61);
         exit;
      end;
   if not BldMap then
      begin
         EditErrorMsg(54);
         exit;
      end;
   pt := Edit(Curwin^.Stream,Curwin^.Topline,wt);
   while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,wt);

   if not BldHdr(pt) then
      begin
         EditErrorMsg(76);
         exit;
      end;

   p := pt^.FwdRef; {now skip header line for the pass}

   Curwin^.Replast := Nothing;

   {pass the file making the appropriate changes}

   if not Pass(p) then EditErrorMsg(76);

   EditWindowTopFile;
   EditRealign;

end;{EditColAdd}

overlay procedure EditColChg;

{  This procedure passes all the rows of the table and changes the width
   and/or header of the specified column.  Decreasing the width may obviously
   result in the truncation of data values.
}

type  maprec = record
                  cb : integer;
                  ce : integer;
               end;

var   map         : array[1..MaxCols] of maprec;
                                    {for each column being sorted, this}
                                    {array holds the beginning and ending}
                                    {txtcols}
```

```
   numcols      : 1..MaxCols;
   p            : dataref;
   pt           : Plinedesc;
   i            : integer;
   NewEnd       : integer;
   NewLen       : integer;

function BldMap : boolean;

{  This procedure extracts data from the curwin header for the columns
   specified in QualStr and builds the map of column positions.
}

const  sep     : char = ',';

var    name    : Varstring;
       c       : integer;
       q       : integer;


begin
   BldMap := true;
   q := 1;
   numcols := 1;
   with Curwin^ do
      begin
         while q <= Length(QualStr) do
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldMap := false;
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1)
                     do c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               numcols := succ(numcols);
```

```
            q := succ(q);
         end;
      numcols := pred(numcols);
      if numcols > 1 then
         begin
            BldMap := False;
            EditErrorMsg(73);
         end;
   end;
end;


function BldHdr(pt : Plinedesc) : boolean;

{  This procedure re-builds the header for the specified window.}

var i,j     : integer;
    newh    : Ptextline;
    row     : Ptextline;
    newl    : integer;
    ib,ie   : integer;
    nb,ne   : integer;
    diffl   : integer;
    pastend : boolean;


begin
   BldHdr := true;
   newl := Curwin^.Hdrlen + Length(Curwin^.OptStr) -
           (succ(Map[1].ce) - Map[1].cb) - 1;
   diffl := newl - Curwin^.Hdrlen;
   ne := succ(Map[1].ce) + diffl;
   NewEnd := ne;
   NewLen := newl;
   if newl > StreamDef[Curwin^.Stream].RecLen then
      begin
         BldHdr := false;
         exit;
      end;

   GetMem(newh,succ(newl));
   GetMem(row,succ(newl));
   i := 1;
   while i <= pred(Map[1].cb) do
      begin
         newh^[i] := Curwin^.Header^[i];
         row^[i] := pt^.txt^[i];
         i := succ(i);
```

```
      end;
   while i <= ne do
      begin
         newh^[i] := ' ';
         row^[i] := ' ';
         i := succ(i);
      end;
   j := Map[1].ce + 2;
   pastend := false;
   while i <= newl do
      begin
         if not pastend then
            begin
               row^[i] := pt^.txt^[j];
               newh^[i] := Curwin^.Header^[j];
            end
         else
            begin
               row^[i] := ' ';
               newh^[i] := ' ';
            end;
         if row^[i] = EndHdr then pastend := true;
         j := succ(j);
         i := succ(i);
      end;
   j := Map[1].cb;
   for i := 1 to Length(Curwin^.OptStr) do
      begin
         row^[j] := Curwin^.OptStr[i];
         case Curwin^.OptStr[i] of
            '-'  : newh^[j] := chr(Horizontal);
            '|'  : newh^[j] := chr(TConnect);
            else   newh^[j] := Curwin^.OptStr[i];
         end;
         j := succ(j);
      end;
   FreeMem(Curwin^.Header,succ(Curwin^.HdrLen));
   Curwin^.Header := newh;
   Curwin^.HdrLen := newl;
   for i := 1 to newl do pt^.txt^[i] := row^[i];
   for i := succ(newl) to pt^.BuffLen do pt^.txt^[i] := ' ';
   pt^.BuffLen := succ(newl);
   FreeMem(row,succ(newl));
end;


function Pass(pl : dataref) : boolean;
```

```
{  This function passes the descriptor file starting at pl, finds the
   corresponding text, and makes the requested change to each record.
}

var plin1,plin2 : dataref;
    pt          : Plinedesc;
    col         : integer;
    ib,ie       : integer;
    num         : string[4];
    EndPass     : boolean;
    row         : Ptextline;

   procedure MakeChange(pt : Plinedesc; row : Ptextline);

   {  This procedure changes the text of the line pointed to by pt as
      follows:
   }

   var

      i,j,k,n  : integer;
      newl     : integer;

   begin
      newl := pt^.BuffLen + Length(Curwin^.OptStr) -
              (succ(Map[1].ce) - Map[1].cb) - 1;
      i := 1;
      while (i <= Map[1].ce) and (i <= NewEnd) do
         begin
            row^[i] := pt^.txt^[i];
            i := succ(i);
         end;
      while i <= NewEnd do
         begin
            row^[i] := ' ';
            i := succ(i);
         end;
      j := Map[1].ce + 2;
      while j <= pt^.BuffLen do
         begin
            row^[i] := pt^.txt^[j];
            j := succ(j);
            i := succ(i);
         end;
      while i <= StreamDef[Curwin^.Stream].RecLen do
         begin
            row^[i] := ' ';
            i := succ(i);
```

```
            end;
         pt^.BuffLen := succ(newl);
         for i := 1 to pt^.BuffLen do pt^.txt^[i] := row^[i];
      end;


   begin
      Pass := false;
      if pl = 0 then exit;
      pt := Edit(Curwin^.Stream,pl,wt);
      plin2 := pt^.FwdRef;
      if Plin2 = 0 then exit;
      EndPass := false;
      GetMem(row,succ(StreamDef[Curwin^.Stream].RecLen));
      repeat
         if Plin2 = 0 then EndPass := true;
         MakeChange(pt,row);
         pt := Edit(Curwin^.Stream,plin2,wt);
         if plin2 <> 0 then plin2 := pt^.FwdRef;
      until EndPass;
      FreeMem(row,succ(StreamDef[Curwin^.Stream].RecLen));
      Pass := true;
   end;

begin{EditColChg}
   if not Curwin^.Formatted then
      begin
         EditErrorMsg(61);
         exit;
      end;
   if not BldMap then
      begin
         EditErrorMsg(54);
         exit;
      end;
   pt := Edit(Curwin^.Stream,Curwin^.Topline,wt);
   while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,wt);

   if not BldHdr(pt) then
      begin
         EditErrorMsg(76);
         exit;
      end;

   p := pt^.FwdRef; {now skip header line for the pass}

   Curwin^.Replast := Nothing;
```

```
   {pass the file making the appropriate changes}

   if not Pass(p) then EditErrorMsg(76);

   EditWindowTopFile;
   EditRealign;

end;{EditColChg}

overlay procedure EditColSw;

{  This procedure passes all the rows of the table and interchanges the
   positions of the two specified columns.
}

type  maprec = record
                  cb : integer;
                  ce : integer;
               end;

var   map         : array[1..MaxCols] of maprec;
                                    {for each column being sorted, this}
                                    {array holds the beginning and ending}
                                    {txtcols}
      numcols     : 1..MaxCols;
      p           : dataref;
      pt          : Plinedesc;
      i           : integer;
      NewEnd      : integer;

   function BldMap : boolean;

   {  This procedure extracts data from the curwin header for the columns
      specified in QualStr and builds the map of column positions.
   }

   const  sep    : char = ',';

   var    name   : Varstring;
          c      : integer;
          q      : integer;


   begin
      BldMap := true;
      q := 1;
      numcols := 1;
      with Curwin^ do
```

```
      begin
         while q <= Length(QualStr) do
            begin
               name := '';
               while (QualStr[q] <> sep) and (q <= Length(QualStr)) do
                  begin
                     name := name + QualStr[q];
                     q := succ(q);
                  end;
               c := EditPos(name,Header,HdrLen);
               if c = 0 then
                  begin
                     BldMap := false;
                     exit;
                  end;
               while (Header^[c] <> chr(TConnect)) and
                     (c > 1)
                     do c := pred(c);
               if c <> 1 then c := succ(c);
               map[numcols].cb := c;
               while Header^[c] <> chr(TConnect) do c := succ(c);
               map[numcols].ce := pred(c);
               numcols := succ(numcols);
               q := succ(q);
            end;
         numcols := pred(numcols);
         if numcols <> 2 then
            begin
               BldMap := False;
               EditErrorMsg(73);
            end;
         if Map[1].cb > Map[2].cb then
            begin
               c := Map[1].cb;
               Map[1].cb := Map[2].cb;
               Map[2].cb := c;
               c := Map[1].ce;
               Map[1].ce := Map[2].ce;
               Map[2].ce := c;
            end;
      end;
end;


function BldHdr(pt : Plinedesc) : boolean;

{  This procedure re-builds the header for the specified window.}
```

```
var i,j      : integer;
    newh     : Ptextline;
    row      : Ptextline;
    ib,ie    : integer;
    pastend  : boolean;


begin
   BldHdr := true;
   GetMem(newh,succ(Curwin^.HdrLen));
   GetMem(row,succ(Curwin^.HdrLen));
   for i := 1 to pred(Map[1].cb) do
      begin
         newh^[i] := Curwin^.Header^[i];
         row^[i] := pt^.txt^[i];
      end;
   j := Map[1].cb;
   for i := Map[2].cb to succ(Map[2].ce) do
      begin
         newh^[j] := Curwin^.Header^[i];
         row^[j] := pt^.txt^[i];
         j := succ(j);
      end;
   for i := (Map[1].ce + 2) to pred(Map[2].cb) do
      begin
         newh^[j] := Curwin^.Header^[i];
         row^[j] := pt^.txt^[i];
         j := succ(j);
      end;
   for i := Map[1].cb to succ(Map[1].ce) do
      begin
         newh^[j] := Curwin^.Header^[i];
         row^[j] := pt^.txt^[i];
         j := succ(j);
      end;
   for i := (Map[2].ce + 2) to Curwin^.HdrLen do
      begin
         newh^[j] := Curwin^.Header^[i];
         row^[j] := pt^.txt^[i];
         j := succ(j);
      end;
   for i := 1 to Curwin^.HdrLen do
      begin
         Curwin^.Header^[i] := newh^[i];
         pt^.txt^[i] := row^[i];
      end;
   FreeMem(row,succ(Curwin^.HdrLen));
   FreeMem(newh,succ(Curwin^.HdrLen));
```

```
end;


function Pass(pl : dataref) : boolean;

{  This function passes the descriptor file starting at pl, finds the
   corresponding text, and makes the requested change to each record.
}

var plin1,plin2 : dataref;
    pt          : Plinedesc;
    col         : integer;
    ib,ie       : integer;
    num         : string[4];
    EndPass     : boolean;
    row         : Ptextline;

   procedure MakeChange(pt : Plinedesc; row : Ptextline);

   {  This procedure changes the text of the line pointed to by pt as
      follows:
   }

   var

      i,j,k  : integer;

   begin
      for i := 1 to pred(Map[1].cb) do row^[i] := pt^.txt^[i];
      j := Map[1].cb;
      for i := Map[2].cb to succ(Map[2].ce) do
         begin
            row^[j] := pt^.txt^[i];
            j := succ(j);
         end;
      for i := (Map[1].ce + 2) to pred(Map[2].cb) do
         begin
            row^[j] := pt^.txt^[i];
            j := succ(j);
         end;
      for i := Map[1].cb to succ(Map[1].ce) do
         begin
            row^[j] := pt^.txt^[i];
            j := succ(j);
         end;
      for i := (Map[2].ce + 2) to Curwin^.HdrLen do
         begin
            row^[j] := pt^.txt^[i];
```

```
            j := succ(j);
          end;
       for i := 1 to Curwin^.HdrLen do pt^.txt^[i] := row^[i];
    end;


  begin
    Pass := false;
    if pl = 0 then exit;
    pt := Edit(Curwin^.Stream,pl,wt);
    plin2 := pt^.FwdRef;
    if Plin2 = 0 then exit;
    EndPass := false;
    GetMem(row,succ(Curwin^.HdrLen));
    repeat
      if Plin2 = 0 then EndPass := true;
      MakeChange(pt,row);
      pt := Edit(Curwin^.Stream,plin2,wt);
      if plin2 <> 0 then plin2 := pt^.FwdRef;
    until EndPass;
    FreeMem(row,succ(Curwin^.HdrLen));
    Pass := true;
  end;

begin{EditColSw}
  if not Curwin^.Formatted then
    begin
      EditErrorMsg(61);
      exit;
    end;
  if not BldMap then
    begin
      EditErrorMsg(54);
      exit;
    end;
  pt := Edit(Curwin^.Stream,Curwin^.Topline,wt);
  while pt^.BakRef <> 0 do pt := Edit(Curwin^.Stream,pt^.BakRef,wt);

  if not BldHdr(pt) then
    begin
      EditErrorMsg(76);
      exit;
    end;

  p := pt^.FwdRef; {now skip header line for the pass}

  Curwin^.Replast := Nothing;
```

```
   {pass the file making the appropriate changes}

   if not Pass(p) then EditErrorMsg(76);

   EditWindowTopFile;
   EditRealign;

end;{EditColSw}

overlay procedure EditCreateRel;

{  This procedure creates a new relation in an empty window  }

var   pt     : Plinedesc;
      p      : dataref;
      i      : integer;

   function BldHdr(pt : Plinedesc; p : dataref) : boolean;

   {  This procedure builds a header for the new table in the window.  }

   var i     : integer;
       newh  : Ptextline;
       newl  : integer;

   begin
      BldHdr := true;
      newl := Length(Curwin^.OptStr);
      if newl > StreamDef[Curwin^.Stream].RecLen then
         begin
            BldHdr := false;
            exit;
         end;
      GetMem(newh,succ(newl));
      for i := 1 to newl do
         begin
            pt^.txt^[i] := Curwin^.OptStr[i];
            case Curwin^.OptStr[i] of
               '-'  : newh^[i] := chr(Horizontal);
               '|'  : newh^[i] := chr(TConnect);
               else   newh^[i] := Curwin^.OptStr[i];
            end;
         end;
      pt^.txt^[succ(newl)] := '\';
      newh^[succ(newl)] := '\';
      Curwin^.Header := newh;
      Curwin^.HdrLen := newl;
      if not EditSizeLine(Curwin^.Stream,p,newl) then;
```

```
      Curwin^.Formatted := true;
   end;


   function ValWin : Boolean;

   {  This procedure checks that the current window is empty and has no
      file attached.  If these tests are passed the function returns true.
   }

   var  w    : integer;

   begin
      Valwin := false;
      with Curwin^ do
         begin
            if Formatted then exit;
            pt := Edit(Stream,Curline,rd);
            if pt^.BakRef <> 0 then exit;
            if pt^.FwdRef <> 0 then exit;
            if Filename <> NoFile then exit;
            for w := 1 to pt^.BuffLen do
               if pt^.txt^[w] <> ' ' then exit;
         end;
      Valwin := true;
   end;


begin{EditCreateRel}

   if not Valwin then
      begin
         EditErrorMsg(63);
         exit;
      end;
   p := Curwin^.Topline;
   pt := Edit(Curwin^.Stream,p,wt);
   if not BldHdr(pt,p) then
      begin
         EditErrorMsg(76);
         exit;
      end;
   with Curwin^ do
      for i := FirstLineNo to LastLineNo do
         if not EditInsBuf(16) then
            begin
               EditErrorMsg(35);
               Exit;
```

```
            end;
    EditWindowTopFile;
    EditScrollDown;
    EditRealign;

end;{EditCreateRel}
```

15. INFERENCE ENGINES

```
{
*****************************************************************************
Program:  TT                          Version 2.0          Date: 20 Nov 87
File:     infercmd.tt                 Version 2.00         Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                      under Contract F30602-C-85-0190

This module implements the 'inference' engines; i.e., relational operators
that exceed the power of a first order logic.
*****************************************************************************
}


overlay procedure EditClosure;

{  This routine accepts exactly two column names as input and then repeatedly
   (1) joins the relation in Curwin to itself, starting at Curline; writes the
   results to Destwin; and then joins Destwin with Curwin.  The repetition
   continues until the Destwin/Curwin join is empty.  Unlike EditJoin, the
   columns are not replicated but, instead, the resulting relation is appended
   with a column designated by the heading '^' to indicate the level (pass) at
   which the result was generated.
}

const Delimiter : char = '&';        {delimits qualifier expressions}
      RelOp     : char = '=';        {delimits column name from value}
      WinPrefix : char = '.';        {used to separate window.column}
      LitPrefix : char = '\';        {used to indicate literal value}
      CloseCol  : string[4] = '-^-!';
      LevelLen     = 3;
      AddLine   : boolean = true;
      SameLine  : boolean = false;
      MaxHits      = 1;              {max no. of occurrences of a value that}
                                     {will be looked at during one iteration}
type  target = record
                 ref : dataref;
                 num : integer;
               end;
```

```
    colmap = record
                swin : byte;       {source window number}
                sname: Varstring;  {source column name}
                twin : byte;       {target window number}
                tname: Varstring;  {target column name}
                tvalue:Varstring;  {value from curline.swin.sname}
                sbcol: integer;    {beginning column in source window}
                secol: integer;    {ending column in source window}
                tbcol: integer;    {beginning column in target window}
                tecol: integer;    {ending column in target window}
                litchar:Boolean;   {literal/variable switch}
             end;

    targetarray = array[1..MaxCols] of target;
    columnarray = array[1..MaxCols] of colmap;


var
  c,i         : integer;
  p           : dataref;
  s           : Varstring;
  qual        : Varstring;                    {qualifier workstring}
  hdr         : Ptextline;                    {file header workstring}
  sw          : Pwindesc;                     {points to source window}
  colset      : columnarray;
  numcols     : 0..MaxCols;                   {number of columns}
  ssi,ksi     : integer;
  targetset   : targetarray;
  SrchLine    : dataref;
  SrchCol     : integer;
  SrchLno     : integer;
  j,k,m,n     : integer;
  SrchSet     : integer;
  Searching   : boolean;
  ch          : byte;
  EndCol      : char;
  Destwin     : Pwindesc;
  row         : Ptextline;
  rowlen      : integer;
  srln        : integer;
  NullRow     : boolean;
  pt          : Plinedesc;
  scw         : Pwindesc;

   procedure ValWin;
   {  This procedure checks that the window specified in DestinStr exists
      and is empty and has no file attached.  If these tests are passed
      then Destwin is set to point to the window otherwise Destwin is set
```

```
   to nil.
}

var  p     : Pwindesc;
     w     : integer;
     pt    : Plinedesc;

begin
   Destwin := nil;
   if not(Curwin^.DestinStr[1] in ['1'..'9']) then exit;
   p := Window1;
   w := 1;
   while (p^.FwdLink <> Window1) and
         (chr(w + ord('0')) <> Curwin^.DestinStr[1]) do
       begin
          p := p^.FwdLink;
          w := succ(w);
       end;
   if chr(w + ord('0')) <> Curwin^.DestinStr[1] then exit;
   pt := Edit(p^.Stream,p^.Curline,rd);
   if pt^.BakRef <> 0 then exit;
   if pt^.FwdRef <> 0 then exit;
   if p^.Filename <> NoFile then exit;
   Destwin := p;
end;

function BldHdr : boolean;
{  This procedure builds a header for the destination window.  This is the
   concatenation of the source window header and a column designated by a
   heading of '^' to indicate the level (pass) at which the join occurred.
}

var i,j  : integer;

begin
   BldHdr := true;
   i := Curwin^.Hdrlen + Length(CloseCol);
   i := pred(((i + 16) div 16) * 16);
   if i > MaxDataRecSize then
      begin
         BldHdr := false;
         exit;
      end;
   if i > StreamDef[Destwin^.Stream].RecLen then with Destwin^ do
      begin
         EditDelStream(Stream);
         Stream := EditNewStream(Nofile,i);
         if Stream = 0 then
```

```
            begin
               BldHdr := false;
               Stream := EditNewStream(Nofile,DefRecSize);
               exit;
            end;
         Curline := EditMakTxtDes(Stream,16);
         Topline := Curline;
      end;
   GetMem(Destwin^.Header,succ(i));
   Destwin^.Hdrlen := i;
   i := 1;
   while sw^.Header^[i] <> EndHdr do
      begin
         Destwin^.Header^[i] := sw^.Header^[i];
         i := succ(i);
      end;
   Curwin^.Deslin := pred(i);
   j := 1;
   while  j <= Length(CloseCol)  do
      begin
         case CloseCol[j] of
            '-'    : Destwin^.Header^[i] := Chr(Horizontal);
            '|'    : Destwin^.Header^[i] := Chr(TConnect);
            else     Destwin^.Header^[i] := CloseCol[j];
         end;
         i := succ(i);
         j := succ(j);
      end;
   Destwin^.Header^[i] := EndHdr;
   Destwin^.Formatted := true;
end;

procedure GetRow(cp : dataref;row : Ptextline);
{  This procedure goes into the text lines of the source window and
   and the current window and concatenates them and puts the result
   in the row buffer.
}

var  c,r    : integer;
     pt     : Plinedesc;
     St     : string[LevelLen];

begin
   NullRow := true;
   FillChar(row^[1],rowlen,' ');
   c := 1;
   r := 1;
   pt := Edit(Curwin^.Stream,cp,rt);
```

```
   with pt^ do
      begin
         while c <= BuffLen do
            begin
               row^[r] := Txt^[c];
               if Txt^[c] <> ' ' then NullRow := false;
               c := succ(c);
               r := succ(r);
            end;
      end;
   Str(Curwin^.Despass:LevelLen,St);
   r := Destwin^.HdrLen;
   while Destwin^.Header^[r] <> EndHdr do r := pred(r);
   r := r - succ(LevelLen);
   for c := 1 to LevelLen do
      begin
         row^[r] := St[c];
         r := succ(r);
      end;
end;

function AddRow(NewLine : boolean; row : Ptextline) : boolean;
{  This procedure adds the contents of the row buffer as a new text
   line in the destination window.  It returns false on error.
}

var   c          : integer;
      SvCurwin   : Pwindesc;
      SvDline    : dataref;
      SvDcol     : integer;
      pt         : Plinedesc;

begin
   AddRow := true;
   SvCurwin := Curwin;
   SvDline := Destwin^.Curline;
   SvDcol := Destwin^.Colno;
   Curwin := Destwin;
   EditWindowBottomFile;
   with Curwin^ do
      begin
         if NewLine then if not(EditInsBuf(rowlen)) then
            begin
               AddRow := false;
               Curline := SvDline;
               Colno := SvDcol;
               EditRealign;
               Curwin := SvCurwin;
```

```
                exit;
             end;
          Colno := 1;
          pt := Edit(Stream,Curline,wt);
          with pt^ do for c := 1 to rowlen do
             begin
                if not EditSizeLine(Stream,Curline,succ(Colno)) then
                   begin
                      AddRow := false;
                      Curline := SvDline;
                      Colno := SvDcol;
                      EditRealign;
                      Curwin := SvCurwin;
                      exit;
                   end
                else
                   begin
                      Txt^[Colno] := row^[c];
                      Colno := succ(Colno);
                   end;
             end;
       end;
    Destwin^.Curline := SvDline;
    Destwin^.Colno := SvDcol;
    EditRealign;
    Curwin := SvCurwin;
 end;


begin  {EditClosure}

  EndCol := chr(TConnect);

  {EditClosure only works with formatted files}

  if not Curwin^.Formatted then
     begin
        if not AbortCmd then EditErrorMsg(61);
        exit;
     end;

  {parse the windows and column names}

with Curwin^ do
 begin
  if qualstr = '' then exit;
  qual := '';
  for numcols := 1 to MaxCols do
```

```
   begin
      colset[numcols].swin := ord('0');
      colset[numcols].sname := '';
      colset[numcols].twin := ord('0');
      colset[numcols].tname := '';
      colset[numcols].tvalue := '';
      colset[numcols].sbcol := 0;
      colset[numcols].secol := 0;
      colset[numcols].tbcol := 0;
      colset[numcols].tecol := 0;
      colset[numcols].litchar := false;
   end;
numcols := 1;
ssi := 1;
while ssi <= Length(qualstr) do
   begin
      while (qualstr[ssi] <> Delimiter) and
            (ssi <= Length(qualstr)) do
          begin
             qual := qual + qualstr[ssi]; {build a qualifier expression}
             ssi := ssi + 1;
          end;
      c := pos(RelOp,qual);              {find the position of relop}
      if c = 0 then
         begin
            if not AbortCmd then EditErrorMsg(51);
            exit;
         end;
      colset[numcols].swin := ord(qual[1]);
      colset[numcols].sname := copy(qual,3,c-3);
      colset[numcols].twin := ord(qual[c + 1]);
      colset[numcols].tname:= copy(qual,c+3,Length(qual)-c-2);
      numcols := numcols + 1;
      if numcols > MaxCols then
         begin
            if not AbortCmd then EditErrorMsg(53);
            exit;
         end;
      ssi := ssi + 1;                    {skip the delimiter}
      qual := '';
   end;
numcols := numcols - 1;                {set to last entry in colset}
for ksi := 1 to numcols do             {check for parse errors}
   begin
      if colset[ksi].sname = '' then
         begin
            if not AbortCmd then EditErrorMsg(55);
            exit;
```

```
         end;
      if colset[ksi].tname= '' then
         begin
            if not AbortCmd then EditErrorMsg(54);
            exit;
         end;
      if not (chr(colset[ksi].swin) in ['1'..'9']) then
         begin
            if not AbortCmd then EditErrorMsg(56);
            exit;
         end;
      if not (chr(colset[ksi].twin) in ['1'..'9']) then
         begin
            if chr(colset[ksi].twin) = LitPrefix then
               begin
                  colset[ksi].litchar := true;
                  colset[ksi].twin := ord('0');
               end
            else
               begin
                  if not AbortCmd then EditErrorMsg(56);
                  exit;
               end;
         end;
   end;

Notfound := false;                  {this is a GLOBAL so we do with it}
                                    {whatever the original code did but}
                                    {we won't use it for anything in this}
                                    {routine}

Searching := true;                  {this is our local control variable}
Replast := Closure;                 {For Ctrl-L command}
EditUpdphyscr;

{ Parse the option string }

IgnCase := false;
WholeWord := false;
AllOccur := false;
OutPut := true;
SetScan := false;

if Pos ('I', Optstr) <> 0 then
  IgnCase := true;
if Pos ('W', Optstr) <> 0 then
  WholeWord := true;
if Pos ('A', Optstr) <> 0 then
```

```
    AllOccur := true;
  if Pos ('S',Optstr) <> 0 then
    SetScan := true;
end;{with Curwin^}

 {parse the file header of Curwin to find the column positions in target win}

 for ksi := 1 to numcols do if not(colset[ksi].litchar) then
    begin
       c := EditPos(colset[ksi].tname,Curwin^.Header,Curwin^.HdrLen);
       if c = 0 then
          begin
             if not AbortCmd then EditErrorMsg(54);
             exit;
          end;
       while (c > 0) and
             (Curwin^.Header^[c] <> EndCol) do
             c := pred(c);
       c := succ(c);
       colset[ksi].tbcol := c;
       while (c <= Curwin^.Hdrlen) and
             (Curwin^.Header^[c] <> EndCol) do
             c := succ(c);
       colset[ksi].tecol := pred(c);
    end;

 {parse the headers of each source window to find the column positions and}
 {while were there, go into curline of each window and get the value in that}
 {column and stuff it in colset[].tvalue}

 for ksi := 1 to numcols do
    begin

       {find the source window pointer}

       sw := Window1;
       i := ord('1');
       while i <> colset[ksi].swin do
          begin
             sw := sw^.Fwdlink;
             i := succ(i);
             if sw = Window1 then
                begin
                   if not AbortCmd then EditErrorMsg(56);
                   exit;
                end;
          end;
```

```
{sw now points to the source window}

if not sw^.Formatted then
   begin
      if not AbortCmd then EditErrorMsg(62);
      exit;
   end;
c := EditPos(colset[ksi].sname,sw^.Header,sw^.Hdrlen);
if c = 0 then
   begin
      if not AbortCmd then EditErrorMsg(55);
      exit;
   end;
while (c > 0) and (sw^.Header^[c] <> EndCol) do c := pred(c);
c := succ(c);
colset[ksi].sbcol := c;
while (c <= sw^.Hdrlen) and
      (sw^.Header^[c] <> EndCol)
      do c := succ(c);
colset[ksi].secol := pred(c);

{sbcol,secol now contain the column positions for Curline in sw}

colset[ksi].tvalue := '';
i := colset[ksi].sbcol;
j := colset[ksi].secol;
pt := Edit(sw^.Stream,sw^.Curline,rt);
if (Curwin^.Nextpass) then
   begin
      sw^.Curline := pt^.FwdRef;
      scw := Curwin;
      Curwin := sw;
      if sw^.Curline = 0 then
         begin
            EditWindowTopFile;
            Curwin^.RepLast := Nothing;
            EditRealign;
            Curwin := scw;
            Exit;
         end;
      EditRealign;
      Curwin := scw;
      Curwin^.Nextpass := false;
      pt := Edit(sw^.Stream,sw^.Curline,rt);
   end;

if pt <> nil then with pt^ do
   begin
```

```
      {we have to check if this is the last column because not all lines}
      {are as long as the header and anything beyond BuffLen of Curline }
      {is garbage or another line}

            if not(colset[ksi].litchar) then
               begin

                  {the target value is variable so we have to extract
                   it from Curline of the source window}

                  if j > Bufflen then j := Bufflen;
                  while Txt^[i] = ' ' do i := succ(i);{strip leading blanks}
                  while Txt^[j] = ' ' do j := pred(j);{strip trailing blanks}
                  while i <= j do
                     begin
                        colset[ksi].tvalue := colset[ksi].tvalue + Txt^[i];
                        i := succ(i);
                     end;
               end
            else
               begin

                  {the target value is a literal which we parsed into
                   tname so we simply assign tname to tvalue and, since
                   the formats are the same, we also assign the same
                   column beginning/ending pointers}

                   colset[ksi].tvalue := colset[ksi].tname;
                   colset[ksi].tbcol := colset[ksi].sbcol;
                   colset[ksi].tecol := colset[ksi].secol;

               end;
         end;

      {colset.tvalue now has the attribute value from the source window}

      if colset[ksi].tvalue = '' then
         begin
            if not AbortCmd then EditErrorMsg(58);
            exit;
         end;
   end;

{if we're doing OutPut and this is the FirsTime, we have to validate the  }
{destination window, build the header, and put it in the window structure }
{and as the first line of the window, initialize the window and line ptrs }
{in our curwin structure, and then turn off FirsTime so we don't do this  }
```

```
{again by mistake.  Then we exit expecting to be recalled by RepLast.}

if (Curwin^.OutPut) and (Curwin^.FirsTime) then
   begin

      {validate the window}

      ValWin;
      if Destwin = nil then
         begin
            EditErrorMsg(44);
            exit;
         end;

      {build the header}

      if not(BldHdr) then
         begin
            EditErrorMsg(35);
            exit;
         end;

      {add the header definition record to the file}

      GetMem(row,succ(Destwin^.Hdrlen));
      rowlen := Destwin^.Hdrlen;
      i := 1;
      while Destwin^.Header^[i] <> EndHdr do
         begin
            case ord(Destwin^.Header^[i]) of
               Horizontal : row^[i] := '-';
               TConnect   : row^[i] := '|';
               else         row^[i] := Destwin^.Header^[i];
            end;
            i := succ(i);
         end;
      row^[i] := EndHdr;
      if not(Addrow(SameLine,row)) then;
      FreeMem(row,succ(rowlen));

      {set the number of levels to 0}

      Curwin^.Despass := 0;
      Curwin^.MTpass := true;
      Curwin^.Nextpass := false;

      {add the current (curline) row in the current (Curwin) window to
       the destination window (Destwin)}
```

```
      GetMem(row,succ(Destwin^.Hdrlen));
      rowlen := Destwin^.Hdrlen;
      GetRow(Curwin^.Curline,row);
      if not NullRow then
         if not (AddRow(AddLine,row)) then;
      FreeMem(row,succ(rowlen));
      Curwin^.Despass := 1;

      {modify the qualstr to indicate that the source window is now
       the destination window}

      Curwin^.QualStr[1] := Curwin^.DestinStr[1];

      i := 1;
      while i <= Length(Curwin^.QualStr) do
         begin
            if (Curwin^.QualStr[i] = Delimiter) and
               (Curwin^.QualStr[i + 2] = WinPreFix) and
               (Curwin^.QualStr[i + 1] in ['0'..'9']) then
               Curwin^.QualStr[i + 1] := Curwin^.DestinStr[1];
            i := succ(i);
         end;
      scw := Curwin;
      Curwin := sw;
      EditWindowTopFile;
      EditScrollDown;
      EditRealign;
      Curwin := scw;
      Curwin^.FirsTime := false;
      Curwin^.Deswin := Destwin;
      if Curwin^.AllOccur then
         begin
            EditPushtbf(ord(^L));
            EditPushtbf(Curwin^.WindNo + ord('0'));
            EditPushtbf(ord(^O));
         end;
      Exit;
   end;

{ Now the search begins.  From here on out the process is identical to }
{ EditSelect except where we optionally write the result to another window}
{ or a file because we have to concat the source window columns.  The reason}
{ we don't just call EditSelect is that we would have to de-parse colset to}
{ create qualstr and this would make Ctrl-L go to EditSelect instead of}
{ EditClosure.  Besides, this is overlay code so it doesn't cost us any space.}
{ (In fact, we couldn't call EditSelect because it is in this overlay area.)}
```

```
with Curwin^ do
  while Searching do
      begin
         for k := 1 to numcols do
            begin
               targetset[k].ref := 0;
               targetset[k].num := 0;
            end;
         for ksi := 1 to numcols do
            begin
               SrchLine := Curline;
               SrchCol := Colno;
               s := colset[ksi].tvalue;
               repeat
                  SrchLine := EditScanFwd (SrchLine,
                                           s,
                                           SrchCol,
                                           SrchLno,
                                           IgnCase,
                                           WholeWord);
                  targetset[ksi].ref := SrchLine;
                  targetset[ksi].num := targetset[ksi].num + SrchLno;
                  SrchCol:=succ(SrchCol);


               until ((SrchCol >= colset[ksi].tbcol) and
                     (SrchCol <= succ(colset[ksi].tecol))) or
                     (SrchLine = 0);


               {this is the FIRST occurrence of each key so if we get}
               {a SrchLine of nil then there are no occurrences so we}
               {may as well quit now}

               if SrchLine = 0 then
                  begin
                     NotFound := true;
                     Searching := false;
                     Nextpass := true;
                     if not MTpass then Despass := succ(Despass);
                     MTpass := true;
                     EditWindowTopFile;
                     EditRealign;
                     if AllOccur then
                        begin
                           EditPushtbf(ord(^L));
                           EditPushtbf(WindNo + ord('0'));
                           EditPushtbf(ord(^O));
```

```
                end
             else if not AbortCmd then EditErrorMsg(38);
             Exit;
          end; {if SrchLine = 0}
   end; {for ksi := 1 to numcols}

{The file has now been searched on every column and a viable
 row has been found for each column.  We now need to check that
 it's the SAME row.}

p := targetset[1].ref;
if (p <> 0) then
   begin
      SrchSet := targetset[1].num;
      for k := 2 to numcols do
         if SrchSet <> targetset[k].num then SrchSet := -1;
      if not(SrchSet<0) then p := targetset[1].ref else p := 0;
   end;

{OK - we have a row that satisfies the qualifier expression and
 we will do with it whatever the user asked us to}

if p <> 0 then
   begin
      Topline := p;
      Curline := p;
      Colno := SrchCol;
      if Colno < 1 then Colno := 1;
      Lineno := 1;
      Notfound := false;
      Searching := false;
      MTpass := false;
      if OutPut then
         begin
            Destwin := Deswin;
            GetMem(row,succ(Destwin^.Hdrlen));
            rowlen := Destwin^.Hdrlen;
            GetRow(Curwin^.Curline,row);
            if not NullRow then
               if not (AddRow(AddLine,row)) then;
            FreeMem(row,succ(rowlen));
         end;
      if SetScan then EditColorLine(true);
      if AllOccur then
         begin
            EditPushtbf(ord(^L));
            EditPushtbf(WindNo + ord('0'));
            EditPushtbf(ord(^O));
```

```
         end;
      EditRealign;
   end;

{We didn't find a row that satisfies the qualifier expression so
 the next step is to determine where to resume the search.}

if p = 0 then
   begin

      {find the largest lineno for which there is a non-nil}
      {pointer in the targetset - we have to make sure the }
      {pointer is non-nil because the lineno will be set to}
      {the file length if the search failed}

      SrchLno := -1;
      for k := 1 to numcols do
         if (targetset[k].ref <> 0) and
            (targetset[k].num > SrchLno) then
            begin
               SrchLno := targetset[k].num;
               p := targetset[k].ref;
            end;

      {if we found one then we set Curline to it }
      {and repeat the process again}

      if not (SrchLno < 0) then
         begin
            NotFound := false;
            Searching := true;
            Topline := p;
            Curline := p;
            Colno := 1;
            Lineno := 1;
            EditRealign;
         end;

      {We didn't find a place to resume the search so we have to
       assume that there aren't any more rows that satisfy the
       qualifier.  Hence, we advance the source window pointer
       to obtain a new qualifier and reset the current window
       to the beinning.}

      if SrchLno < 0 then
         begin
            NotFound := true;
            Searching := false;
```

```
                        with Deswin^ do
                           begin
                              if not MTpass then Despass := succ(Despass);
                              MTpass := true;
                              Nextpass := true;
                           end;
                        EditWindowTopFile;
                        EditRealign;
                        if AllOccur then
                           begin
                              EditPushtbf(ord(^L));
                              EditPushtbf(Curwin^.WindNo + ord('0'));
                              EditPushtbf(ord(^O));
                           end;
                     end;{if SrchLno < 0}
               end;{if p <> nil}
         end;{while Searching}
end; {EditClosure}
```

16. MISCELLANEOUS COMMANDS

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     slowcom.tt                  Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                     under Contract F30602-C-85-0190

Function:  Miscellaneous editing commands
*****************************************************************************
}

overlay procedure EditUpPage;
{  This routine scrolls up the current window n-1 lines, where n is
   the number of displayable lines in the window.
}
var
  Pagesize : integer;
  i        : integer;
  pt       : Plinedesc;

begin {EditUpPage}

  with Curwin^ do
    begin
      pt := Edit(Stream,Topline,rd);
      if pt^.BakRef = 0 then
      Begin
        Lineno := 1;
        CLineno := 1;
        Curline := Topline;
        Exit
      end;
      Pagesize := Lastlineno - Firstlineno;
      I := 1;
      while (I < Pagesize) and (pt^.BakRef <> 0) do
        begin
          Topline := pt^.BakRef;
          pt := Edit(Stream,Curline,rd);
          Curline := pt^.BakRef;
```

```
          pt := Edit(Stream,Topline,rd);
          I := Succ (I)
        end
    end
end; {EditUpPage}

overlay procedure EditDownPage;
{  This routine scrolls down the current window n-1 lines, where
   n-1 is the number of displayable lines in the window.
}
var  Pagesize, i : integer;
     pt          : Plinedesc;

begin {EditDownPage}
  with Curwin^ do
    begin
      Pagesize := Lastlineno - Firstlineno;
      I := 1;
      pt := Edit(Stream,Topline,rd);
      while (I < Pagesize) and (pt^.FwdRef <> 0) do
        begin
          Topline := pt^.FwdRef;
          I := Succ (I);
          pt := Edit(Stream,Curline,rd);
          if pt^.FwdRef = 0 then
            Lineno := Pred (Lineno)
          else
            Curline := pt^.FwdRef;
          pt := Edit(Stream,Topline,rd);
        end
    end
end; {EditDownPage}

overlay procedure EditToggleInsert;
{  This routine toggles the insert mode for the current window.  }
begin {EditToggleInsert}
  with Curwin^ do
    if Insertflag = Insert then
      Insertflag := Typeover
    else
      Insertflag := Insert
end; {EditToggleInsert}

overlay procedure EditInsertLine;
{  This routine splits the current line at the cursor position. }
var  pt1,pt2: Plinedesc;
     l : integer;
     c : integer;
```

```
begin {EditInsertLine}
  EditChangeFlag := true;
  pt1 := Edit(Curwin^.Stream,Curwin^.Curline,wt);
  with Curwin^ do
    begin
      c := pt1^.BuffLen;
      while (c > 0) and (pt1^.Txt^ [c] = ' ') do { find last non-blank }
        c := Pred (c);
      l := Succ(c - Colno); { Number of characters to move to new line     }
      if l < 0 then l := 0; { Could be none at all, if past end of line!   }
      if not(EditInsBuf(l)) then { Makes min size line if l = 0 }
        begin
          EditErrormsg (35);
          exit
        end;

      { Now split the line: stuff to the right of the cursor on the old
        line goes to the new, and fill old line on the right with spaces }

      if l > 0 then
        begin
          pt1 := Edit(Stream,Curline,wt);
          pt2 := Edit(Stream,pt1^.BakRef,wt);
          Move (pt2^.Txt^ [Colno], pt1^.Txt^ [1], l);
          Fillchar (pt2^.Txt^ [Colno], l, ' ')
        end
    end;
  EditRealign                {Realign other windows; we inserted stuff}
end; {EditInsertLine}

overlay procedure EditBeginningEndLine;
{  This routine positions the cursor to column 1 of the current line
   if not already in column 1, or after the last non-blank character in
   the line if already in column 1.
}
begin {EditBeginningEndLine}
  if Curwin^.Colno = 1 then
    EditEndLine
  else
    EditBeginningLine;
end; {EditBeginningEndLine}
```

17. USER INTERFACE PROCEDURES

```
{
*******************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     cp.tt                       Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                      under Contract F30602-C-85-0190

Function:  These routines perform all the console processing for the
           commands that do the work.
*******************************************************************************
}


overlay procedure EditCpSearchNext;
{  This routine repeats the previous search executed in the current window.
   Furthermore, if any other windows are joined to this window, their joins
   are repeated after the current window is searched, and any windows that
   are joined to them have their joins repeated, etc.  No window is done
   more than once to prevent infinite recursion on cyclic joins.
}

var   did        : array[1..9] of Pwindesc;
      i          : integer;
      SaveCurwin : Pwindesc;
      St         : Varstring;

   function wnum(wp : Pwindesc) : byte;
   {  This function takes a window pointer and returns the number of the
      window in the range ['1'..'9'].
   }

      var   p : Pwindesc;
            i : integer;

      begin{wnum of EditCpSearchNext}
         wnum := ord('0');
         p := Window1;
         i := 1;
         while (p <> wp) and (p^.FwdLink <> Window1) do
```

```
         begin
            p := p^.FwdLink;
            i := succ(i);
         end;
      if p = wp then wnum := i + ord('0');
   end;{wnum of EditCpSearchNext}

procedure DoDepJoin(wp : Pwindesc);
{  This procedure uses tail recursion to repeat the last join for any
   window that is join-dependent on wp.  Since cycles are allowed, the
   outer variable did[] is used to prevent infinite recursion.
}

   var  p    : Pwindesc;
        i    : integer;
        wno  : byte;
        wst  : string[2];
        done : boolean;

   begin{DoDepJoin of EditCpSearchNext}

      {find the first window that is join-dependent on wp and isn't done}

      p := Window1;
      wno := wnum(wp);
      if wno = ord('0') then exit;
      wst := chr(wno) + '.';
      repeat
         if (p^.RepLast = Join) and (pos(wst,p^.QualStr) = 1) then
            begin

               {we found a window dependent on wp so see if we}
               {it's already been done and, if not, do it}

               done := false;
               for i := 1 to 9 do if did[i] = p then done := true;
               if not done then
                  begin
                     Curwin := p;
                     EditZapCmdnam;
                     St := 'Join ' + Curwin^.QualStr + ' -' + Curwin^.Optstr;
                     EditAppCmdnam(St);
                     EditJoin;
                     i := 1;
                     while did[i] <> nil do i := succ(i);
                     did[i] := p;
                     if not NotFound then DoDepJoin(p);
                  end;
```

```
            end;
         p := p^.FwdLink
      until p = Window1;
   end;{DoDepJoin of EditCpSearchNext}

begin{EditCpSearchNext}
   SaveCurwin := Curwin;
   EditZapCmdnam;

   {repeat the last search operation on the current window}

   with Curwin^ do
      begin
         case RepLast of
            Find    : begin
                        St := 'Find: ' + SearchStr + ' -' + OptStr;
                        EditAppCmdnam(St);
                        EditFind;
                      end;
            Replace : begin
                        St := 'Replace: ' + SearchStr + ' with '
                              + ReplaceStr + ' -' + OptStr;
                        EditAppCmdnam(St);
                        EditReplace;
                      end;
            Keyword : begin
                        St := 'Keyword: ' + QualStr + ' -' + OptStr;
                        EditAppCmdnam(St);
                        EditKeyword;
                      end;
            Select  : begin
                        St := 'Select: ' + QualStr + ' -' + OptStr;
                        EditAppCmdnam(St);
                        EditSelect;
                      end;
            Join    : begin
                        St := 'Join: ' + QualStr + ' -' + OptStr;
                        EditAppCmdnam(St);
                        EditJoin;
                      end;
            Closure : begin
                        St := 'Closure: ' + QualStr + ' -' + OptStr;
                        EditAppCmdnam(St);
                        EditClosure;
                      end;
         end;
   end;{with Curwin}
```

```
    {initialize the did list so it can be used by DoDepJoin to prevent}
    {infinite recursion because DoDepJoin is a recursive procedure    }

    if not NotFound then
       begin
          did[1] := Curwin;
          for i := 2 to 9 do did[i] := nil;
          DoDepJoin(Curwin);
          Curwin := SaveCurwin;
       end;
 end;{EditCpSearchNext}

¦overlay procedure EditCpfind;
¦{  This routine asks for the find parameter }
¦var
¦  St      : Varstring;
¦
¦begin  {EditCpfind}
¦  EditAppcmdnam ('Find:');
¦  EditAskfor (St);
¦  if Abortcmd then exit;                    {If ctrl-u received during read}
¦  if Length (St) > 0 then
¦     begin
¦        Curwin^.SearchStr := St;
¦        EditAppCmdnam(' Options:');
¦        EditAskfor (St);
¦        if Abortcmd then exit;             {if ctrl-u received during read}
¦        EditUpcase (St);
¦        Curwin^.Optstr := St;
¦        if pos('X',Curwin^.Optstr) > 0 then EditFind
¦        else
¦           begin
¦              Curwin^.Replast := Find;
¦              EditZapCmdnam;
¦           end;
¦     end
¦  else if Curwin^.RepLast = Find then
¦     begin
¦        St :=' ' +  Curwin^.SearchStr + ' -' + Curwin^.Optstr;
¦        EditAppCmdnam(St);
¦        EditAskFor(St);
¦        EditZapCmdnam;
¦     end;
¦end; {EditCpfind}

¦overlay procedure EditCpreplace;
¦{  This routine asks for the find parameter }
¦var
```

```
  St      : Varstring;

begin  {EditCpreplace}
  EditAppcmdnam ('Find:');
  EditAskfor (St);
  if Abortcmd then exit;                    {if ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.Searchstr := St;
        EditAppcmdnam (' Replace with:');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        Curwin^.Replacestr := St;           {zero length replace string is ok}
        EditAppCmdnam(' Options:');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        EditUpcase (St);
        Curwin^.Optstr := St;
        if pos('X',Curwin^.Optstr) > 0 then EditReplace
        else
           begin
              Curwin^.Replast := Replace;
              EditZapCmdnam;
           end;
     end
  else if Curwin^.RepLast = Replace then
     begin
        St := ' ' +
              Curwin^.SearchStr + ' with ' + Curwin^.ReplaceStr + ' -' +
              Curwin^.Optstr;
        EditAppCmdnam(St);
        EditAskFor(St);
        EditZapCmdnam;
     end;
end;

overlay procedure EditCpKeyword;
{  This routine asks for the Keyword Search parameters }
var
  St      : Varstring;

begin  {EditCpKeyword}
  EditAppcmdnam ('Keywords:');
  EditAskfor (St);
  if Abortcmd then exit;                    {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
```

```
        EditAppCmdnam(' Options:');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        EditUpcase (St);
        Curwin^.Optstr := St;
        if pos('X',Curwin^.Optstr) > 0 then EditKeyword
        else
           begin
              Curwin^.RepLast := Keyword;
              EditZapCmdnam;
           end;
     end
  else if Curwin^.RepLast = Keyword then
     begin
        St := ' ' +  Curwin^.QualStr + ' -' + Curwin^.Optstr;
        EditAppCmdnam(St);
        EditAskFor(St);
        EditZapCmdnam;
     end;
end; {EditCpKeyword}

overlay procedure EditCpSelect;
{  This routine asks for the Relational Select parameters }
var
  St      : Varstring;

begin  {EditCpSelect}
  EditAppcmdnam ('Select:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        Curwin^.FirsTime := true;
        EditAppCmdnam(' Options:');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        EditUpcase (St);
        Curwin^.Optstr := St;


        {if Output is an option then we need someplace to put the result}

        if pos('O',Curwin^.Optstr) <> 0 then
           begin
              EditAppCmdnam(' Window:');
              EditAskFor(St);
              if Abortcmd then exit;
```

```
            Curwin^.DestinStr := St;
         end;
      if pos('X',Curwin^.Optstr) > 0 then EditSelect
      else
         begin
            Curwin^.RepLast := Select;
            EditZapCmdnam;
         end;
   end
  else if Curwin^.RepLast = Select then
   begin
      St := ' ' +  Curwin^.QualStr + ' -' + Curwin^.Optstr;
      EditAppCmdnam(St);
      if pos('O',Curwin^.Optstr) <> 0 then
         begin
            St := ' Window:' + Curwin^.DestinStr;
            EditAppCmdnam(St);
         end;
      EditAskFor(St);
      EditZapCmdnam;
   end;
end; {EditCpSelect}

overlay procedure EditCpJoin;
{  This routine asks for the Relational Join parameters }
var
  St      : Varstring;

begin  {EditCpJoin}
  EditAppcmdnam ('Join:');
  EditAskfor (St);
  if Abortcmd then exit;            {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        Curwin^.FirsTime := true;
        EditAppCmdnam(' Options:');
        EditAskfor (St);
        if Abortcmd then exit;            {if ctrl-u received during read}
        EditUpcase (St);
        Curwin^.Optstr := St;

        {if Output is an option then we need someplace to put the result}

        if pos('O',Curwin^.Optstr) <> 0 then
           begin
              EditAppCmdnam(' Window:');
              EditAskFor(St);
```

```
            if Abortcmd then exit;
            Curwin^.DestinStr := St;
         end;
      if pos('X',Curwin^.Optstr) > 0 then EditJoin
      else
         begin
            Curwin^.RepLast := Join;
            EditZapCmdnam;
         end;
   end
  else if Curwin^.RepLast = Join then
   begin
      St := ' ' +  Curwin^.QualStr + ' -' + Curwin^.Optstr;
      EditAppCmdnam(St);
      if pos('O',Curwin^.Optstr) <> 0 then
         EditAppCmdnam(' Window:' + Curwin^.DestinStr);
      EditAskFor(St);
      EditZapCmdnam;
   end;
end; {EditCpJoin}

overlay procedure EditCpProject;
{  This routine asks for the Relational Project parameters }
var
  St      : Varstring;

begin  {EditCpProject}
  EditAppcmdnam ('Project:');
  EditAskfor (St);
  if Abortcmd then exit;            {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        EditAppCmdnam(' Options:');
        EditAskfor (St);
        if Abortcmd then exit;            {if ctrl-u received during read}
        EditUpcase (St);
        Curwin^.Optstr := St;
        St := '';
        EditAppCmdnam(' Window:');
        EditAskFor(St);
        if Abortcmd then exit;
        Curwin^.DestinStr := St;
        EditProject;
     end
  else if Curwin^.RepLast = Project then
     begin
        St := ' ' +  Curwin^.QualStr + ' Window:' + Curwin^.DestinStr;
```

```
        EditAppCmdnam(St);
        EditAskFor(St);
        EditZapCmdnam;
      end;
end; {EditCpProject}

overlay procedure EditCpcrewin;
{  This routine creates a window by asking for the window size, file name,
   and the window to take the space from.  It calls a routine in ocmd.tt
   module (defined earlier) to actually generate the window.
}
var
  Sizest  : Varstring;
  Winst   : Varstring;
  Widst   : Varstring;
  Size    : integer;
  Win     : integer;
  Fn      : Varstring;
  Wid     : integer;
  i       : integer;
  p,q     : Pwindesc;
  f       : file;

begin {EditCpcrewin}
  EditAppcmdnam ('Open Window - Length:');
  EditAskfor (Sizest);
  if Abortcmd then exit;
  Size := 10;
  Val (Sizest, Size, i);
  if Size >= 3 then
    begin
      EditAppcmdnam(' File:');
      EditAskFor(Fn);
      if Abortcmd then exit;
      if Fn = '' then
         begin
            Fn := Nofile;
            EditAppcmdnam(' Width:');
            EditAskFor(Widst);
            if Abortcmd then exit;
            Wid := DefRecSize;
            Val (Widst,Wid,i);
            if (i = 0) then
               begin
                  Wid := pred(((Wid + 16) div 16) * 16);
                  if Wid > MaxDataRecSize then
                     begin
                        EditErrorMsg(41);
```

```
                        exit;
                     end;
               end;
         end
      else
         begin
            assign(f,Fn + DefExt);
            reset(f);
            if IOresult = 1 then
               begin
                  EditErrorMsg(1);
                  close(f);
                  exit;
               end;
            close(f);
         end;
      EditAppcmdnam (' Overlay:');
      EditAskfor (Winst);
      EditAppcmdnam(' * Please Wait *');
      EditUpdPhyScr;
      if Abortcmd then exit;
      Win := 1;
      Val (Winst, Win, i);
      if Win > 0 then
        q := EditWindowCreate (Size, Win, Wid, Fn);
      if q <> nil then
         begin
            p := Window1;
            i := 1;
            while p <> q do
               begin
                  p := p^.Fwdlink;
                  i := succ(i);
               end;
            EditWindowGoto(i);  {we made a new window so let's go to it}
         end
      else EditErrorMsg(IOstatus);
    end;
end; {EditCpcrewin}

overlay procedure EditCpdelwin;
{  This routine processes the delete window command by asking for the
   window to delete.  If the window contains a temporary file the user
   is prompted to rename it before it is closed.  Whatever file is in
   the window is properly closed before the window is deleted.
}
var
  Wst     : Varstring;
```

```
  Wno      : integer;
  i        : integer;
  p        : Pwindesc;
  Fn       : Varstring;
  NameOK   : boolean;

   function Empty(w : Pwindesc) : boolean;

   var  pt : Plinedesc;
        i  : integer;

   begin
      Empty := false;
      pt := Edit(w^.Stream,w^.Topline,rt);
      with pt^ do
         begin
            if BakRef <> 0 then exit;
            if FwdRef <> 0 then exit;
            i := 1;
            Empty := true;
            while i <= Bufflen do
               begin
                  if Txt^[i] <> ' ' then Empty := false;
                  i := succ(i);
               end;
         end;
   end;

begin {Cpdelwin}
  EditAppcmdnam ('Close Window:');
  EditAskfor (Wst);
  if Abortcmd then exit;
  Wno := Curwin^.WindNo;
  Val (Wst, Wno, i);
  if i = 0 then
     begin
        p := Window1;
        i := 1;
        while (p^.FwdLink <> Window1) and (i <> Wno) do
           begin
              p := p^.FwdLink;
              i := succ(i);
           end;
        if i <> Wno then
           begin
              EditErrorMsg(42);
              exit;
           end;
```

```
    end
  else
    begin
      p := Curwin;
    end;
  if StreamDef[p^.Stream].InTransaction then
    begin
      EditErrorMsg(83);
      exit;
    end;
  if (p^.Filename = Nofile) and (not Empty(p)) then with p^ do
    repeat
      EditAppcmdnam(' Save ' + Nofile + chr(ord('0') + Stream) + ' As:');
      EditAskFor(Fn);
      if Abortcmd then exit;
      NameOK := EditRenStream(Stream,Fn);
      if not NameOK then EditErrorMsg(64);
    until NameOK;
  EditAppcmdnam(' * Please Wait *');
  EditUpdPhyScr;
  if p = p^.FwdLink then {this is the only window and it is curwin}
    begin
      EditWindowDeleteText(Nofile);
      if p^.Stream = 0 then
        begin
          EditErrorMsg(IOstatus);
          halt;
        end;
    end
  else
    EditWindowDelete (Wno);
end; {EditCpdelwin}

overlay procedure EditCpClearWindow;
{  This routine processes the clear window command by asking for the
   window to clear.  If the window contains a temporary file the user
   is prompted to rename it before it is closed.  Whatever file is in
   the window is properly closed before the window is cleared.
}
var
  Wst     : Varstring;
  Wno     : integer;
  i       : integer;
  SvCurwin: Pwindesc;
  p       : Pwindesc;
  Fn      : Varstring;
  NameOK  : boolean;
```

```
   function Empty(w : Pwindesc) : boolean;

   var  pt : Plinedesc;
        i  : integer;

   begin
      Empty := false;
      pt := Edit(w^.Stream,w^.Topline,rt);
      with pt^ do
         begin
            if BakRef <> 0 then exit;
            if FwdRef <> 0 then exit;
            i := 1;
            Empty := true;
            while i <= Bufflen do
               begin
                  if Txt^[i] <> ' ' then Empty := false;
                  i := succ(i);
               end;
         end;
   end;

begin {CpClearWindow}
  EditAppcmdnam ('Clear Window:');
  EditAskfor (Wst);
  if Abortcmd then exit;
  Wno := Curwin^.WindNo;
  Val (Wst, Wno, i);
  if i = 0 then
     begin
        p := Window1;
        i := 1;
        while (p^.FwdLink <> Window1) and (i <> Wno) do
           begin
              p := p^.FwdLink;
              i := succ(i);
           end;
        if i <> Wno then
           begin
              EditErrorMsg(42);
              exit;
           end;
     end
  else
     begin
        p := Curwin;
     end;
  if StreamDef[p^.Stream].InTransaction then
```

```
      begin
         EditErrorMsg(83);
         exit;
      end;
   if (p^.Filename = Nofile) and (not Empty(p)) then with p^ do
      repeat
         EditAppcmdnam(' Save ' + Nofile + chr(ord('0') + Stream) + ' As:');
         EditAskFor(Fn);
         if Abortcmd then exit;
         NameOK := EditRenStream(Stream,Fn);
         if not NameOK then EditErrorMsg(64);
      until NameOK;
   SvCurwin := Curwin;
   Curwin := p;
   EditAppcmdnam(' * Please Wait *');
   EditUpdPhyScr;
   EditWindowDeleteText(Nofile);
   if p^.Stream = 0 then
      begin
         EditErrorMsg(IOstatus);
         halt;
      end
   else
      Curwin := SvCurwin;
end; {EditCpClearWindow}

overlay procedure EditCpOpenFile;
{  This routine processes open file.  If the window contains a temporary file,
   the user is prompted to rename it before the file is closed.  Whatever
   file is in the window is closed before the new file is opened.
}
var
   Wst      : Varstring;
   Wno      : integer;
   i        : integer;
   Fn       : Varstring;
   f        : file;
   NameOK   : boolean;

   function Empty(w : Pwindesc) : boolean;

   var  pt : Plinedesc;
        i  : integer;

   begin
      Empty := false;
      pt := Edit(w^.Stream,w^.Topline,rt);
      with pt^ do
```

```
         begin
            if BakRef <> 0 then exit;
            if FwdRef <> 0 then exit;
            i := 1;
            Empty := true;
            while i <= Bufflen do
               begin
                  if Txt^[i] <> ' ' then Empty := false;
                  i := succ(i);
               end;
         end;
   end;

begin {CpOpenFile}
  if StreamDef[Curwin^.Stream].InTransaction then
     begin
        EditErrorMsg(83);
        exit;
     end;
  Wno := Curwin^.WindNo;
  if (Curwin^.Filename = Nofile) and (not Empty(Curwin)) then with Curwin^ do
     repeat
        EditAppcmdnam('Save ' + Nofile + chr(ord('0') + Stream) + ' As:');
        EditAskFor(Fn);
        EditAppcmdnam(' ');
        if Abortcmd then exit;
        NameOK := EditRenStream(Stream,Fn);
        if not NameOK then EditErrorMsg(64);
     until NameOK;
  EditAppcmdnam('Open File:');
  EditAskFor(Fn);
  if Abortcmd then exit;
  EditAppCmdnam(' * Please Wait *');
  EditUpdPhyScr;
  if Fn = '' then Fn := Nofile else
     begin
        assign(f,Fn + DefExt);
        reset(f);
        if IOresult = 1 then
           begin
              EditErrorMsg(1);
              close(f);
              exit;
           end;
        close(f);
     end;
  EditWindowDeleteText(Fn);
  if Curwin^.Stream = 0 then
```

```
      begin
         EditErrorMsg(IOstatus);
         EditWindowDeleteText(Nofile);
         if Curwin^.Stream = 0 then
            begin
               EditErrorMsg(IOstatus);
               halt;
            end;
      end;
end; {EditCpOpenFile}

overlay procedure EditCpCloseFile;
{  This routine processes close file. If the window contains a temporary file,
   the user is prompted to rename it before the file is closed.  Whatever
   file is in the window is closed before the new file is opened.
}
var
  Wst      : Varstring;
  Wno      : integer;
  i        : integer;
  Fn       : Varstring;
  NameOK   : boolean;

   function Empty(w : Pwindesc) : boolean;

   var  pt : Plinedesc;
        i  : integer;

   begin
      Empty := false;
      pt := Edit(w^.Stream,w^.Topline,rt);
      with pt^ do
         begin
            if BakRef <> 0 then exit;
            if FwdRef <> 0 then exit;
            i := 1;
            Empty := true;
            while i <= Bufflen do
               begin
                  if Txt^[i] <> ' ' then Empty := false;
                  i := succ(i);
               end;
         end;
   end;

begin {CpCloseFile}
  if StreamDef[Curwin^.Stream].InTransaction then
     begin
```

```
        EditErrorMsg(83);
        exit;
      end;
  Wno := Curwin^.WindNo;
  if (Curwin^.Filename = Nofile) and (not Empty(Curwin)) then with Curwin^ do
     repeat
        EditAppcmdnam('Save ' + Nofile + chr(ord('0') + Stream) + ' As:');
        EditAskFor(Fn);
        EditAppcmdnam(' ');
        if Abortcmd then exit;
        NameOK := EditRenStream(Stream,Fn);
        if not NameOK then EditErrorMsg(64);
     until NameOK;
  EditAppcmdnam('Close File (Y/N)?');
  repeat
    EditUpdphyscr;
    if Abortcmd then
      exit;
  until EditKeypressed;
  if UpCase(Char(EditGetinput)) <> 'Y' then exit;
  EditAppcmdnam(' * Please Wait *');
  EditUpdPhyScr;
  EditWindowDeleteText(Nofile);
  if Curwin^.Stream = 0 then
     begin
        EditErrorMsg(IOstatus);
        halt;
     end;
end; {EditCpCloseFile}

overlay procedure EditCplnkwin;
{  This routine is called by the command dispatcher to link a window to
   another window's text stream. It asks for the window to link and the
   window to link to, and calls EditWindowLink (described earlier) to
   perform the link function.
}
var
  Wsource    : Varstring;
  Wdest      : Varstring;
  Wto, Wfrom : integer;
  i          : integer;

begin {EditCplnkwin}
  EditAppcmdnam ('Link From Window:');
  EditAskfor (Wsource);
  if Abortcmd then exit;
  Wfrom := 0;
  Val (Wsource, Wfrom, i);
```

```
    if Wfrom <= 0 then exit;
    EditAppcmdnam (' To Window:');
    EditAskfor (Wdest);
    if Abortcmd then exit;
    Wto := 0;
    Val (Wdest, Wto, i);
    if Wto <= 0 then exit;
    EditWindowLink (Wto, Wfrom);
  end; {EditCplnkwin}

| overlay procedure EditCpgotowin;
| {  This routine asks for the window number to place the cursor in next.
|    The user responds with a number which is evaluated modulo the number
|    of windows defined. It calls a routine described earlier in this module
|    to perform to goto.
| }
| var
|   St : Varstring;
|   Wno : integer;
|   i    : integer;
|
| begin {EditCpgotowin}
|   EditAppcmdnam ('Select Window:');
|   EditAskfor (St);
|   Wno := 0;
|   Val (St, Wno, i);
|   if Wno > 0 then
|     EditWindowGoto (Wno);
| end; {EditCpgotowin}

  overlay procedure EditCpexit;
  {  This routine processes the exit command, which allows the user
     to leave the editor, provided he confirms that he wants to do it.
  }

  var p : Pwindesc;

  begin {EditCpexit}
    EditAppcmdnam ('Exit (Y/N)?');
    repeat
      EditUpdphyscr;
      if Abortcmd then
        exit;
    until EditKeypressed;
    if UpCase(Char(EditGetinput)) <> 'Y' then exit;
    p := Window1;
    repeat
       with StreamDef[p^.Stream] do
```

```
        if InTransaction then
           begin
              EditErrorMsg(83);
              exit;
           end;
        p := p^.FwdLink;
  until p = Window1;
  EditExit;
end; {EditCpexit}

overlay procedure EditCptabdef;
{  This routine asks for a new tab width, and sends that to
   EditDefineTab to set Tabsize.
}
var
  Sizest : Varstring;
  Size, i : integer;

begin {EditCptabdef}
  EditAppcmdnam ('Tab Width:');
  EditAskfor (Sizest);
  if Abortcmd then exit;
  if Length (Sizest) = 0 then
    EditDefineTab (Pred (Curwin^.Colno))    {Set tab at current position}
  else
    begin
      Val (Sizest, Size, i);          {Expecting a number, then}
      if i = 0 then
        EditDefineTab (Size)
    end
end; {EditCptabdef}

overlay procedure EditCprfw;
{  This routine asks the user for an ascii text filename and then inserts the
   text of that file into the current window, using EditReaTxtFil.
}
var
  Fname : Varstring;

begin {EditCprfw}
  EditAppcmdnam ('Read Text File:');
  EditAskfor (Fname);
  if Abortcmd then exit;
  EditAppcmdnam (' * Please Wait *');
  EditUpdPhyscr;
  if Length (Fname) < 1 then exit;
  EditChangeFlag := true;
  EditReatxtfil (Fname);              {Does all the work}
```

```
  EditRealign                          {Realign other windows; we inserted stuff}
end; {EditCprfw}

overlay procedure EditCpwfw;
{  This routine processes the write window to ascii file command. It asks
   the user for the filename to write the entire window to, and
   then calls EditWriteToFile with the filename.
}

var
  Fname    : Varstring;
  p        : dataref;
  pt       : Plinedesc;
  foo      : boolean;


begin {EditCpwfw}
  EditAppcmdnam ('Write Text File:');
  EditAskfor (Fname);
  if Abortcmd then exit;
  EditAppcmdnam (' * Please Wait *');
  EditUpdPhyscr;
  if (Length (Fname) < 1) then exit;        { exit if filename is blank }
  with Curwin^ do
   begin
      p := Topline;
      pt := Edit(Stream,Topline,rd);
      while pt^.BakRef <> 0 do
         begin
            { Search for beginning of stream }
            p := pt^.BakRef;
            pt := Edit(Stream,pt^.BakRef,rd);
         end;
      foo := EditWriteToFile(Fname, p, 0);
   end;
end; {EditCpwfw}

overlay procedure EditCpOrder;
{  This routine asks for the sort parameters }

var
  St       : Varstring;

begin  {EditCpOrder}
  EditAppcmdnam ('Order:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
```

```
    begin
       Curwin^.QualStr := St;
       EditAppCmdnam(' Options:');
       EditAskfor (St);
       if Abortcmd then exit;              {if ctrl-u received during read}
       EditUpcase (St);
       Curwin^.Optstr := St;
       EditAppCmdnam(' * Please Wait *');
       EditUpdPhyScr;
       EditOrder;
    end
  else if Curwin^.RepLast = Sort then
    begin
       St := ' ' +  Curwin^.QualStr + ' -' + Curwin^.OptStr;
       EditAppCmdnam(St);
       EditAskFor(St);
       EditZapCmdnam;
    end;
end; {EditCpOrder}

overlay procedure EditCpUnique;
{  This procedure asks for the parameters for the remove duplicate rows
   procedure.  Also, the table must be sorted in order for this command
   to work.  Thus, the user is given the option of requesting a sort if
   he knows the table is unsorted.
}

var  St : Varstring;

begin{EditCpUnique}
  EditAppcmdnam ('Unique:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
    begin
       Curwin^.QualStr := St;
       EditAppCmdnam(' Options:');
       EditAskfor (St);
       if Abortcmd then exit;         {if ctrl-u received during read}
       EditUpcase (St);
       Curwin^.Optstr := St;
       EditAppCmdnam(' * Please Wait *');
       EditUpdPhyScr;
       if pos('S',St) <> 0 then EditOrder;
       EditUnique;
    end
  else if Curwin^.RepLast = Unique then
    begin
```

```
        St := ' ' +  Curwin^.QualStr + ' -' + Curwin^.OptStr;
        EditAppCmdnam(St);
        EditAskFor(St);
        EditZapCmdnam;
      end;
end;{EditCpUnique}

overlay procedure EditCpBatRead;
{  This procedure gets the window number from which a batch is to be read
   into the current window and calls EditBatRead to do the work.
}
var
  Wst     : Varstring;
  Wno     : integer;
  i       : integer;
  p       : Pwindesc;

   function Empty(w : Pwindesc) : boolean;

   var  pt : Plinedesc;
        i  : integer;

   begin
      Empty := false;
      pt := Edit(w^.Stream,w^.Topline,rt);
      with pt^ do
         begin
            if BakRef <> 0 then exit;
            if FwdRef <> 0 then exit;
            i := 1;
            Empty := true;
            while i <= Bufflen do
               begin
                  if Txt^[i] <> ' ' then Empty := false;
                  i := succ(i);
               end;
         end;
   end;

   function Compatible(w1,w2 : Pwindesc) : boolean;

   var i : integer;

   begin
      Compatible := false;
      if w1^.Formatted <> w2^.Formatted then exit;
      if StreamDef[w2^.Stream].RecLen < StreamDef[w1^.Stream].RecLen then exit;
      Compatible := true;
```

```
      if not(w1^.Formatted) then exit;
      for i := 1 to w1^.Hdrlen do
         if w1^.Header^[i] <> w2^.Header^[i] then Compatible := false;
   end;

   procedure Format(w1,w2 : Pwindesc);

   var i         : integer;
       pt        : Plinedesc;

   begin
      if w1^.Formatted then FreeMem(w1^.Header,succ(w1^.Hdrlen));
      GetMem(w1^.Header,succ(w2^.Hdrlen));
      w1^.Hdrlen := w2^.Hdrlen;
      move(w2^.Header^[1],w1^.Header^[1],w1^.Hdrlen);
      if not EditSizeLine(w1^.Stream,w1^.Curline,w1^.Hdrlen) then;
      pt := Edit(w1^.Stream,w1^.Curline,wt);
      with pt^ do
         for i := 1 to w1^.Hdrlen do
            case ord(w1^.Header^[i]) of
               Horizontal : Txt^[i] := '-';
               TConnect   : Txt^[i] := '|';
               else         Txt^[i] := w1^.Header^[i];
            end;
      w1^.Formatted := true;
   end;


begin
  EditAppcmdnam ('Read From Window:');
  EditAskfor (Wst);
  if Abortcmd then exit;
  EditAppcmdnam(' * Please Wait *');
  EditUpdPhyScr;
  Wno := 0;
  Val (Wst, Wno, i);
  p := Window1;
  i := 1;
  while (p^.FwdLink <> Window1) and (i <> Wno) do
     begin
        p := p^.FwdLink;
        i := succ(i);
     end;
  if (i <> Wno) or (p = Curwin) then
     begin
        EditErrorMsg(42);
        exit;
     end;
```

```
  if (Curwin^.Formatted) and
     (not(Empty(Curwin))) and
     (not(Compatible(p,Curwin))) then
     begin
        EditErrorMsg(63);
        exit;
     end;
  if (Empty(Curwin)) and (p^.Formatted) then Format(Curwin,p);
  EditBatRead(p);
end;

overlay procedure EditCpBatWrite;
{  This procedure gets the window number to which a batch is to be written
   from the current window and calls EditBatWrite to do the work.
}
var
  Wst     : Varstring;
  Wno     : integer;
  i       : integer;
  p       : Pwindesc;

   function Empty(w : Pwindesc) : boolean;

   var  pt : Plinedesc;
        i  : integer;

   begin
      Empty := false;
      pt := Edit(w^.Stream,w^.Topline,rt);
      with pt^ do
         begin
            if BakRef <> 0 then exit;
            if FwdRef <> 0 then exit;
            i := 1;
            Empty := true;
            while i <= Bufflen do
               begin
                  if Txt^[i] <> ' ' then Empty := false;
                  i := succ(i);
               end;
         end;
   end;

   function Compatible(w1,w2 : Pwindesc) : boolean;

   var i : integer;

   begin
```

```
      Compatible := false;
      if w1^.Formatted <> w2^.Formatted then exit;
      if StreamDef[w1^.Stream].RecLen < StreamDef[w2^.Stream].RecLen then exit;
      Compatible := true;
      if not(w1^.Formatted) then exit;
      for i := 1 to w1^.Hdrlen do
         if w1^.Header^[i] <> w2^.Header^[i] then Compatible := false;
   end;

   procedure Format(w1,w2 : Pwindesc);

   var i         : integer;
       pt        : Plinedesc;

   begin
      if w1^.Formatted then FreeMem(w1^.Header,succ(w1^.Hdrlen));
      GetMem(w1^.Header,succ(w2^.Hdrlen));
      w1^.Hdrlen := w2^.Hdrlen;
      move(w2^.Header^[1],w1^.Header^[1],w1^.Hdrlen);
      if not EditSizeLine(w1^.Stream,w1^.Curline,w1^.Hdrlen) then;
      pt := Edit(w1^.Stream,w1^.Curline,wt);
      with pt^ do
         for i := 1 to w1^.Hdrlen do
            case ord(w1^.Header^[i]) of
               Horizontal : Txt^[i] := '-';
               TConnect   : Txt^[i] := '|';
               else         Txt^[i] := w1^.Header^[i];
            end;
      w1^.Formatted := true;
   end;

begin
  EditAppcmdnam ('Write To Window:');
  EditAskfor (Wst);
  if Abortcmd then exit;
  EditAppcmdnam(' * Please Wait *');
  EditUpdPhyScr;
  Wno := 0;
  Val (Wst, Wno, i);
  p := Window1;
  i := 1;
  while (p^.FwdLink <> Window1) and (i <> Wno) do
     begin
        p := p^.FwdLink;
        i := succ(i);
     end;
  if (i <> Wno) or (p = Curwin) then
     begin
```

```
        EditErrorMsg(42);
        exit;
     end;
  if (p^.Formatted) and
     (not(Empty(p))) and
     (not(Compatible(p,Curwin))) then
     begin
        EditErrorMsg(63);
        exit;
     end;
  if (Empty(p)) and (Curwin^.Formatted) then Format(p,Curwin);
  EditBatWrite(p);
end;

overlay procedure EditCpTransAbort;
{  This procedure processes the abort transaction command }

var i : integer;
    p : Pwindesc;
    s : Varstring;

begin
   EditAppcmdnam ('Abort Transaction (Y/N)?');
   repeat
      EditUpdphyscr;
      if Abortcmd then exit;
   until EditKeypressed;
   if UpCase(Char(EditGetinput)) <> 'Y' then exit;
   EditAppcmdnam(' * Please Wait *');
   EditUpdPhyScr;
   with Curwin^, BufMap[Curwin^.Stream], StreamDef[Curwin^.Stream] do
      begin
         if not InTransaction then
            begin
               EditErrorMsg(81);
               exit;
            end;
         if NTstream <> 0 then
            begin
               EditErrorMsg(85);
               exit;
            end;
         for i := 1 to BufSize do
            with slot^[i], slot^[i].Dptr^ do
               if (Flags and Dmask) <> 0 then
                  begin
                     if (Flags and Dins) <> 0 then
                        begin
```

```
                        if Dref <> 0 then DeleteRec(DesHndl,Dref);
                        if Tref <> 0 then DeleteRec(TxtHndl,Tref);
                     end;
                  Dref := 0;
                  Tref := 0;
                  BufCnt := pred(BufCnt);
               end;
         InTransaction := false;
         TofRef := BTtofref;
         p := FwdLink;
         s := '';
         while p <> Curwin do
            begin
               if p^.NTstream = Stream then
                  begin
                     s := ^O + chr(p^.WindNo + Ord('0')) + ^O^A + 'y';
                     EditUserPush(s);
                     p^.NTstream := 0;
                  end;
               p := p^.FwdLink;
            end;
         if s <> '' then EditUserPush(^O + chr(WindNo + Ord('0')));
      end;
   p := Window1;
   repeat
      if p^.Stream = Curwin^.Stream then
         begin
            p^.Curline := p^.BTcurline;
            p^.Topline := p^.BTtopline;
         end;
      p := p^.FwdLink;
   until p = Window1;
   EditRealign;
end;

overlay procedure EditCpTransBegin;
{  This procedure processes the begin transaction command }

var i : integer;
    p : Pwindesc;

begin
   EditAppcmdnam ('Begin Transaction (Y/N)?');
   repeat
      EditUpdphyscr;
      if Abortcmd then exit;
   until EditKeypressed;
   if UpCase(Char(EditGetinput)) <> 'Y' then exit;
```

```
   EditAppcmdnam(' * Please Wait *');
   EditUpdPhyScr;
   with Curwin^, BufMap[Curwin^.Stream], StreamDef[Curwin^.Stream] do
      begin
         if InTransaction then
            begin
               EditErrorMsg(80);
               exit;
            end;
         for i := 1 to BufSize do
            with slot^[i],slot^[i].Dptr^ do
               if (Flags and Dmask) <> 0 then
                  begin
                     EditSync(Stream,i);
                     if (Flags and Ddel) <> 0 then
                        begin
                           Dref := 0;
                           Tref := 0;
                           BufCnt := pred(BufCnt);
                        end;
                  end;
         InTransaction := true;
         BTtofref := TofRef;
         NTstream := 0;
      end;
   p := Window1;
   repeat
      if p^.Stream = Curwin^.Stream then
         begin
            p^.BTcurline := p^.Curline;
            p^.BTtopline := p^.Topline;
         end;
      p := p^.FwdLink;
   until p = Window1;
end;

overlay procedure EditCpTransCommit;
{  This procedure processes the commit transaction command }

var i : integer;
    p : Pwindesc;
    s : Varstring;

begin
   EditAppcmdnam ('Commit Transaction (Y/N)?');
   repeat
      EditUpdphyscr;
      if Abortcmd then exit;
```

```
   until EditKeypressed;
   if UpCase(Char(EditGetinput)) <> 'Y' then exit;
   EditAppcmdnam(' * Please Wait *');
   EditUpdPhyScr;
   with Curwin^, BufMap[Curwin^.Stream], StreamDef[Curwin^.Stream] do
      begin
         if not InTransaction then
            begin
               EditErrorMsg(81);
               exit;
            end;
         if NTstream <> 0 then
            begin
               EditErrorMsg(85);
               exit;
            end;
         for i := 1 to BufSize do
            with slot^[i], slot^[i].Dptr^ do
               begin
                  if (Flags and Dmask) <> 0 then EditSync(Stream,i);
                  if (Flags and Ddel) <> 0 then
                     begin
                        Dref := 0;
                        Tref := 0;
                        BufCnt := pred(BufCnt);
                     end;
               end;
         InTransaction := false;
         p := FwdLink;
         s := '';
         while p <> Curwin do
            begin
               if p^.NTstream = Stream then
                  begin
                     s := ^O + chr(p^.WindNo + Ord('0')) + ^O^C + 'y';
                     EditUserPush(s);
                     p^.NTstream := 0;
                  end;
               p := p^.FwdLink;
            end;
         if s <> '' then EditUserPush(^O + chr(WindNo + Ord('0')));
      end;
end;

overlay procedure EditCpTransLeave;
{  This procedure processes the leave transaction command }

begin
```

```
   EditAppcmdnam ('Leave Transaction (Y/N)?');
   repeat
      EditUpdphyscr;
      if Abortcmd then exit;
   until EditKeypressed;
   if UpCase(Char(EditGetinput)) <> 'Y' then exit;
   with Curwin^, StreamDef[Curwin^.Stream] do
      begin
         if not InTransaction then
            begin
               EditErrorMsg(81);
               exit;
            end;
         NTstream := 0;
      end;
end;

overlay procedure EditCpTransEnter;
{  This procedure processes the enter transaction command }

var i : integer;
    p : Pwindesc;
    s : Varstring;
    w : integer;

begin
   EditAppcmdnam ('Enter Transaction (Y/N)?');
   repeat
      EditUpdphyscr;
      if Abortcmd then exit;
   until EditKeypressed;
   if UpCase(Char(EditGetinput)) <> 'Y' then exit;
   EditAppcmdnam(' With Window:');
   EditAskFor(s);
   if Abortcmd then exit;
   w := 0;
   Val(s,w,i);
   p := Window1;
   i := 1;
   while (p^.FwdLink <> Window1) and (i <> w) do
      begin
        i := succ(i);
        p := p^.FwdLink;
      end;
   if (i <> w) or (p = Curwin) then
      begin
         EditErrorMsg(82);
         exit;
```

```
      end;
   if not StreamDef[p^.Stream].InTransaction then
      begin
         EditErrorMsg(81);
         exit;
      end;
   EditAppcmdnam(' * Please Wait *');
   EditUpdPhyScr;
   with Curwin^, BufMap[Curwin^.Stream], StreamDef[Curwin^.Stream] do
      begin
         if InTransaction then
            begin
               EditErrorMsg(80);
               exit;
            end;
         for i := 1 to BufSize do
            with slot^[i],slot^[i].Dptr^ do
               if (Flags and Dmask) <> 0 then
                  begin
                     EditSync(Stream,i);
                     if (Flags and Ddel) <> 0 then
                        begin
                           Dref := 0;
                           Tref := 0;
                           BufCnt := pred(BufCnt);
                        end;
                  end;
         InTransaction := true;
         BTtofref := TofRef;
         NTstream := p^.Stream;
      end;
   p := Window1;
   repeat
      if p^.Stream = Curwin^.Stream then
         begin
            p^.BTcurline := p^.Curline;
            p^.BTtopline := p^.Topline;
         end;
      p := p^.FwdLink;
   until p = Window1;
end;

overlay procedure EditCpTransScan;
{  This procedure processes the scan transaction command }

var i : integer;

begin
```

```
   EditAppcmdnam ('Scan Transaction (Y/N)?');
   repeat
      EditUpdphyscr;
      if Abortcmd then exit;
   until EditKeypressed;
   if UpCase(Char(EditGetinput)) <> 'Y' then exit;
   EditAppcmdnam(' * Please Wait *');
   EditUpdPhyScr;
   with BufMap[Curwin^.Stream], StreamDef[Curwin^.Stream] do
      begin
         if not InTransaction then
            begin
               EditErrorMsg(81);
               exit;
            end;
         for i := 1 to BufSize do
            with slot^[i].Dptr^ do
               if (Flags and Dmask) <> 0 then
                  Flags := Flags or Colored;
      end;
end;

overlay procedure EditCpTransShare;
{  This procedure processes the acquire shared lock request }
begin
end;

overlay procedure EditCpTransExcl;
{  This procedure processes the acquire exclusive lock request }
begin
end;

overlay procedure EditCpInsertCtrlChar;
{  This routine reads a character from the typeahead buffer and inserts
   its corresponding control character into the text by calling the
   text input routine, EditPrctxt (Ch).
}
var
  Ch : byte;

begin {EditInsertCtrlChar}
  ch := EditCtrlChar;
  if ch = 255 then exit
              else EditPrctxt (Ch);
end; {EditInsertCtrlChar}

overlay procedure EditCpColAlign;
```

```
{  This routine asks for the Align Column parameters }

var
  St      : Varstring;

begin  {EditCpColAlign}
  EditAppcmdnam ('Align:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        EditAppCmdnam(' Options:');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        EditUpcase (St);
        Curwin^.Optstr := St;
        if Abortcmd then exit;
        EditColAlign;
     end;
end; {EditCpColAlign}

overlay procedure EditCpColAdd;

{  This routine asks for the Add Column parameters }

var
  St      : Varstring;

begin  {EditCpColAdd}
  EditAppcmdnam ('Add after:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        EditAppCmdnam(' New Col Header:');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        Curwin^.Optstr := St;
        if Abortcmd then exit;
        EditColAdd;
     end;
end; {EditCpColAdd}

overlay procedure EditCpColChg;

{  This routine asks for the Change Column parameters }
```

```
var
  St      : Varstring;

begin  {EditCpColChg}
  EditAppcmdnam ('Change:');
  EditAskfor (St);
  if Abortcmd then exit;            {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        EditAppCmdnam(' New Col Header:');
        EditAskfor (St);
        if Abortcmd then exit;           {if ctrl-u received during read}
        Curwin^.Optstr := St;
        if Abortcmd then exit;
        EditColChg;
     end;
end; {EditCpColChg}

overlay procedure EditCpColDrop;

{  This routine asks for the Drop Column parameters }

var
  St      : Varstring;

begin  {EditCpColDrop}
  EditAppcmdnam ('Drop:');
  EditAskfor (St);
  if Abortcmd then exit;            {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        EditAppCmdnam(' Confirm(Y/N):');
        EditAskfor (St);
        if Abortcmd then exit;           {if ctrl-u received during read}
        if (St <> 'Y') and (St <> 'y') then exit;
        Curwin^.Optstr := '';
        if Abortcmd then exit;
        EditColChg;
     end;
end; {EditCpColDrop}

overlay procedure EditCpColSw;

{  This routine asks for the Switch Column parameters }
```

```
var
  St      : Varstring;

begin  {EditCpColSw}
  EditAppcmdnam ('Switch:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
        Curwin^.QualStr := St;
        EditAppCmdnam(' Confirm(Y/N):');
        EditAskfor (St);
        if Abortcmd then exit;              {if ctrl-u received during read}
        if (St <> 'Y') and (St <> 'y') then exit;
        Curwin^.Optstr := '';
        if Abortcmd then exit;
        EditColSw;
     end;
end; {EditCpColSw}


overlay procedure EditCpCreateRel;

{  This routine asks for the create relation parameters }

var
  St      : Varstring;

begin  {EditCpCreateRel}
   EditAppCmdnam(' Header:');
   EditAskfor (St);
   if Abortcmd then exit;              {if ctrl-u received during read}
   Curwin^.Optstr := St;
   if Abortcmd then exit;
   EditCreateRel;
end; {EditCpCreateRel}

overlay procedure EditCpClosure;
{  This routine asks for the Relational closure parameters }
var
  St      : Varstring;

begin  {EditCpClosure}
  EditAppcmdnam ('Closure:');
  EditAskfor (St);
  if Abortcmd then exit;              {If ctrl-u received during read}
  if Length (St) > 0 then
     begin
```

```
      Curwin^.QualStr := St;
      Curwin^.FirsTime := true;
      EditAppCmdnam(' Options:');
      EditAskfor (St);
      if Abortcmd then exit;              {if ctrl-u received during read}
      EditUpcase (St);
      Curwin^.Optstr := St;

      {we need someplace to put the interim results; i.e., a scratchpad}

      EditAppCmdnam(' Window:');
      EditAskFor(St);
      if Abortcmd then exit;
      Curwin^.DestinStr := St;

      if pos('X',Curwin^.Optstr) > 0 then EditClosure
      else
         begin
            Curwin^.RepLast := Closure;
            EditZapCmdnam;
         end;
   end
else if Curwin^.RepLast = Closure then
   begin
      St := ' ' +  Curwin^.QualStr + ' -' + Curwin^.Optstr;
      EditAppCmdnam(St);
      if pos('O',Curwin^.Optstr) <> 0 then
         EditAppCmdnam(' Window:' + Curwin^.DestinStr);
      EditAskFor(St);
      EditZapCmdnam;
   end;
end; {EditCpClosure}
```

18. PREFIXED COMMAND DISPATCHER

```
{
*****************************************************************************
Program:  TT                          Version 2.0          Date: 20 Nov 87
File:     koq.tt                      Version 2.00         Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                      under Contract F30602-C-85-0190

Function:  These routines dispatch the second Ctrl character received if
           the first was Ctrl-k, Ctrl-o or Ctrl-q.
*****************************************************************************
}


procedure EditK;
{  This routine is the utility supercommand processor for the
   editor.  The Ctrl-K command is simply a submenu of commands
   to which file I/O and block commands may be added. You can remove
   commands from the editor by removing their references in the case
   statement in this procedure.  If all of the commands in a module
   are to be deleted (say, all block commands), then you can simply
   omit that include and comment out the references in this procedure.
}
var
  ch :  byte;

begin {EditK}
  EditZapcmdnam;
  Asking := true;
  EditAppcmdnam ('^K');
  ch := EditCtrlChar;
  case ch of                              {COMMAND DESCRIPTION       }
    Ctrlb : EditBatBegin;                 {begin marking batch lines }
    Ctrlc : EditBatCopy;                  {copy batch of lines       }
    Ctrle : EditBatExclude;               {remove line from batch    }
    Ctrlg : EditCpBatRead;                {read batch of lines       }
    Ctrlh : EditBatClear;                 {clear batch of all lines  }
    Ctrli : EditBatInclude;               {include line in batch     }
    Ctrlk : EditBatEnd;                   {end marking batch lines   }
    Ctrll : EditBatPrior;                 {go to prior batch line    }
```

```
    Ctrln : EditBatNext;              {go to next batch line     }
    Ctrlp : EditCpBatWrite;           {write batch of lines      }
    Ctrlv : EditBatMove;              {move batch of lines       }
    Ctrly : EditBatDelete;            {delete batch of lines     }
    Ctrlr : EditCprfw;                {read file into window     }
    Ctrlo : EditCpOpenFile;           {open file into window     }
    Ctrlt : EditCptabdef;             {define tab width          }
    Ctrlw : EditCpwfw;                {write file from window    }
    Ctrlx : EditCpexit;               {exit editor               }
    Ctrlz : EditCpCloseFile;          {close file in window      }
  end;
Asking := false;
EditPrcTxt(NotAvailable);             {swap in char processor}
end; {EditK}


procedure EditO;
{  This routine processes the Ctrl-O command processing for the
   editor.  The Ctrl-O command is simply a submenu of commands
   to which text manipulation commands may be added. You can remove
   commands from the editor by removing their references in the case
   statement in this procedure.  If all of the commands in a module
   are to be deleted, then you can simply omit that include and
   comment out the references in this procedure.

}
var
   ch : byte;

begin {EditO}
  EditZapcmdnam;
  Asking := true;
  EditAppcmdnam ('^O');
  ch := EditCtrlChar;
  case ch of                          {COMMAND DESCRIPTION      }
    Ctrla : EditCpTransAbort;         {abort transaction        }
    Ctrlb : EditCpTransBegin;         {begin transaction        }
    Ctrlc : EditCpTransCommit;        {commit transaction       }
    Ctrld : EditCpTransLeave;         {leave transaction        }
    Ctrle : EditCpTransEnter;         {enter transaction        }
    Ctrlg : EditCpgotowin;            {goto window (prompt)     }
    Ctrli : EditGotoColumn;           {goto next relational col}
    Ctrlj : EditCplnkwin;             {link (join) window       }
    Ctrlk : EditChangeCase;           {change case              }
    Ctrll : EditCenterLine;           {center text              }
    Ctrln : EditCpTransScan;          {include transaction scan}
    Ctrlo : EditCpcrewin;             {open new window          }
    Ctrls : EditCpTransShare;         {get shared read lock     }
```

```
    Ctrlw : EditWindowUp;                {up window                }
    Ctrlx : EditCpTransExcl;             {get exclusive write lock}
    Ctrly : EditCpdelwin;                {destroy window           }
    Ctrlz : EditWindowDown;              {down window              }
 One..Nine: EditWindowGoto(ch - Zero); {jump to window #         }
   end;
  Asking := false;
  EditPrcTxt(NotAvailable);              {swap in char processor}
end; {EditO}



procedure EditQ;
{  This routine processes the Ctrl-Q command processing for the
   editor.  The Ctrl-Q command is simply a submenu of commands
   to which window and searching commands may be added. You can remove
   commands from the editor by removing their references in the case
   statement in this procedure.  If all of the commands in a module
   are to be deleted (say, all window commands), then you can simply
   omit that include and comment out the references in this procedure.
}

var
   ch: byte;

begin {EditQ}
  EditZapcmdnam;
  Asking := true;
  EditAppcmdnam ('^Q');
  ch := EditCtrlChar;
  case ch of                                { COMMAND DESCRIPTION     }
      Ctrla : EditCpreplace;                { Find and replace        }
      Ctrlc : EditWindowBottomFile;         { Bottom of window        }
      Ctrld : EditEndLine;                  { End of current line     }
      Ctrle : EditCpClearWindow;            { Empty contents of window}
      Ctrlf : EditCpfind;                   { Find pattern            }
      Ctrlg : EditCpColAdd;                 { Add Column              }
      Ctrlh : EditCpColChg;                 { Change Column           }
      Ctrli : EditToggleAutoindent;         { Toggle autoindent mode  }
      Ctrlj : EditCpColAlign;               { Justify column rt/left  }
      Ctrlk : EditCpColDrop;                { Drop Column             }
      Ctrll : EditCpKeyword;                { Keyword search          }
      Ctrln : EditCpSelect;                 { Relational select       }
      Ctrlo : EditCpJoin;                   { Relational join         }
      Ctrlp : EditCpProject;                { Relational project      }
      Ctrlq : EditQueryClear;               { Clear query definition  }
      Ctrlr : EditWindowTopFile;            { Top of window           }
      Ctrls : EditBeginningLine;            { Beg of current line     }
```

```
      CtrlT : EditCpColSw;                { Swap Columns              }
      CtrlV : EditCpOrder;                { Sort Rows                }
      CtrlW : EditCpUnique;               { Remove duplicate rows    }
      CtrlX : EditCpCreateRel;            { Create Relation          }
      CtrlY : EditDeleteTextRight;        { Delete text to eol       }
      CtrlZ : EditCpClosure;              { Relational closure       }
  end;
Asking := false;
EditPrcTxt(NotAvailable);                 {swap in char processor}
end; {EditQ}
```

19. PULLDOWN MENU SYSTEM

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     pulldown.tt                 Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  This is the pulldown menu system.
*****************************************************************************
}
     { THE FOLLOWING ARE ALL BORLAND PROPRIETARY CODE USED AS IS }

}
procedure SetOnOff(var c; state: boolean);
procedure ClearSubMenu(ScreenPosition,Width:integer);
procedure WriteMain(Address:integer; Mode:Byte);
procedure WriteSelection(Address, Selection, Width, ScreenPosition:integer;
                         Mode:Byte);
procedure InitMainMenu;
procedure ExitPulldown;
function PulldownMenu : VarString;
   begin

      ... the following is the ONLY code in PulldownMenu that is TT-specific

        Case CurrSubmenu of
           1:Case CurrSelection of
               1:pdms := ^K^I;                  { include line in batch }
               2:pdms := ^K^E;                  { exclude line from batch }
               3:pdms := ^K^B;                  { begin defining batch }
               4:pdms := ^K^K;                  { end defining batch }
               5:pdms := ^K^H;                  { clear batch definition }
               6:pdms := ^K^G;                  { read batch from window }
               7:pdms := ^K^N;                  { go to next line in batch }
               8:pdms := ^K^L;                  { go to prior line in batch }
               9:pdms := ^K^Y;                  { delete batch }
              10:pdms := ^K^V;                  { move batch }
              11:pdms := ^K^C;                  { copy batch }
              12:pdms := ^K^P;                  { write batch to window }
```

```
  end;
2:Case CurrSelection of
    1:pdms := ^Q^F;                    { find }
    2:pdms := ^Q^A;                    { find and replace }
    3:pdms := ^Q^L;                    { keyword search }
    4:pdms := ^Q^N;                    { relational select }
    5:pdms := ^Q^P;                    { relational project }
    6:pdms := ^Q^O;                    { relational join }
    7:pdms := ^L;                      { repeat last search }
    8:pdms := ^Q^Q;                    { clear last search spec }
    9:pdms := ^U;                      { relational union }
   10:pdms := ^U;                      { relational difference }
  end;
3:Case CurrSelection of
    1:pdms := ^U;                      { Create index }
    2:pdms := ^U;                      { Remove index }
    3:pdms := ^U;                      { Open index }
    4:pdms := ^U;                      { Close index }
    5:pdms := ^U;                      { Synch index }
    6:pdms := ^U;                      { Synch all indexes }
  end;
4:Case CurrSelection of
    1:pdms := ^O^B;                    { begin transaction }
    2:pdms := ^O^C;                    { commit transaction }
    3:pdms := ^O^A;                    { abort transaction }
    4:pdms := ^O^E;                    { enter transaction }
    5:pdms := ^O^D;                    { leave transaction }
    6:pdms := ^O^S;                    { request share lock }
    7:pdms := ^O^X;                    { request exclusive lock }
    8:pdms := ^O^N;                    { scan transaction }
  end;
5:case CurrSelection of
    1:pdms := ^O^G;                    { goto window }
    2:pdms := ^O^O;                    { open window }
    3:pdms := ^O^Y;                    { close window }
    4:pdms := ^O^J;                    { link window }
    5:pdms := ^Q^E;                    { clear window }
  end;
6:case CurrSelection of
   1:pdms := ^K^O;                     { open file }
   2:pdms := ^K^Z;                     { close file }
   3:pdms := ^K^R;                     { read ascii file }
   4:pdms := ^K^W;                     { write ascii file }
   5:pdms := ^K^X;                     { close all files and exit }
   6:pdms := ^Q^R;                     { top of file }
   7:pdms := ^Q^C;                     { bottom of file }
   8:pdms := ^U;                       { synchronize file }
  end;
```

```
          7:case CurrSelection of
              1:pdms := ^Q^X;                    { Create table }
              2:pdms := ^Q^T;                    { Swap Columns }
              3:pdms := ^Q^J;                    { Align column }
              4:pdms := ^Q^V;                    { Sort table }
              5:pdms := ^Q^G;                    { Add column }
              6:pdms := ^Q^K;                    { Drop column }
              7:pdms := ^Q^H;                    { Change column }
              8:pdms := ^Q^W;                    { remove duplicate rows }
            end;
          8:case CurrSelection of
            1:pdms := ^Q^Z;                      { Closure }
            2:pdms := ^U;                        { Match }
            3:pdms := ^U;                        { Part-whole }
            4:pdms := ^U;                        { Variance }
            5:pdms := ^U;                        { Temporal }
            6:pdms := ^U;                        { Resource }
            7:pdms := ^U;                        { Begin engine }
            8:pdms := ^U;                        { End engine }
            9:pdms := ^U;                        { checKpoint engine }
           10:pdms := ^U;                        { reStart engine }
           end;
        end;
  PullDownMenu := pdms;
}
```

20. UNPREFIXED COMMAND DISPATCHER

```
{
*******************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     disp.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                        under Contract F30602-C-85-0190

Function:  This routine dispatches the first Ctrl character received.
*******************************************************************************
}

procedure EditPrccmd (ch : byte);
{  This routine receives control from EditClsinp to dispatch a built-in
   Editor Toolbox command.
}
begin {EditPrccmd}
  case ch of
    Del   : EditDeleteRightChar;        { Delete character under cursor  }
    Ctrlb : if Curwin^.Formatted then
               EditBackColumn           { Tab to prior relational column }
            else
               EditBackTab;             { Tab to prior word of prior line}
    Ctrlc : EditDownPage;               { Down page                      }
    Ctrld : EditRightChar;              { Right character                }
    Ctrle : EditUpLine;                 { Up line                        }
    Ctrlg : EditDeleteRightChar;        { Delete character under cursor  }
    Ctrlh : EditDeleteLeftChar;         { Destructive backspace          }
    Ctrli : if Curwin^.Formatted then
               EditGotoColumn           { Tab to next relational column  }
            else
               EditTab;                 { Tab to next word of prior line }
    Ctrlj : EditBeginningEndLine;       { Beginning/end of line          }
    Ctrlk : EditK;                      { ^K commands                    }
    Ctrll : EditCpSearchNext;           { Repeat last search command     }
    Ctrlm : EditNewLine;                { New line in text buffer        }
    Ctrln : EditInsertLine;             { Insert line                    }
    Ctrlo : EditO;                      { ^O commands                    }
    Ctrlp : EditCpInsertCtrlChar;       { Insert control character       }
    Ctrlq : EditQ;                      { ^Q commands                    }
```

```
    Ctrlr : EditUpPage;                { Up page                          }
    Ctrls : EditLeftChar;              { Left character                   }
    Ctrlv : EditToggleInsert;          { Toggle insert mode               }
    Ctrlw : EditScrollUp;              { Scroll up                        }
    Ctrlx : EditDownLine;              { Down line                        }
    Ctrly : EditDeleteLine;            { Delete line                      }
    Ctrlz : EditScrollDown             { Scroll down                      }
  end; {case}
  Updcurflag := true;
  EditPrcTxt(NotAvailable);            { this swaps in the normal character }
                                       { processor in case it was swapped out}
end; {EditPrccmd}
```

21. CHARACTER DISPATCHER

```
{
*****************************************************************************
Program:  TT                          Version 2.0              Date: 20 Nov 87
File:     task.tt                     Version 2.00             Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                            Boulder, Colorado 80303

                          under Contract F30602-C-85-0190

Function:  These routines handle all the asynchronous tasks.
*****************************************************************************
}

{All of the following are Borland proprietary code used as is}

{
procedure EditClsinp;
procedure EditBackground;
procedure EditSchedule;
procedure EditSystem;
}
```

22. KEYBOARD DRIVER

```
{
*****************************************************************************
Program:  TT                          Version 2.0          Date: 20 Nov 87
File:     input.tt                    Version 2.00         Date: 20 Nov 87

                    Prepared for Rome Air Development Center
                                    by
                      Linguistic Research Institute, Inc.
                        5600 Arapahoe Road, Suite #206
                           Boulder, Colorado 80303

                      under Contract F30602-C-85-0190

Function:

  This module contains routines which are used to accept input from the
  keyboard and save it on a typeahead buffer. The routines EditKeypressed
  and EditGetinput can be used to determine if input is available on this
  typeahead buffer, and to get the actual characters from the buffer.

*****************************************************************************
}

{All of the following are Borland proprietary code used as is}

{
procedure EditAbort;
procedure EditBreathe;
procedure Pokechr {(Ch : char )};
procedure EditUserpush {(s : varstring)};
}
```

23. INDEX TO PROCEDURES AND FUNCTIONS