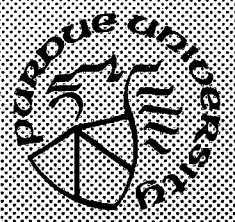
4-1-1990

Source Code for the Nihongo Tutor

Kei Wai Leung
Purdue University

Anthony A. Maciejewski
Purdue University



Source Code for the Nihongo Tutor

Kei Wai Leung
Anthony A. Maciejewski

TR-EE 90-28
April 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

SOURCE CODE FOR THE NIHONGO TUTOR

Kei Wai Leung

Anthony A. Maciejewski

School of Electrical Engineering
Purdue University
West Lafayette, IN 47907

TR-EE 90-28
April 1990

This material is based upon work supported by the National Science Foundation under Grant No. INT-8818039. The Government has certain rights in this material.

This document contains a listing of the Pascal source code for the Nihongo Tutor program. This code is designed to be executed on Macintosh computers under the Kanjitalk operating system. It was written by Kei Wai Leung under the supervision of Anthony A. Maciejewski as part of the Nihongo Tutorial System, a set of programs designed to assist scientists and engineers acquire a reading knowledge of technical Japanese. As this is an ongoing project, portions of this code are under revision. This project is funded by the National Science Foundation under Grant No. INT-8818039. The government has certain rights in this material.

TABLE OF CONTENTS

```
unit CAIGlobals;

{*****************************************************************************}
{                                                                             }
{  This unit contains all global data for the editor/translator. In particular, it specifies  }
{  constants, types, and variables.                                           }
{                                                                             }
{*****************************************************************************}

{*****************************************************************************}
{                                                                             }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All  }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or  }
{ combinations thereof, now known or later developed.                         }
{                                                                             }
{*****************************************************************************}


interface


 uses
  PrintTraps;

 const

  AppleId = 1;
  FileId = 2;
  EditId = 3;
  FunctionId = 4;
  DictionaryId = 5;
  WindowId = 6;

  AppleM = 1;                    { menu constants    }
  FileM = 2;
  EditM = 3;
  FunctionM = 4;
  DictionaryM = 5;
  WindowM = 6;

  FunctionMId = 500;
  SearchOptionMId = 231;         { pop-up menu       }
  DictionaryMId = 501;
  SpecialtyDMId = 232;
  FontMId = 233;
  SizeMId = 234;

  NewCommand = 1;                { file menu constants  }
  OpenCommand = 2;
  CloseCommand = 3;
  SaveCommand = 5;
  SaveAsCommand = 6;
```

```
RevertCommand = 7;
PageSetUpCommand = 9;
PrintCommand = 10;
QuitCommand = 12;

UndoCommand = 1;                    { edit menu constants }
CutCommand = 3;
CopyCommand = 4;
PasteCommand = 5;
ClearCommand = 6;
SelectAllCommand = 8;
FindWhatCommand = 10;
FindCommand = 11;
ReplaceCommand = 12;

EditCommand = 1;                    { function menu constants }
LearnTextCommand = 3;
TranslateCommand = 5;
FullTranslationCommand = 6;
CharacterInfoCommand = 7;
SearchDictCommand = 9;
RadicalInfoCommand = 10;
FontCommand = 11;
SizeCommand = 12;

CharacterDictionary = 1;            { dictionary menu constants }
GeneralDictionary = 2;
SpecialtyDictionary = 3;
GrammarDictionary = 4;

TextW = 1;                          { window menu constants }
TranslationW = 2;
CharacterW = 3;

AboutDId = 128;                     { dialog constants }
OkDId = 129;
OkCanDId = 130;
EditDId = 131;
WaitDId = 133;
FindDId = 134;
YesNoDId = 135;
RadicalDId = 136;
GrammarDId = 137;
SearchDId = 138;

AboutItem = 1;
NumberMenus = 6;
Active = 0;
InActive = 255;
HorizMax = 63;
HorizUnit = 8;
SBarWidth = 16;
UnFamCount = 5;
```

```
DecrementCount = 1;
NodeMax = 1024;
PrimeFactor = 1021;
TableSize = (PrimeFactor - 1);
ArrayMax = 255;
EditLengthMax = 255;
CharLengthMax = 2;
WordLengthMax = 4;
StringLengthMax = 20;
NelsonInputMax = 5500;
RadicalInputMax = 220;
StrokeInputMax = 25;

SBKanji = 1;                    { search options }
SBKatakana = 2;
SBHiragana = 3;
SBNelsonDictionary = 4;
SBRadicals = 5;
SBStrokeCount = 6;
SBEnglish = 7;

Communications = 1;             { specialty dictionary }

HiraLB = 33439;                 { JIS code constants   }
HiraUB = 33521;
KataLB = 33601;
KataUB = 33686;
KanjiLB = 34975;
KanjiUB = 39026;
EnglishLB = 32;
EnglishUB = 126;

CDKanji1LB = 34975;             { char. dictionary file boundaries}
CDKanji1UB = 35648;
CDKanji2LB = 35649;
CDKanji2UB = 36022;
CDKanji3LB = 36023;
CDKanji3UB = 36672;
CDKanji4LB = 36673;
CDKanji4UB = 36994;
CDKanji5LB = 36995;
CDKanji5UB = 37441;
CDKanji6LB = 37442;
CDKanji6UB = 38031;
CDKanji7LB = 38032;
CDKanji7UB = 38464;
CDKanji8LB = 38465;
CDKanji8UB = 39026;
CDFiles = 8;
GDFiles = 2;                    { no. of compound dictionary files}
SDCommFiles = 14;               { no. of specialty dictionary files}
GramFiles = 3;
```

```
  delimiter = '';                    { string constants }
  DoubleDQ = '"';                    { string constants }
  IDString = '.tr';
  PersonId = '.db';
  CDKanji = 'CDKanji';
  CDNelson = 'CDNelson';
  CDRadical = 'CDRadical';
  CDStroke = 'CDStroke';
  GDKanji = 'GDKanji';
  SDComm = 'SDComm';
  Grammar = 'Grammar';
  CharDictionary = 'Character Dictionary';
  GenDictionary = 'Compound Dictionary ';
  SpecDictionary = 'Specialty Dictionary';
  GramDictionary = 'Grammar Dictionary';
  SelectionFile = 'User Selection';
  SysTitle = 'Text';
  TransWTitle = 'Translation';
  CharWTitle = 'Dictionary';
  AppleMark = $14;
  Tab = $09;
  CR = $0D;
  BS = $08;
  SP = $20;
  Null = '';

type

  RWType = (ReadF, WriteF);          { enumerated types }
  DialogType = (AboutD, OkD, OkCanD, YesNoD, RadicalD);
  EditDType = (EditD, GrammarD);
  PhoneticType = (Kanji, Katakana, Hiragana, English);

  WordType = record
    WordBegin, WordLength: Integer;
   end;
  InfoArrayPtr = ^InfoArrayType;     { info array type  }
  InfoArrayType = record
    InfoContent: str255;
    Next: InfoArrayPtr;
   end;
  ChildPtr = ^ChildNode;
  NodePtr = ^Node;
  ChildNode = record                 { children node list }
    Son: NodePtr;
    Next: ChildPtr;
   end;
  Node = record                      { data structure of a node }
    NodeId: integer;
    TEStart, TEEnd: Longint;
    JapaneseWords, EnglishWords, Pronoun: WordType;
    JapInfo, EngInfo: InfoArrayPtr;
    Child: ChildPtr;
```

```
    Parent: NodePtr;
   end;
  NodeArrayType = array[1..NodeMax] of NodePtr;
  HashTableType = array[0..TableSize] of InfoArrayPtr;


 var
  Menus: array[1..NumberMenus] of MenuHandle;
  OptionsMenu, SpecialtyDMenu, FontMenu, SizeMenu: MenuHandle;
  TextRect, TranslationRect, CharacterRect, DestRect, DragRect, GrowLimitRect: Rect;
  DBJap, DBTrans: InfoArrayPtr;
  ArticlePtr, CurSenPtr, OldPtr: NodePtr;
  UnFHashTable, WKHashTable: HashTableType;
  NewNodePtr: NodeArrayType;
  WaitDialog: DialogPtr;
  TextWindow, TranslationWindow, CharacterWindow: WindowPtr;
  CurDictName, DefaultVolName, DelimiterString, SearchString, ReturnString, SpaceString, UserName:
      str255;
  FindString, ReplaceString, FurRef, PronounString, ArrowString, DisplayString, LineString, FontName:
      str255;
  FixedCursor, FullTranslate, EditFunction, DictOpened, TextOpened, PrintStyle: Boolean;
  WordStart, WordEnd, OldSelStart, OldSelEnd, DQCount: Longint;
  SearchOption, SpecialtyDOption, FontOption, SizeOption, GrammarOption: integer;
  CurrentFunction, CurrentNode, CurFunction, CurDictionary, DBCount, DBSize, FontNum, FontSize:
      integer;
  TELines, TEWidth, TransLines, TransWidth, CharLines, CharWidth, DictvRef, vRef, lineHt: integer;
  vTextSB, hTextSB, vTransSB, hTransSB, vCharSB, hCharSB: ControlHandle;
  FileHandle, DictHandle, DBHandle: Handle;
  DBChar: CharsHandle;
  Watch, IBeam: CursHandle;
  TEText, TranslationText, CharacterText: TEHandle;
  UpdateEvent: EventRecord;
  fInfo: FontInfo;
  prRecHandle: THPrint;


implementation

end.
```

```
unit CAIAdjust;

{**************************************************************************}
{                                                                          }
{ This unit adjusts the view of a window by scrolling the text and updating the scroll bars.  }
{ It also provides routines for creating a text handle as well as a window.  }
{                                                                          }
{**************************************************************************}

{**************************************************************************}
{                                                                          }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All  }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or  }
{ combinations thereof, now known or later developed.                      }
{                                                                          }
{**************************************************************************}


interface

 uses
  PrintTraps, CAIGlobals;


 function GetCurrentTime: str255;

 function NumberToString (InNumber: Longint): str255;

 function GetMin (x, y: Longint): Longint;

 function GetMax (x, y: Longint): Longint;

 procedure SetFont (fontNum: integer);

 procedure ChangeFont (theNum, theSize: integer;
          var theWindow: WindowPtr;
          var teh: TEHandle;
          var Lines: integer;
          var vScrollBar: ControlHandle);

 procedure AdjustText (var teh: TEHandle;
          var vScrollBar, hScrollBar: ControlHandle);

 procedure AdjustVScrollMax (var theWindow: WindowPtr);

 procedure FixText (var theWindow: WindowPtr;
          var teh: TEHandle;
          var vScrollBar, hScrollBar: ControlHandle);

 procedure ShowInsertion (var theWindow: WindowPtr;
          var teh: TEHandle;
```

```
            var Lines: integer;
            var vScrollBar, hScrollBar: ControlHandle);

  procedure UpdateWindow (var theWindow: WindowPtr;
            var teh: TEHandle;
            var Lines, Width: integer;
            var vScrollBar, hScrollBar: ControlHandle);

  procedure HandleUpdate (var theEvent: EventRecord);

  procedure ScrollContent (var ScrollBar: ControlHandle;
            theWindow: WindowPtr;
            location: integer;
            where: Point);

  procedure NewTEHandle (var teh: TEHandle;
            fName: str255;
            theWindow: WindowPtr;
            visibleW: Boolean);

  procedure OpenWindow (var theWindow: WindowPtr;
            var vScrollBar, hScrollBar: ControlHandle;
            var teh: TEHandle;
            fName: string;
            theRect: Rect;
            var Lines, Width: integer;
            visibleW: Boolean);


implementation

 { function to get the current time and return it as a string }
 function GetCurrentTime: str255;
  var
   TimeString: str255;
   CurrentTime: Longint;
 begin { GetCurrentTime }
  GetDateTime(CurrentTime);
  NumToString(CurrentTime, TimeString);
  GetCurrentTime := TimeString;
 end; { GetCurrentTime }


{ function to convert a number to a string }
 function NumberToString;
  var
   ResultString: str255;
 begin { NumberToString }
  NumToString(InNumber, ResultString);
  NumberToString := ResultString;
 end; { NumberToString }
```

```
{ function to find the minimum of two numbers }
function GetMin (x, y: Longint): Longint;
begin { NumberToString }
 if (x < y) then
  GetMin := x
 else
  GetMin := y;
end; { NumberToString }


{ function to find the maximum of two numbers }
function GetMax (x, y: Longint): Longint;
begin { NumberToString }
 if (x > y) then
  GetMax := x
 else
  GetMax := y;
end; { NumberToString }


{ procedure to set up the fontsize menu according to the fontNum selected    }
 procedure SetFont (fontNum: integer);
  const
   fontMax = 127;
  var
   TheNum, i, TheSize: integer;
   fontName, FSize, SizeName: str255;
 begin { SetFont }
  TheNum := fontNum;
  FSize := Null;
  if (fontNum = applFont) then
   begin
    GetFontName(fontNum, fontName);
    GetFNum(fontName, TheNum);
   end;
  for i := CountMItems(SizeMenu) downto 1 do    { remove all items }
   DelMenuItem(SizeMenu, i);
  DrawMenuBar;
  for i := 1 to fontMax do                      { add new items }
   if (RealFont(TheNum, i)) then
    FSize := Concat(NumberToString(i), ';', FSize);
  if (FSize <> Null) then
   if (FSize[Length(FSize)] = ';') then
    delete(FSize, Length(FSize), 1);
  InsMenuItem(SizeMenu, FSize, 0);
  DrawMenuBar;
  SizeOption := 1;
  repeat
   GetItem(SizeMenu, SizeOption, SizeName);
   ReadString(SizeName, TheSize);
   if (TheSize <> fontSize) then
    SizeOption := SizeOption + 1
   else
```

```
    CheckItem(SizeMenu, SizeOption, true);
  until (TheSize = fontSize) or (SizeOption > CountMItems(SizeMenu));
 end; { SetFont }


{ procedure to update the character font in the windows }
 procedure ChangeFont (theNum, theSize: integer;
           var theWindow: WindowPtr;
           var teh: TEHandle;
           var Lines: integer;
           var vScrollBar: ControlHandle);
  var
   r: Rect;
   OldPort: GrafPtr;
 begin { ChangeFont }
  FontNum := theNum;
  FontSize := theSize;
  GetPort(OldPort);
  SetPort(theWindow);
  with theWindow^.portRect, teh^^ do
   begin
    txFont := fontNum;
    txSize := fontSize;
    TextFont(fontNum);
    TextSize(fontSize);
    GetFontInfo(fInfo);
    with fInfo do
     begin
      lineHeight := ascent + descent + leading;
      lineHt := ascent + descent + leading;
      fontAscent := ascent;
     end;
    viewRect.bottom := bottom - SBarWidth - 2;
    Lines := (viewRect.bottom - viewRect.top) div lineHeight;
    viewRect.bottom := viewRect.top + Lines * lineHeight;
    destRect := viewRect;
    AdjustVScrollMax(theWindow);                    { adjust vertical scroll bar   }
    SetCtlValue(VScrollBar, 0);
    r := theWindow^.portRect;
    r.right := r.right - SBarWidth + 1;
    r.bottom := r.bottom - SBarWidth + 1;
    InvalRect(r);
    TECalText(teh);
   end;
  SetPort(OldPort);
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows       }
   HandleUpdate(UpdateEvent);
 end; { ChangeFont }


{ procedure to scroll the text horizontally and vertically, if necessary,  }
{ to reflect the current control settings of either scroll bar.            }
 procedure AdjustText;
```

```
  var
    oldScroll, newScroll, change: integer;
 begin { AdjustText }
  with teh^^ do
    begin
      oldScroll := viewRect.top - destRect.top;
      newScroll := GetCtlValue(vScrollBar) * lineHeight;
      change := oldScroll - newScroll;
      if (change <> 0) then                    { scroll text vertically    }
        TEScroll(0, change, teh);
      oldScroll := viewRect.left - destRect.left;
      newScroll := GetCtlValue(hScrollBar) * HorizUnit;
      change := oldScroll - newScroll;
      if (change <> 0) then                    { scroll text horizontally  }
        TEScroll(change, 0, teh)
    end;
 end; { AdjustText }


{ procedure to readjust the vertical text scroll bar value to    }
{ show current text setting and the insertion point.             }
 procedure ShowInsertPt (Lines: integer;
           var teh: TEHandle;
           var vScrollBar, hScrollBar: ControlHandle);
  var
    selLine: integer;
 begin { ShowInsertPt }
  with teh^^ do
    begin
      selLine := 0;
      while (selStart >= lineStarts[selLine]) do       { locate insertion point    }
        selLine := selLine + 1;
      SetCtlValue(vScrollBar, selLine - Lines div 2);  { reset control setting     }
      AdjustText(teh, vScrollBar, hScrollBar);         { adjust the text           }
    end;
 end; { ShowInsertPt }


{ procedure to  return the correct number of lines in the text of a window.      }
 function TotalLines (teh: TEHandle): integer;
  var
    lines: integer;
    txt: CharsHandle;
 begin { TotalLines }
  with teh^^ do
    begin
      lines := nLines;
      txt := CharsHandle(hText);                   { convert text into characters   }
      if (teLength > 0) then
        if (txt^^[teLength - 1] = chr(CR)) then    { check if last char. is a CR    }
          lines := lines + 1;
      TotalLines := lines
    end;
```

```
 end; { TotalLines }


{ procedure to adjust the max control setting for a given vertical scroll bar.    }
 procedure AdjustVScrollMax;
  var
   ctlMax: integer;
 begin { AdjustVScrollMax }
  if (theWindow = TextWindow) then              { compute correct control max   }
   ctlMax := TotalLines(TEText) - TELines
  else if (theWindow = TranslationWindow) then
   ctlMax := TotalLines(TranslationText) - TransLines
  else if (theWindow = CharacterWindow) then
   ctlMax := TotalLines(CharacterText) - CharLines;
  if ctlMax < 0 then
   ctlMax := 0;
  if (theWindow = TextWindow) then              { set correct control max }
   SetCtlMax(vTextSB, ctlMax)
  else if (theWindow = TranslationWindow) then
   SetCtlMax(vTransSB, ctlMax)
  else if (theWindow = CharacterWindow) then
   SetCtlMax(vCharSB, ctlMax);
 end; { AdjustVScrollMax }


{ procedure to resize the window's scroll bar and text rectangle   }
{ when a window is being zoomed in or out.                         }
 procedure FixText;
  var
   firstChar, maxTop, windowHeight, theLine: integer;
 begin { FixText }
  HideControl(vScrollBar);                      { hide controls while updating }
  HideControl(hScrollBar);
  with theWindow^.portRect do                   { update the scroll bars       }
   begin
    MoveControl(vScrollBar, right - (SBarWidth - 1), -1);
    MoveControl(hScrollBar, -1, bottom - (SBarWidth - 1));
    SizeControl(vScrollBar, SBarWidth, (bottom + 1) - (top - 1) - (SBarWidth - 1));
    SizeControl(hScrollBar, (right + 1) - (left - 1) - (SBarWidth - 1), SBarWidth);
   end;
  ShowControl(vScrollBar);
  ShowControl(hScrollBar);
  ValidRect(vScrollBar^^.contrlRect);
  ValidRect(hScrollBar^^.contrlRect);
  with teh^^ do
   begin
    firstChar := lineStarts[GetCtlValue(vScrollBar)];
    viewRect := theWindow^.portRect;
    InsetRect(viewRect, 4, 4);
    with viewRect do                            { reset text rectangle }
     begin
      right := right - (SBarWidth - 1);
      bottom := ((bottom - (SBarWidth - 1)) div lineHeight) * lineHeight;
```

```
        windowHeight := (bottom - top) div lineHeight;
        maxTop := nLines - windowHeight;
       end;
     destRect := viewRect;
     TECalText(teh);
     if (maxTop <= 0) then                    { readjust vertical scroll bar }
      begin
       maxTop := 0;
       HiliteControl(vScrollBar, InActive);
      end
     else
      HiliteControl(vScrollBar, Active);
     SetCtlMax(vScrollBar, maxTop);
     theLine := 0;
     while (lineStarts[theLine + 1] <= firstChar) do
      theLine := theLine + 1;
     SetCtlValue(vScrollBar, theLine);
     AdjustText(teh, vScrollBar, hScrollBar);{ adjust the text on the window }
    end;
  end; { FixText }


{ procedure to scroll the text vertically to show the insertion point of the window.    }
  procedure ShowInsertion;
   var
    top, bottom: integer;
  begin { ShowInsertion }
   AdjustVScrollMax(theWindow);                { adjust vertical control max.}
   AdjustText(teh, vScrollBar, hScrollBar);    { adjust the text              }
   with teh^^ do
    begin
     top := GetCtlValue(vScrollBar);
     bottom := top + Lines;
     if (selStart < lineStarts[top]) then      { locate insertion point       }
      ShowInsertPt(Lines, teh, vScrollBar, hScrollBar)
     else if (selStart >= lineStarts[bottom]) then
      ShowInsertPt(Lines, teh, vScrollBar, hScrollBar);
    end;
  end; { ShowInsertion }


{ procedure to update the size of a window    }
  procedure UpdateWindow (var theWindow: WindowPtr;
            var teh: TEHandle;
            var Lines, Width: integer;
            var vScrollBar, hScrollBar: ControlHandle);
   var
    portR, r: Rect;
    OldCtlVal: integer;
  begin { UpdateWindow }
   portR := theWindow^.portRect;
   OldCtlVal := GetCtlValue(vScrollBar);          { get old control values}
   with portR, teh^^ do
```

```
    begin
     r := viewRect;                                    { update text view rect}
     viewRect.bottom := bottom - SBarWidth - 2;
     viewRect.right := right - SBarWidth - 4;
     Lines := (viewRect.bottom - viewRect.top) div lineHeight;
     viewRect.bottom := viewRect.top + Lines * lineHeight;
     Width := (viewRect.right - viewRect.left) div HorizUnit;
     destRect := viewRect;
     HidePen;                                          { update the scroll bars}
     MoveControl(vScrollBar, right - (SBarWidth - 1), -1);
     MoveControl(hScrollBar, -1, bottom - (SBarWidth - 1));
     SizeControl(vScrollBar, SBarWidth, (bottom + 1) - (top - 1) - (SBarWidth - 1));
     SizeControl(hScrollBar, (right + 1) - (left - 1) - (SBarWidth - 1), SBarWidth);
     AdjustVScrollMax(theWindow);                      { adjust vertical scroll bar   }
     AdjustText(teh, vScrollBar, hScrollBar);          { adjust the text              }
     ShowPen;
     If (GetCtlValue(vScrollBar) = oldCtlVal) then     { if not change in vSBar}
      begin
       If (viewRect.right < r.right) then
        r.right := viewRect.right;
       If (viewRect.bottom < r.bottom) then
        r.bottom := viewRect.bottom;
       ValidRect(r);                                   { prevent redundant update   }
      end;
    end;
  end;{ UpdateWindow }


{ procedure to update the text of a window when the size box is changed          }
  procedure HandleUpdate;
   var
    OldPort: GrafPtr;
    theWindow: WindowPtr;
  begin { HandleUpdate }
   theWindow := WindowPtr(theEvent.Message);
   GetPort(OldPort);
   SetPort(theWindow);
   BeginUpdate(theWindow);                           { update the window    }
   EraseRect(thewindow^.portRect);
   DrawControls(theWindow);                          { draw the controls    }
   DrawGrowIcon(theWindow);                          { draw the size box    }
   If (theWindow = TextWindow) then                  { update the text      }
    TEUpdate(TEText^^.viewRect, TEText)
   else If (theWindow = TranslationWindow) then
    TEUpdate(TranslationText^^.viewRect, TranslationText)
   else If (theWindow = CharacterWindow) then
    TEUpdate(CharacterText^^.viewRect, CharacterText);
   EndUpdate(theWindow);
   SetPort(OldPort);
  end; { HandleUpdate }


{ procedure to handle mouse down in upline, downline, uppage, and }
```

```
{ downpage of a scroll bar.                                    }
 procedure DoScroll (var ScrollBar: ControlHandle;
          PageSize, location: integer);
  var
   Change: integer;
 begin { DoScroll }
  case location of                          { check location of mouse down  }
   InUpButton:
    change := -1;
   InDownButton:
    change := 1;
   InPageUp:
    change := -PageSize;
   InPagedown:
    change := PageSize;
   otherwise
  end;                                      { update control value          }
  SetCtlValue(ScrollBar, GetCtlValue(ScrollBar) + Change);
 end; { DoScroll }


{ function to handle autoscrolling of the clikloop routine of the text.      }
{ The text in the window is scrolled up or down depending upon whether       }
{ the mouse is moved above or below the text.                                }
 function AutoScroll: Boolean;              { a Boolean funtion that returns true    }
  var
   mouse: Point;
   OldClip: RgnHandle;
   Lines: integer;
 begin { AutoScroll }
  AutoScroll := true;
  OldClip := NewRgn;
  GetClip(OldClip);
  ClipRect(FrontWindow^.portRect);
  GetMouse(mouse);                          { get mouse location   }
  if (FrontWindow = TextWindow) then
   begin                                    { scroll up if above window rect }
    if (mouse.v < TEText^^.viewRect.top) then
     DoScroll(vTextSB, (TELines - 1), InUpButton)
    else if (mouse.v > TEText^^.viewRect.bottom) then
     DoScroll(vTextSB, (TELines - 1), InDownButton)
   end                                      { scroll down if below window rect  }
  else if (FrontWindow = TranslationWindow) then
   begin                                    { scroll up if above window rect }
    if (mouse.v < TranslationText^^.viewRect.top) then
     DoScroll(vTransSB, (TransLines - 1), InUpButton)
    else if (mouse.v > TranslationText^^.viewRect.bottom) then
     DoScroll(vTransSB, (TransLines - 1), InDownButton)
   end                                      { scroll down if below window rect  }
  else if (FrontWindow = CharacterWindow) then
   begin                                    { scroll up if above window rect }
    if (mouse.v < CharacterText^^.viewRect.top) then
     DoScroll(vCharSB, (CharLines - 1), InUpButton)
```

```
    else if (mouse.v > CharacterText^^.viewRect.bottom) then
      DoScroll(vCharSB, (CharLines - 1), InDownButton)
    end;                                    { scroll down if below window rect }
  SetClip(OldClip);
  DisposeRgn(OldClip);
  if (FrontWindow = TextWindow) then
    AdjustText(TEText, vTextSB, hTextSB)      { adjust the text       }
  else if (FrontWindow = TranslationWindow) then
    AdjustText(TranslationText, vTransSB, hTransSB)
  else if (FrontWindow = CharacterWindow) then
    AdjustText(CharacterText, vCharSB, hCharSB);
end; { AutoScroll }


{ procedure to handle mouse down in a scroll bar.  First find out    }
{ whether it is vertical or horizontal scroll bar. Then the          }
{ appropriate scroll updating routine is called for the window.      }
 procedure HandleScroll (ScrollBar: ControlHandle;
           location: integer);
 begin { HandleScroll }
  if (location <> 0) then
    if (ScrollBar = vTextSB) or (ScrollBar = hTextSB) then
      if (ScrollBar = vTextSB) then          { update text window scroll bar          }
        DoScroll(vTextSB, (TELines - 1), location)
      else
        DoScroll(hTextSB, (TEWidth div 2), location)
    else if (ScrollBar = vTransSB) or (ScrollBar = hTransSB) then
      if (ScrollBar = vTransSB) then         { update translation window scroll bar  }
        DoScroll(vTransSB, (TransLines - 1), location)
      else
        DoScroll(hTransSB, (TransWidth div 2), location)
    else if (ScrollBar = vCharSB) or (ScrollBar = hCharSB) then
      if (ScrollBar = vCharSB) then          { update character window scroll bar    }
        DoScroll(vCharSB, (CharLines - 1), location)
      else
        DoScroll(hCharSB, (CharWidth div 2), location);
 end; { HandleScroll }


{ procedure to handle mouse down in a scroll bar.  All controls other than the thumb   }
{ are handled by HandleScroll; both call AdjustText to handle adjusting the text.      }
 procedure ScrollContent;
 begin { ScrollContent }
  if (location <> InThumb) then                { check if in thumb }
    location := TrackControl(ScrollBar, where, @HandleScroll)
  else
    location := TrackControl(ScrollBar, where, nil);
  If (theWindow = TextWindow) then             { adjust the text   }
    AdjustText(TEText, vTextSB, hTextSB)
  else if (theWindow = TranslationWindow) then
    AdjustText(TranslationText, vTransSB, hTransSB)
  else if (theWindow = CharacterWindow) then
    AdjustText(CharacterText, vCharSB, hCharSB)
```

```
  end; { ScrollContent }


{ procedure to provide a new text handle for a window  }
  procedure NewTEHandle (var teh: TEHandle;
            fName:  str255;
            theWindow:  WindowPtr;
            visibleW:  Boolean);
  begin { NewTEHandle }
    TEDispose(teh);                                 { new data window content}
    SetPort(theWindow);
    with theWindow^.portRect do
      SetRect(DestRect, 4, 4, (right - SBarWidth - 4), (bottom - SBarWidth - 2));
    teh := TENew(DestRect, DestRect);               { create new text handle  }
    SetClikLoop(@AutoScroll, teh);
    if (fName <> TransWTitle) and (fName <> CharWTitle) then
      teh^^.crOnly := 1;
    TextFont(FontNum);
    TextSize(FontSize);
    if (fName <> Null) and (fName <> TransWTitle) and (fName <> CharWTitle) then
      begin                                         { if open an old file       }
        DisposHandle(teh^^.hText);                  { discard old text chars.   }
        teh^^.hText := FileHandle;                  { read in new text chars.   }
        teh^^.teLength := GetHandleSize(FileHandle);
        TECalText(teh);
        TESetSelect(0, 0, teh);
      end;
    if (fName <> Null) then                         { if open an old file       }
      SetWTitle(theWindow, fName)                   { rename the window         }
    else
      SetWTitle(theWindow, SysTitle);               { set default title         }
    if (visibleW) then                              { make it invisible if required }
      begin
        ShowWindow(theWindow);
        SelectWindow(theWindow);
      end;
  end; { NewTEHandle }


{ procedure to open a window and create text handle associated with it.  }
  procedure OpenWindow;
    var
      r:  Rect;
      behindW:  WindowPtr;
  begin { OpenWindow }
    if visibleW then
      behindW := Pointer(-1)                        { make it the front window   }
    else
      behindW := nil;                               { zoom window procedure definition }
    theWindow := NewWindow(nil, theRect, fName, visibleW, ZoomDocProc, behindW, true, 0);
    teh := TENew(theRect, theRect);                 { create text handle for the window }
    NewTEHandle(teh, fName, theWindow, visibleW);
    with theWindow^.portRect do
```

```
    begin                                   { create controls for window }
      SetRect(r, (right - SBarWidth + 1), -1, (right + 1), (bottom - SBarWidth + 2));
      vScrollBar := NewControl(theWindow, r, Null, true, 0, 0, 0, ScrollBarProc, 0);
      SetRect(r, -1, (bottom - SBarWidth + 1), (right - SBarWidth + 2), (bottom + 1));
      hScrollBar := NewControl(theWindow, r, Null, true, 0, 0, HorizMax, ScrollBarProc, 0);
    end;
  Lines := (DestRect.bottom - DestRect.top) div lineHt;
  Width := (DestRect.right - DestRect.left) div HorizUnit;
  If (visibleW) then
    AdjustVScrollMax(theWindow);
end; { OpenWindow }


end.
```

```
unit CAIDialog;

{*****************************************************************}
{                                                                 }
{ This unit manipulates most of dialogs used by the editor/translator. It extracts dialogs }
{ from the resource file and then displays them with the formal arguments. It returns all  }
{ results that the user responds to the dialog.                   }
{                                                                 }
{*****************************************************************}

{*****************************************************************}
{                                                                 }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All      }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or          }
{ combinations thereof, now known or later developed.             }
{                                                                 }
{*****************************************************************}

Interface

 uses
  PrintTraps, CAIGlobals, CAIAdjust;

 procedure RemoveLastChar (var InString: str255;
           TestString: str255);

 function GetSingleArea (InString, Delimiter: str255;
           Counter: integer): str255;

 procedure SetDText (eDialog: DialogPtr;
           EitemNumber: integer;
           msg: str255);

 procedure DisplayDialog (DType: DialogType;
           errmsg: str255;
           var Result: integer);

 procedure EditDialog (msg: str255;
           EType: EditDType;
           var editmsg: str255;
           var Cancelled: Boolean);

 procedure FindDialog (var Findmsg, Replacemsg: str255;
           var Cancelled: Boolean);

 procedure GrammarDialog (JapString, EngString, DescString: str255;
           var Example: InfoArrayPtr;
           var result: integer);

Implementation
```

```
{ procedure to show an error message when a dialog cannot be open.  }
{ This will happen if the dialog resource is not found.             }
 procedure ShowError (Error: integer);
 begin { ShowError }
  SetTextRect(TranslationRect);
  ShowText;
  writeln('Resource Error while opening a dialog, error id = ', Error);
 end; { ShowError }


{ procedure to activate or inactivate a radial button of a dialog according to the mode }
 procedure MakeRadial (eDialog: DialogPtr;
           EitemNumber, Mode: integer);
  var
   itemType: integer;
   itemHandle: Handle;
   itemBox: Rect;
 begin { MakeRadial }
  GetDItem(eDialog, EitemNumber, itemType, itemHandle, itemBox);
  SetCtlValue(ControlHandle(itemHandle), Mode);   { make radial active or inactive }
 end; { MakeRadial }


{ procedure to activate or inactivate a button of a dialog according to the mode }
 procedure MakeButton (eDialog: DialogPtr;
           EitemNumber, Mode: integer);
  var
   itemType: integer;
   itemHandle: Handle;
   itemBox: Rect;
 begin { MakeButton }
  GetDItem(eDialog, EitemNumber, itemType, itemHandle, itemBox);
  HiliteControl(ControlHandle(itemHandle), Mode);  { make button active or inactive }
 end; { MakeButton }


{ function to remove the last character of a string if it matches the test string. }
 procedure RemoveLastChar (var InString: str255;
           TestString: str255);
  var
   OutString: str255;
 begin { RemoveLastChar }
  OutString := InString;
  If (OutString <> Null) then
   If (OutString[Length(OutString)] = TestString) then
    Delete(OutString, Length(OutString), 1);
  InString := OutString;
 end; { RemoveLastChar }


{ function to find a substring in InString indexed by Counter. InString is a}
```

```
{ combination of all area fields separated by the delimiter. Leading      }
{ spaces are all removed.                                                 }
 function GetSingleArea (InString, Delimiter: str255;
          Counter: integer): str255;
  var
   OutString: str255;
   index, result: integer;
 begin { GetSingleArea }
  OutString := InString;                    { work on a local copy of instring}
  result := Pos(Delimiter, InString);
  if (result <> 0) then                     { check if there exists a delimiter}
   begin
    for index := 1 to (Counter - 1) do
     begin
      result := Pos(Delimiter, InString);   { find first counter-1 items      }
      Delete(InString, 1, result);          { remove these items              }
     end;
    result := Pos(Delimiter, InString);
    if (result = 0) then                    { check if last area }
     result := ArrayMax;
    OutString := Copy(InString, 1, (result - 1));
    if (OutString <> Null) then
     while (OutString[1] = SpaceString) do{ remove leading spaces   }
      Delete(OutString, 1, 1);
   end;
  GetSingleArea := OutString;
 end; { GetSingleArea }


{ procedure to get the text of an item of a dialog.     }
 procedure GetDText (eDialog: DialogPtr;
          EitemNumber: integer;
          var Editmsg: str255);
  var
   itemType: integer;
   itemHandle: Handle;
   itemBox: Rect;
 begin { GetDText }
  GetDItem(eDialog, EitemNumber, itemType, itemHandle, itemBox);
  GetIText(itemHandle, Editmsg);
 end; { GetDText }


{ procedure to set the text of an item of a dialog according to msg }
 procedure SetDText (eDialog: DialogPtr;
          EitemNumber: integer;
          msg: str255);
  var
   itemType: integer;
   itemHandle: Handle;
   itemBox: Rect;
 begin { SetDText }
  GetDItem(eDialog, EitemNumber, itemType, itemHandle, itemBox);
```

```
  SetIText(itemHandle, msg);
 end; { SetDText }


{ procedure to set the example of a dialog according to the content of Example  }
 procedure SetExample (eDialog: DialogPtr;
           EitemNumber: integer;
           var Example: InfoArrayPtr);
 begin { SetExample }
  if (Example <> nil) then
   begin
    SetDText(eDialog, EitemNumber, Example^.InfoContent);
    Example := Example^.Next;
   end
  else
   SetDText(eDialog, EitemNumber, Nuli);
 end; { SetExample }


{ procedure to frame an item in a dialog so that it becomes the default button   }
 procedure FrameItem (eDialog: DialogPtr;
           itemNumber: integer);
  var
   itemType: integer;
   itemHandle: Handle;
   itemBox: Rect;
   oldPenState: PenState;
 begin { FrameItem }
  GetPenState(oldPenState);
  GetDItem(eDialog, itemNumber, itemType, itemHandle, itemBox);
  InsetRect(itemBox, -4, -4);      { set a hilited large enclosing rectangular box }
  PenSize(3, 3);
  FrameRoundRect(itemBox, 16, 16);
  SetPenState(oldPenState);
 end; { FrameItem }


{ procedure to activate a dialog when the following occurs:    }
{ AboutD : about dialog to show the editor/translator credits  }
{ OkD    : Ok dialog to show a message with Ok button          }
{ OkCanD : Ok Cancel dialog to show a message with Ok and      }
{          cancel buttons                                      }
{ YesNoD : Yes No dialog to show a message with yes, no and    }
{          cancel buttons                                      }
{ RadicalD: Radical dialog to show all instances of radicals   }
{ It returns the result which the button no. that the user hits. }
 procedure DisplayDialog;
  var
   item, error: integer;
   eDialog: DialogPtr;
   oldPort: GrafPtr;
   DialogStorage: DialogRecord;
 begin { DisplayDialog }
```

```
  SetCursor(Arrow);
  item := Ok;
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows   }
   HandleUpdate(UpdateEvent);
  GetPort(oldPort);
  ParamText(errmsg, Null, Null, Null);
  if (DType <> AboutD) and (DType <> RadicalD) then
   SysBeep(10);                                    { beep a sound          }
  if (DType = AboutD) then                         { call the right dialog  }
   eDialog := GetNewDialog(AboutDId, @DialogStorage, WindowPtr(-1))
  else if (DType = OkD) then
   eDialog := GetNewDialog(OkDId, @DialogStorage, WindowPtr(-1))
  else if (DType = OkCanD) then
   eDialog := GetNewDialog(OkCanDId, @DialogStorage, WindowPtr(-1))
  else if (DType = YesNoD) then
   eDialog := GetNewDialog(YesNoDId, @DialogStorage, WindowPtr(-1))
  else if (DType = RadicalD) then
   eDialog := GetNewDialog(RadicalDId, @DialogStorage, WindowPtr(-1));
  Error := ResError;
  if (Error = noErr) then
   begin
    SetPort(eDialog);
    FrameItem(eDialog, Ok);
    ModalDialog(nil, item);                        { wait for the user's response   }
    CloseDialog(eDialog);
   end
  else
   ShowError(Error);
  Result := item;                                  { return the result }
  SetPort(oldPort);
 end; { DisplayDialog }


{ procedure to activate an edit dialog to read in some strings that the user types in.  }
{ It will search through the text, translation and character window and put in an       }
{ hilited initial content if there is some text hilited. The sequence of searching is text }
{ window, then translation window, then finally the character window. It stops once     }
{ some hilited contents are found.                                                      }
 procedure EditDialog;
  const
   On = 1;
   Off = 0;
  var
   item, Error, EItemNumber, OldItem: integer;
   eDialog: DialogPtr;
   oldPort: GrafPtr;
   DialogStorage: DialogRecord;
   Paste: Boolean;
 begin { EditDialog }
  SetCursor(Arrow);
  item := 0;
  Editmsg := '                    ';
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows   }
```

```
  HandleUpdate(UpdateEvent);
 Paste := ((FrontWindow = TextWindow) and (TEText^^.SelEnd > TEText^^.SelStart) and ((TEText^^.SelEnd
     - TEText^^.SelStart) <= EditLengthMax));
 If Paste then                                  { search thru a window to copy hilited text to scrap }
  TECopy(TEText)
 else
  begin
   Paste := ((FrontWindow = TranslationWindow) and (TranslationText^^.SelEnd >
     TranslationText^^.SelStart) and ((TranslationText^^.SelEnd - TranslationText^^.SelStart) <=
     EditLengthMax));
   If Paste then
    TECopy(TranslationText)
   else
    begin
     Paste := ((FrontWindow = CharacterWindow) and (CharacterText^^.SelEnd >
     CharacterText^^.SelStart) and ((CharacterText^^.SelEnd - CharacterText^^.SelStart) <=
     EditLengthMax));
     If Paste then
      TECopy(CharacterText);
    end;
  end;
 GetPort(oldPort);
 ParamText(msg, Null, Null, Null);
 If (Etype = EditD) then
  eDialog := GetNewDialog(EditDId, @DialogStorage, WindowPtr(-1))
 else If (Etype = GrammarD) then
  eDialog := GetNewDialog(SearchDId, @DialogStorage, WindowPtr(-1));
 Error := ResError;
 If (Error = noErr) then
  begin
   SetPort(eDialog);
   FrameItem(eDialog, Ok);
   EItemNumber := 4;
   SelIText(eDialog, EitemNumber, 0, EditLengthMax);
   DlgDelete(eDialog);
   If Paste then                                { put in initial content }
    begin
     DlgPaste(eDialog);
     SelIText(eDialog, EitemNumber, 0, TEGetScrapLen);
    end;
   OldItem := GrammarOption + 5;
   if (EType = GrammarD) then
    begin
     MakeRadial(eDialog, OldItem, On);
     if (OldItem = 6) then
      MakeRadial(eDialog, 7, Off)
     else
      MakeRadial(eDialog, 6, Off);
    end;
   while (item <> Ok) and (item <> Cancel) do
    begin
     ModalDialog(nil, item);
     If (item <> Ok) and (item <> Cancel) then
```

```
        if (item <> OldItem) then
          begin
            MakeRadial(eDialog, OldItem, Off);        { deactivate old selection     }
            MakeRadial(eDialog, item, On);          { activate new selected radial}
            OldItem := item;
            GrammarOption := OldItem - 5;
          end;
      end;
      if (item = Ok) then                          { return the result }
       GetDText(eDialog, EitemNumber, Editmsg);
      Cancelled := (item = Cancel);
      CloseDialog(eDialog);
    end
   else
    ShowError(Error);
   SetPort(oldPort);
 end; { EditDialog }


{ procedure to activate a find and change dialog to read some strings. }
{ The strings are the find and the replace strings. Some hilited text in}
{ the text, translation, or character window will be put in the find    }
{ item as its initial content if the actual find argument is null.      }
 procedure FindDialog;
  var
   item, Error, EItemNumber: integer;
   eDialog: DialogPtr;
   oldPort: GrafPtr;
   DialogStorage: DialogRecord;
   Paste: Boolean;
 begin { FindDialog }
  SetCursor(Arrow);
  item := Ok;
  Findmsg := FindString;
  Replacemsg := ReplaceString;
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows   }
   HandleUpdate(UpdateEvent);
  if (FindString = Null) then              { search thru a window to copy hilited text to scrap }
   begin
    Paste := ((FrontWindow = TextWindow) and (TEText^^.SelEnd > TEText^^.SelStart) and
      ((TEText^^.SelEnd - TEText^^.SelStart) <= EditLengthMax));
    if Paste then
     TECopy(TEText)
    else
     begin
      Paste := ((FrontWindow = TranslationWindow) and (TranslationText^^.SelEnd >
      TranslationText^^.SelStart) and ((TranslationText^^.SelEnd - TranslationText^^.SelStart) <=
      EditLengthMax));
      If Paste then
       TECopy(TranslationText)
      else
       begin
        Paste := ((FrontWindow = CharacterWindow) and (CharacterText^^.SelEnd >
```

```
        CharacterText^^.SelStart) and ((CharacterText^^.SelEnd - CharacterText^^.SelStart) <=
        EditLengthMax));
          if Paste then
            TECopy(CharacterText);
        end;
      end;
    end;
  GetPort(oldPort);
  eDialog := GetNewDialog(FindDld, @DialogStorage, WindowPtr(-1));
  Error := ResError;
  if (Error = noErr) then
    begin
      SetPort(eDialog);
      FrameItem(eDialog, Ok);
      EItemNumber := 6;
      SetDText(eDialog, EitemNumber, ReplaceString);
      EItemNumber := 5;                                  { put in initial content }
      SetDText(eDialog, EitemNumber, FindString);
      SelIText(eDialog, EitemNumber, 0, EditLengthMax);
      If ((FindString = Null) and Paste) then
        begin
          DlgPaste(eDialog);
          SelIText(eDialog, EitemNumber, 0, TEGetScrapLen);
        end;
      ModalDialog(nil, item);
      If (item = Ok) then                                { return the result }
        begin
          GetDText(eDialog, 5, Findmsg);                 { get the find string}
          GetDText(eDialog, 6, Replacemsg);              { get the replace string}
        end;
      Cancelled := (item = Cancel);                      { check if cancelled }
      CloseDialog(eDialog);
    end
  else
    ShowError(Error);
  SetPort(oldPort);
end; { FindDialog }


{ procedure to activate a grammar dialog to show grammatical info   }
{ about a Japanese phrase. The information includes its general     }
{ definitions, English descriptions, and examples of usages with    }
{ translation.                                                      }
 procedure GrammarDialog;
  var
    item, Error: integer;
    eDialog: DialogPtr;
    oldPort: GrafPtr;
    DialogStorage: DialogRecord;
 begin { GrammarDialog }
  SetCursor(Arrow);
  item := 3;
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows   }
```

```
    HandleUpdate(UpdateEvent);
  GetPort(oldPort);
  eDialog := GetNewDialog(GrammarDId, @DialogStorage, WindowPtr(-1));
  Error := ResError;
  if (Error = noErr) then
   begin
    SetPort(eDialog);
    FrameItem(eDialog, Ok);
    SetDText(eDialog, 12, JapString);                { put in initial content }
    SetDText(eDialog, 13, EngString);
    SetDText(eDialog, 14, DescString);
    repeat
     SetExample(eDialog, 15, Example);
     SetExample(eDialog, 16, Example);
     if (Example <> nil) then
      MakeButton(eDialog, 3, Active)
     else
      MakeButton(eDialog, 3, InActive);
     ModalDialog(nil, item);
    until (item >= Ok) and (item <= 2);
    result := item;
    CloseDialog(eDialog);
   end
  else
   ShowError(Error);
  SetPort(oldPort);
 end; { GrammarDialog }


end.
```

```
unit CAILinkedList;

{*************************************************************}
{                                                             }
{ This unit deals with creating a generalized tree for an article by building a separate tree }
{ for each separate sentence.  A doubly-linked list is constructed by linking together the }
{ translations for each sentence. The node in this linked list contains information about the }
{ translation for a sentence.  A range of hilited words is searched through this list to locate }
{ the sentence they belong to.  Then in that particular sentence tree, the node in the lowest }
{ level that contains all the information for complete translation is located.  Postorder }
{ traversal is then used to print out all the complete translation for the hilited words. For }
{ just translation, the minimal highest node is chosen.       }
{                                                             }
{*************************************************************}

{*************************************************************}
{                                                             }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or }
{ combinations thereof, now known or later developed.         }
{                                                             }
{*************************************************************}

Interface

 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog;


 function NextLine (ShowRect: Boolean;
          TotalCount: Longint): str255;

 procedure GetNextString (var DBInfo: InfoArrayPtr;
          Delimiter: str255;
          Index: integer;
          var InfoString: str255);

 procedure CheckArea (DBInfo: InfoArrayPtr;
          Delimiter, TestString: str255;
          var Found: Boolean;
          var OutInfoPtr, OldInfoPtr: InfoArrayPtr);

 procedure ReadStr (ShowRect: Boolean;
          TotalCount: Longint;
          var InfoPtr: InfoArrayPtr);

 procedure ReadExample (InfoEnd: integer;
          var InfoPtr: InfoArrayPtr);

 procedure CreateSentenceList (var ArticlePtr: NodePtr;
```

```
        var DBText: TEHandle);

procedure PutInfo (var UserText: TEHandle;
        InfoPtr:  InfoArrayPtr);

procedure AddContent (var CurrentPtr: InfoArrayPtr;
        CharString, ReadingString, Delimiter: str255;
        Counter, Index: integer);

procedure RemoveContent (var Bucket, CurrentPtr, OldPtr: InfoArrayPtr);

procedure UpdateReading (CurrentPtr: InfoArrayPtr;
        ReadingString, Delimiter: str255;
        Decrement: Boolean;
        Counter: integer;
        var OutCount: integer;
        var WordString, OutReading: str255);

procedure UpdateArea (CharString, ReadingString: str255);

function JapOrdinal (JapString: str255;
        index: integer): Longint;

procedure ClearHashTable (var HashTable: HashTableType);

procedure CreateHashTable (var HashTable: HashTableType);

procedure TableToArea (HashTable: HashTableType;
        var Area:  InfoArrayPtr);

procedure GetBucket (HashTable: HashTableType;
        CharString:  str255;
        var BucketNumber: integer);

procedure HashArea (var Area: InfoArrayPtr;
        var HashTable: HashTableType);

procedure LocateWordinNode (SentenceRoot: NodePtr;
        var Node: NodePtr);

procedure LocateWordinSentence (var SentenceRoot: NodePtr;
        var ErrorCode: integer);

procedure DisplayArrow (teh: TEHandle);

procedure PostOrderTraversal (Ptr: NodePtr);

procedure GetPronoun (SentCount, NodeCount: integer;
        CharString:  str255;
        var ReadingString: str255);

procedure SetSelection (Ptr: NodePtr);
```

```
 procedure DisposeLinkedList;

 procedure DisposeInfoContent (var Root: InfoArrayPtr);


implementation

 const
  Left = 20;
  Top = 65;
  Bottom = 85;
  Right = 210;

{ function to find the end node of a tree (for the sentence tree list)}
 function EndNode (Root: NodePtr): NodePtr;
  var
   Ptr: NodePtr;
 begin { EndNode }
  Ptr := Root;
  if (Ptr <> nil) then
   while (Ptr^.Child^.next <> nil) do        { search until the last child   }
    Ptr := Ptr^.Child^.next^.Son;
  EndNode := Ptr;
 end; { EndNode }


{ function to remove the leading characters (before a sequence of spaces) of a string. }
{ example: instring like "nihongo     tutor" would give rise to "tutor"                 }
 function RemoveLeadingChars (InString: str255): str255;
  var
   Index: integer;
 begin { RemoveLeadingChars }
  RemoveLeadingChars := Null;
  if (InString <> Null) then
   begin
    Index := Pos(SpaceString, InString);     { skip all chars until a space is hit   }
    repeat                                   { remove all leading spaces   }
     if ((Index + 1) < Length(InString)) and (InString[Index + 1] = SpaceString) then
      Index := Index + 1;
    until (Index >= Length(InString)) or (InString[Index + 1] <> SpaceString);
    Delete(InString, 1, Index);
   end;
  RemoveLeadingChars := InString;
 end; { RemoveLeadingChars }


{ function to return the next line of characters in the translation data base. }
{ It includes all characters starting from the location indexed by DBCount     }
{ until it hits a carriage return, a delimiter string, or the number of chars  }
{ exceed 255. All trailing spaces are removed.                                  }
 function NextLine (ShowRect: Boolean;
           TotalCount: Longint): str255;
  const
```

```
    DisplayFactor = 5;
  var
    LineString: str255;
    Counter, i: integer;
    found: Boolean;
    HighLiteRect: Rect;
  begin { NextLine }
   NextLine := Null;
   LineString := DisplayString;
   i := 1;
   while (DBChar^^[DBCount] <> ReturnString) and (DBChar^^[DBCount] <> DelimiterString) and (i <= 255)
       do
    begin
     LineString[i] := DBChar^^[DBCount];      { store all characters }
     DBCount := DBCount + 1;
     i := i + 1;
     If (ShowRect) and ((i mod DisplayFactor) = 0) then
      begin                                   { display performance rect  }
       SetRect(HighLiteRect, Left, Top, (Left + LoWord(round(DBCount / TotalCount * (Right - Left)))),
       Bottom);
       FillRect(HighLiteRect, Black);
      end;
    end;
   while (DBChar^^[DBCount] = ReturnString) or (DBChar^^[DBCount] = DelimiterString) do
    DBCount := DBCount + 1;                   { skip blank lines and delimiters }
   repeat
    Counter := Length(LineString);
    If (Counter > 0) then
     begin
      found := (LineString[Counter] = ' ');   { check if any trailing spaces }
      if found then
       Delete(LineString, Counter, 1);        { remove trailing space      }
     end;
   until (Counter = 0) or (not found);
   NextLine := LineString;
  end; { NextLine }


{ procedure to read in some characters and store it into an info array node. }
 procedure ReadInfo (var DBInfo: InfoArrayPtr;
           InfoString: str255);
 begin { ReadInfo }
  New(DBInfo);                                { create new storage  }
  DBInfo^.InfoContent := InfoString;          { store the string    }
  DBInfo^.Next := nil;
 end; { ReadInfo }


{ procedure to get a substring pointed by DBinfo, if not nil.  If the string }
{ is consisted of a series of substrings separated by some delimiters,       }
{ the one indexed by Index will be retrieved. Result is null if input is null.}
{ DBInfo is advanced to point to the next slot.                              }
 procedure GetNextString (var DBInfo: InfoArrayPtr;
```

```
            Delimiter: str255;
            Index: integer;
            var InfoString: str255);
 begin { GetNextString }
  InfoString := Null;
  if (DBInfo <> nil) then
    begin                                   { output the string     }
      InfoString := GetSingleArea(DBInfo^.InfoContent, Delimiter, Index);
      DBInfo := DBInfo^.Next;               { point to next string  }
    end;
 end; { GetNextString }


{ procedure to find whether the string teststring is in a certain area  }
{ pointed by DBInfo. If it is found, OutInfoPtr will point to the location }
{ where it is, and OldInfoPtr will point to its parent.                 }
 procedure CheckArea (DBInfo: InfoArrayPtr;
            Delimiter, TestString: str255;
            var Found: Boolean;
            var OutInfoPtr, OldInfoPtr: InfoArrayPtr);
  var
    InfoPtr, CurrentInfoPtr: InfoArrayPtr;
    NodeChar: str255;
 begin { CheckArea }
  CurrentInfoPtr := nil;
  OldInfoPtr := DBInfo;                 { initialize to point to the head}
  InfoPtr := DBInfo;
  Found := false;
  repeat
    CurrentInfoPtr := InfoPtr;          { keep track of the location  }
    GetNextString(InfoPtr, Delimiter, 1, NodeChar);
    Found := ((IUEqualString(NodeChar, TestString) = 0) and (NodeChar <> Null));
    if (not Found) then
      OldInfoPtr := CurrentInfoPtr;     { update the parent pointer   }
  until (NodeChar = Null) or Found;
  OutInfoPtr := CurrentInfoPtr;
 end; { CheckArea }


{ procedure to read into an info array a variable number of }
{ contents. More strings will be used if it does not fit in  }
{ only one string. The string should be followed by a        }
{ delimiter string, a space and a number. A negative no.     }
{ signifies the end of the series.                           }
 procedure ReadStr (ShowRect: Boolean;
            TotalCount: Longint;
            var InfoPtr: InfoArrayPtr);
  var
    HeadPtr, LastPtr, Ptr: InfoArrayPtr;
    ArrayId: integer;
 begin { ReadStr }
  LineString := NextLine(ShowRect, TotalCount);
  ReadInfo(Ptr, LineString);            { store content into info array   }
```

```
  HeadPtr := Ptr;
  LineString := NextLine(false, 1);     { check if no more array  }
  ReadString(LineString, ArrayId);
  while (ArrayId > 0) do
   begin
    LastPtr := Ptr;                     { connected to its parent  }
    Ptr := Ptr^.Next;
    LineString := NextLine(ShowRect, TotalCount);
    ReadInfo(Ptr, LineString);          { store more info arrays  }
    LastPtr^.Next := Ptr;
    LineString := NextLine(ShowRect, TotalCount); { check if no more array  }
    ReadString(LineString, ArrayId);
   end;
  InfoPtr := HeadPtr;
 end; { ReadStr }


{ procedure to read into an info array a variable number of }
{ examples.  More strings will be used if it does not fit in  }
{ only one string.                                            }
 procedure ReadExample (InfoEnd: integer;
           var InfoPtr: InfoArrayPtr);
  var
   HeadPtr, LastPtr, Ptr: InfoArrayPtr;
 begin { ReadExample }
  ReadInfo(Ptr, NextLine(false, 1));   { store content into info array  }
  HeadPtr := Ptr;
  while (DBCount < InfoEnd) do
   begin
    LastPtr := Ptr;                     { connected to its parent  }
    Ptr := Ptr^.Next;
    ReadInfo(Ptr, NextLine(false, 1));{ store more info arrays  }
    LastPtr^.Next := Ptr;
   end;
  InfoPtr := HeadPtr;
 end; { ReadExample }


{ procedure to create a tree consisting of translation information }
{ for a sentence. It uses the node space from the array           }
{ NewNodePtr. CurrentNode always points to next available slot.   }
{ If the no. of nodes requested for building the tree exceed the  }
{ array limit, the entire will terminate.                         }
 procedure CreateSentenceTree (var Root: NodePtr;
           NumberOfNodes: integer);
  var
   Number, i, j, NodeNumber, result: integer;
   Ptr: ChildPtr;
 begin { CreateSentenceTree }
  if ((CurrentNode + NumberofNodes - 1) > NodeMax) then
   begin                                { check memory limit  }
    DisplayDialog(OkD, 'Error -- not enough internal memory!!', result);
    ExitToShell;
```

```
    end;
   for i := CurrentNode to (CurrentNode + NumberofNodes - 1) do
    New(NewNodePtr[i]);                          { initialize before storage }
   for i := CurrentNode to (CurrentNode + NumberofNodes - 1) do
    with NewNodePtr[i]^ do
     begin                                       { read info. of a node   }
      LineString := NextLine(true, DQCount);
      ReadString(LineString, NodeId, TEStart, TEEnd, JapaneseWords.WordBegin,
       JapaneseWords.WordLength, EnglishWords.WordBegin, EnglishWords.WordLength, Pronoun.WordBegin,
       Pronoun.WordLength, Number);
      for j := 1 to (5 * 2) do                   { discard numbers read}
       LineString := RemoveLeadingChars(LineString);
      New(Child);
      Ptr := Child;                              { build children list     }
      while (Number > 0) and ((CurrentNode + Number - 1) <= NodeMax) do
       begin
        Ptr^.Son := NewNodePtr[CurrentNode + Number - 1];
        ReadString(LineString, Number);
        LineString := RemoveLeadingChars(LineString);
        if (Number > 0) then                     { a negative no. implies no more children   }
         begin
          New(Ptr^.next);                        { connect to its children}
          Ptr := Ptr^.next;
         end;
       end;
      if (Ptr = Child) then
       Ptr^.Son := nil;
      Ptr^.next := nil;
      ReadString(LineString, Number);            { point to its parent}
      Parent := NewNodePtr[CurrentNode + Number - 1];
     end;
   CurrentNode := CurrentNode + NumberofNodes;   { update currentnode   }
   root := NewNodePtr[CurrentNode - 1];          { return the root of the sentence }
  end; { CreateSentenceTree }


{ procedure to add a sentence to the list of sentences. If the tree  }
{ is empty, the root pointer ArticlePtr will points to this new      }
{ sentence. Otherwise, the new sentence is appended to the end of }
{ the sentence list.                                                  }
  procedure AddSentence (sentenceId: integer);
   var
    NewPtr, LastNode: NodePtr;
    NumberOfNodes: integer;
  begin { AddSentence }
   New(NewPtr);                          { create new sentence node   }
   New(NewPtr^.Child);
   NewPtr^.Child^.next := nil;
   If (ArticlePtr = nil) then
    begin                                { create first node when list is empty }
     ArticlePtr := NewPtr;
     ArticlePtr^.Parent := NewPtr;
    end
```

```
    else
      begin                          { append the new sentence to the sent. list }
        LastNode := EndNode(ArticlePtr);
        New(LastNode^.Child^.next);
        LastNode^.Child^.next^.Son := NewPtr;
        LastNode^.Child^.next^.next := nil;
        NewPtr^.Parent := LastNode;  { connect to its parent }
      end;
    NewPtr^.NodeId := sentenceId;
    LineString := NextLine(true, DQCount);      { read sentence boundary in text window  }
    ReadString(LineString, NewPtr^.TEStart, NewPtr^.TEEnd);
    ReadStr(true, DQCount, NewPtr^.JapInfo); { read Japanese information  }
    ReadStr(true, DQCount, NewPtr^.EngInfo); { read translation information}
    LineString := NextLine(false, 1);
    ReadString(LineString, NumberOfNodes);
    if (NumberOfNodes > 0) then
      CreateSentenceTree(NewPtr^.Child^.Son, NumberOfNodes);
{ Each node in the sentence linked list points to its tranlsation subtree }
  end; { AddSentence }


{ procedure to create a list of sentences. If a positive number follows}
{ all the information about a sentence, that means there exists some  }
{ more sentence data. If the number is negative, the sentence list is  }
{ complete.                                                            }
  procedure CreateSentenceList;
    var
      sentenceId: integer;
      SearchName: str255;
  begin { CreateSentenceList }
    DBText := TENew(DestRect, DestRect);
    DisposHandle(DBText^^.hText);
    DBText^^.hText := DBHandle;                  { read in db text chars.}
    DBText^^.teLength := GetHandleSize(DBHandle);
    TECalText(DBText);
    SearchName := Concat(DoubleDQ, ReturnString);
    DQCount := Munger(DBText^^.hText, 0, POINTER(ord(@SearchName) + 1), Length(SearchName), nil, 0);
    DBChar := TEGetText(DBText);                 { chars handle that store all text contents }
    DBCount := 0;
    DBSize := DBText^^.teLength;
    repeat
      LineString := NextLine(false, 1);
      ReadString(LineString, sentenceId);        { get next sentence id  }
      If (sentenceId > 0) then                   { check if more sentence  }
        begin
          SetDText(WaitDialog, 1, Concat('Configuring Sentence: ', LineString));
          AddSentence(sentenceId);               { append to sentence list  }
        end;
    until (sentenceId <= 0);
  end; { CreateSentenceList }


{ procedure to write out information content of an info array InfoPtr }
```

```
{ to the user text. If there are more than one string, the first one      }
{ will be followed by a number one, second one by two, and so on. The}
{ last one will be trailed by a negative number. If the array is empty, }
{ a single space will be outputted.                                       }
 procedure PutInfo (var UserText: TEHandle;
          InfoPtr: InfoArrayPtr);
  var
   DataString: str255;
   Ptr: InfoArrayPtr;
   i: integer;
 begin { PutInfo }
  if (InfoPtr <> nil) then
   begin
    i := 1;
    Ptr := InfoPtr;
    repeat
     DataString := Ptr^.InfoContent;      { write the content    }
     TEInsert(POINTER(ord(@DataString) + 1), Length(DataString), UserText);
     if (Ptr^.Next = nil) then            { signify the last string}
      DataString := Concat(DelimiterString, SpaceString, NumberToString(-1), ReturnString)
     else
      DataString := Concat(DelimiterString, SpaceString, NumberToString(i), ReturnString);
     TEInsert(POINTER(ord(@DataString) + 1), Length(DataString), UserText);
     i := i + 1;
     Ptr := Ptr^.Next;                    { write out next string }
    until (Ptr = nil);
   end
  else
   begin                                  { put out a space instead of nothing  }
    DataString := Concat(SpaceString, DelimiterString, SpaceString, NumberToString(-1), ReturnString);
    TEInsert(POINTER(ord(@DataString) + 1), Length(DataString), UserText);
   end;
 end; { PutInfo }


{ procedure to add a new content to the beginning of an area.    }
{ if index =1, information is added to the well-known area       }
{     index = 2, information is added to the unfamiliar area     }
{ It wont add the content if it is already found in the info     }
{ array. CharString is the key for searching.                    }
 procedure AddContent (var CurrentPtr: InfoArrayPtr;
          CharString, ReadingString, Delimiter: str255;
          Counter, Index: integer);
  var
   NewAreaPtr, NextPtr, TempPtr, OldPtr: InfoArrayPtr;
   NewString: str255;
   Found: Boolean;
 begin { AddContent }
  CheckArea(CurrentPtr, Delimiter, CharString, Found, TempPtr, OldPtr);
  if (not Found) then          { do nothing if found}
   begin
    NewString := Null;
    if (Index = 1) then        { content is charstring and reading  }
```

```
      NewString := Concat(CharString, Delimiter, ReadingString, Delimiter)
     else if (Index = 2) then   { content is charstring, counter, reading  }
      NewString := Concat(CharString, Delimiter, NumberToString(Counter), Delimiter, ReadingString,
       Delimiter);
     ReadInfo(NewAreaPtr, NewString);
     NextPtr := CurrentPtr;     { add to the beginning of currentptr }
     CurrentPtr := NewAreaPtr;
     CurrentPtr^.Next := NextPtr;
    end;
 end; { AddContent }


{ procedure to remove the content pointed to by currentptr.        }
 procedure RemoveContent (var Bucket, CurrentPtr, OldPtr: InfoArrayPtr);
  var
   NextPtr: InfoArrayPtr;
 begin { RemoveContent }
  NextPtr := CurrentPtr^.Next;
  if (NextPtr = nil) then                { if only one item   }
   begin                                 { set to nil if no more item}
    if (Bucket^.Next = nil) then
     Bucket := nil;
    OldPtr^.Next := nil;
   end
  else if (OldPtr <> CurrentPtr) then{ point to next item }
   OldPtr^.Next := NextPtr
  else {if (index = 1) then   }
   Bucket := NextPtr;                    { the item to be removed is the first element, skip it }
  Dispose(CurrentPtr);
 end; { RemoveContent }


{ procedure to update the reading of a character or word in the unfamilar area  }
{  pointed by currentptr. It returns the word/char of the string in wordstring, }
{ and all readings in newstring. If decrement is false, the access count is reset }
{ to unfamcount; otherwise, it is decremented by an amount of counter. The      }
{ reading is added to the newstring if it is not there.                         }
 procedure UpdateReading (CurrentPtr: InfoArrayPtr;
           ReadingString, Delimiter: str255;
           Decrement: Boolean;
           Counter: integer;
           var OutCount: integer;
           var WordString, OutReading: str255);
  var
   SearchString, NumCount, NewString: str255;
   OldPtr: InfoArrayPtr;
   result: integer;
 begin { UpdateReading }
  OldPtr := CurrentPtr;
  GetNextString(CurrentPtr, DelimiterString, 1, NewString);
  WordString := GetSingleArea(NewString, Delimiter, 1);
  Delete(NewString, 1, Length(WordString));    { get all readings   }
  NumCount := GetSingleArea(NewString, Delimiter, 2);
```

```
    Delete(NewString, 1, (Length(NumCount) + 1));
    OutReading := NewString;
    ReadString(NumCount, OutCount);                    { get counter value }
    If (not Decrement) then                            { reset access count}
     NewString := Concat(WordString, Delimiter, NumberToString(UnFamCount), NewString)
    else
     NewString := Concat(WordString, Delimiter, NumberToString(OutCount - Counter), NewString);
    if (ReadingString <> Null) then
     begin
      SearchString := Concat(Deiimiter, ReadingString, Delimiter);
      result := Pos(SearchString, NewString);
      If (result <= 0) then                            { add more reading }
       begin
        NewString := Concat(NewString, ReadingString, Delimiter);
        OutReading := Concat(OutReading, ReadingString, Delimiter);
       end;
     end;
    OldPtr^.InfoContent := NewString;
  end; { UpdateReading }


{ procedure to update the user information on the well-known or unfamiliar  }
{ area according to the charstring. If the charstring is already in the     }
{ unfamiliar area, its access count is reset to unfamcount. If it is in the well}
{ known area, it is removed from it and added to the unfamiliar area.       }
{ Otherwise, it is added to the unfamiliar area.                            }
  procedure UpdateArea (CharString, ReadingString: str255);
    const
      Delimiter = ';';
    var
      CurrentPtr, OldPtr: InfoArrayPtr;
      WordString, NewString: str255;
      Found: Boolean;
      NumCount, BucketNumber: integer;
  begin { UpdateArea }
    GetBucket(UnFHashTable, CharString, BucketNumber);
    CheckArea(UnFHashTable[BucketNumber], Delimiter, CharString, Found, CurrentPtr, OldPtr);
    If Found then
     UpdateReading(CurrentPtr, ReadingString, Delimiter, false, 0, NumCount, WordString, NewString)
    else
     AddContent(UnFHashTable[BucketNumber], CharString, ReadingString, Delimiter, UnFamCount, 2);
    GetBucket(WKHashTable, CharString, BucketNumber);
    CheckArea(WKHashTable[BucketNumber], Delimiter, CharString, Found, CurrentPtr, OldPtr);
    if (Found) then                                    { check if in well-known area }
     RemoveContent(WKHashTable[BucketNumber], CurrentPtr, OldPtr); { move from wk to unf area  }
  end; { UpdateArea }


{ procedure to convert a Japanese character into its ordinal value ( a long integer).   }
  function JapOrdinal (JapString: str255;
                       index: integer): Longint;
    var
      OneByteValue, StringCode: Longint;
```

```
 begin { JapOrdinal }
  OneByteValue := 256;          { multiply the first byte by 256 and add to the second   }
  StringCode := ord(JapString[index]) * OneByteValue + ord(JapString[index + 1]);
  JapOrdinal := StringCode;
 end; { JapOrdinal }


{ procedure to clear the content of the hash table }
 procedure ClearHashTable (var HashTable: HashTableType);
  var
   i: integer;
 begin { ClearHashTable }
  for i := 0 to TableSize do
   if (HashTable[i] <> nil) then
    DisposeInfoContent(HashTable[i]);
 end; { ClearHashTable }


{ procedure to create an empty hash table }
 procedure CreateHashTable (var HashTable: HashTableType);
  var
   i: integer;
 begin { CreateHashTable }
  for i := 0 to TableSize do
   HashTable[i] := nil;
 end; { CreateHashTable }


{ procedure to find the next non-empty bucket in the hash table}
 procedure GetNextBucket (HashTable: HashTableType;
           var found: Boolean;
           var Index: integer;
           var  Bucket: InfoArrayPtr);
  var
   i: integer;
 begin { GetNextBucket }
  i := Index;
  found := false;
  while (not found) and (i <= TableSize) do
   if (HashTable[i] = nil) then
    i := i + 1                  { find first non-empty bucket }
   else
    found := true;
  if (i <= TableSize) and (found) then
   begin
    Bucket := HashTable[i];
    Index := i;
   end;
 end; { GetNextBucket }


{ procedure to transfer the content of a hash table to an info array   }
 procedure TableToArea (HashTable: HashTableType;
```

```
            var Area: InfoArrayPtr);
  var
    i: integer;
    StartPtr, Bucket, CurrentPtr: InfoArrayPtr;
    found: Boolean;
 begin { TableToArea }
  i := 0;
  StartPtr := nil;
  GetNextBucket(HashTable, found, i, Bucket);
  if found then
    begin
     StartPtr := Bucket;
     repeat
       CurrentPtr := Bucket;
       while (CurrentPtr^.Next <> nil) do    { find last entry in the chain  }
         CurrentPtr := CurrentPtr^.Next;
       i := i + 1;
       GetNextBucket(HashTable, found, i, Bucket);
       if found then
         CurrentPtr^.Next := Bucket;          { connect with last non-empty bucket  }
     until (i > TableSize) or (not found);
    end;
  Area := StartPtr;
 end; { TableToArea }


{ procedure to get the bucket pointer in the hash table  }
 procedure GetBucket (HashTable: HashTableType;
            CharString: str255;
            var BucketNumber: integer);
  var
    OrdinalSum: longint;
    i: integer;
 begin { GetBucket }
  OrdinalSum := 0;
  for i := 1 to (Length(CharString) div 2) do      { add ordinal value of each character  }
    OrdinalSum := OrdinalSum + JapOrdinal(CharString, 2 * i - 1);
  BucketNumber := OrdinalSum mod PrimeFactor;  { use hash function }
 end; { GetBucket }


{ procedure to hash the content of an area into a hash table }
 procedure HashArea (var Area: InfoArrayPtr;
            var HashTable: HashTableType);
  const
    Delimiter = ';';
  var
    AreaPtr, CurrentPtr, OldPtr, NewAreaPtr: InfoArrayPtr;
    TextString, CharString: str255;
    Found: Boolean;
    BucketNumber: integer;
 begin { HashArea }
  AreaPtr := Area;
```

```
  while (AreaPtr <> nil) do
   begin
    GetNextString(AreaPtr, DelimiterString, 1, TextString);
    CharString := GetSingleArea(TextString, Delimiter, 1);
    GetBucket(HashTable, CharString, BucketNumber);
    CheckArea(HashTable[BucketNumber], Delimiter, CharString, Found, CurrentPtr, OldPtr);
    If (not Found) then                    { do nothing if found}
     begin
      ReadInfo(NewAreaPtr, TextString);
      if (CurrentPtr <> nil) then
       CurrentPtr^.Next := NewAreaPtr     { add to the end of the chain  }
      else
       HashTable[BucketNumber] := NewAreaPtr;
     end;
   end;
  DisposeInfoContent(Area);
 end; { HashArea }


{ function to search through the children of a node under the specified condition.}
{ condition = 1 : check if hilited words are in exactly one child of the node.    }
{ condition = 2 : check where the hilited words start in any one of the children. }
{ condition = 3 : check where the hilited words end in any one of the children.   }
{ SearchChild is true if that node is not found; false otherwise. If found, that  }
{ child is also returned.                                                         }
 function SearchChild (Condition: integer;
          var Node: NodePtr): Boolean;
  var
   search: Boolean;
   Ptr: ChildPtr;
 begin { SearchChild }
  Ptr := Node^.Child;
  search := not ((Ptr^.Son = nil) and (Ptr^.next = nil)); { skip if at leaf ndoe   }
  while (search) and (Ptr <> nil) do                      { stop if no more child}
   begin
    case condition of                                     { check the condition  }
     1:
      search := not ((Ptr^.Son^.TEStart <= WordStart) and (Ptr^.Son^.TEEnd >= WordEnd));
     2:
      search := not (Ptr^.Son^.TEEnd >= WordStart);
     3:
      search := not (Ptr^.Son^.TEEnd >= WordEnd);
     otherwise
    end;
    if (not search) then
     node := Ptr^.Son       { return the child that contains the hilited words }
    else
     Ptr := Ptr^.next;                                    { search next child }
    if (Ptr = nil) then          { not found if all the children are searched }
     search := true;
   end;
  SearchChild := search;
 end; { SearchChild }
```

```
{ procedure to locate a range of words in a particular sentence    }
 procedure LocateWordinNode;
  var
   Ptr, TempPtr: NodePtr;
   found: Boolean;
 begin { LocateWordinNode }
  Ptr := SentenceRoot;
  found := (Ptr^.TEStart = WordStart) and (Ptr^.TEEnd = WordEnd);
  while (not found) and (Ptr^.Child^.Son <> nil) and (Ptr^.Child^.next <> nil) do
   found := SearchChild(1, Ptr);
  Node := Ptr;

   { adjust selection range to include all related characters into the  }
   {  selection range even though not hilited                          }
  TempPtr := Node;
  found := (TempPtr^.TEStart = WordStart) and (TempPtr^.TEEnd = WordEnd);
  while (not found) and (TempPtr^.Child^.Son <> nil) and (TempPtr^.Child^.next <> nil) do
   found := SearchChild(2, TempPtr);
  WordStart := TempPtr^.TEStart;          { get new WordStart   }
  TempPtr := Node;
  found := (TempPtr^.TEStart = WordStart) and (TempPtr^.TEEnd = WordEnd);
  while (not found) and (TempPtr^.Child^.Son <> nil) and (TempPtr^.Child^.next <> nil) do
   found := SearchChild(3, TempPtr);
  WordEnd := TempPtr^.TEEnd;              { get new WordEnd     }
 end; { LocateWordinNode }


{ procedure to locate a range of words in some sentence of an article    }
{ assume that a selection is made within a complete sentence, that is    }
{ selection would not cross sentence boundary.  If hilited range exceeds }
{the limit of one sentence, translation is not provided at all.          }
 procedure LocateWordinSentence;
  var
   Ptr: NodePtr;
   found: Boolean;
 begin { LocateWordinSentence }
  found := false;
  if (CurSenPtr = nil) then
   CurSenPtr := ArticlePtr;
  Ptr := CurSenPtr;                       { search from current sentence  }
  while (Ptr^.Parent <> Ptr) and (Ptr^.TEStart > TEText^^.SelStart) do
   Ptr := Ptr^.Parent;                    { move backward until it includes highlited text  }
{    Ptr := ArticlePtr;}
  ErrorCode := 0;
  while (not found) do
   if ((Ptr^.TEEnd + 1) > TEText^^.SelStart) then  { check each node until found     }
    found := true
   else if (Ptr^.Child^.next <> nil) then
    Ptr := Ptr^.Child^.next^.Son
   else
    begin
```

```
      found := true;                                { error happens since outside range }
      Ptr := nil;
      ErrorCode := 1;
     end;
  if (Ptr <> nil) then
   begin
    CurSenPtr := Ptr;                               { CurSenPtr points to the sentence hilited  }
    WordStart := TEText^^.SelStart - Ptr^.TEStart;{ compute relative selection range in a sentence }
    if ((TEText^^.SelEnd - 1) <= Ptr^.TEEnd) then
     WordEnd := (TEText^^.SelEnd - 1) - Ptr^.TEStart
    else
     begin
      Ptr := nil;
      ErrorCode := 2;
     end;
   end;
  if (Ptr <> nil) then
   begin
    DBJap := CurSenPtr^.JapInfo;                    { point to japanese and english info  }
    DBTrans := CurSenPtr^.EngInfo;
    SentenceRoot := CurSenPtr^.Child^.Son           { SentenceRoot points to the sentence tree }
   end
  else
   SentenceRoot := nil;                             { return nil if not found}
 end; { LocateWordinSentence }


{ procedure to display an arrow in the specified window          }
 procedure DisplayArrow;
 begin { DisplayArrow }
  TEInsert(Pointer(Ord(@ArrowString) + 1), Length(ArrowString), teh);
 end; { DisplayArrow }


{ function to find some words in the info array pointed by DBPtr specified   }
{ by DBWord. If it is empty, it returns the string furref. Otherwise, it     }
{ locates the info string and extract the substring from it.                 }
 function DisplayWords (DBWord: WordType;
          DBPtr: InfoArrayPtr): str255;
  var
   location, Count, i: integer;
   InfoWord: WordType;
   InfoPtr: InfoArrayPtr;
   WordString: str255;
 begin { DisplayWords }
  InfoWord := DBWord;
  InfoPtr := DBPtr;
  WordString := Null;
  if (InfoWord.WordBegin <= 0) and (InfoWord.WordLength <= 0) then
   WordString := FurRef                                  { display further ref.  }
  else if (InfoWord.WordLength <= ArrayMax) then
   begin
    Count := (InfoWord.WordBegin - 1) div ArrayMax;      { check if end of words}
```

```
      for i := 1 to Count do
       if (InfoPtr^.next <> nil) then
         InfoPtr := InfoPtr^.next;                                  { point to correct array}
       location := (InfoWord.WordBegin - 1) mod ArrayMax + 1;   { get word array index}
       WordString := Copy(InfoPtr^.InfoContent, location, InfoWord.WordLength);
      end;
    DisplayWords := WordString;
  end; { DisplayWords }


{ procedure to show translation of some hilited Japanese words    }
{ in the translation text.                                        }
  procedure ShowTranslation (Ptr: NodePtr);
    var
      WordString, CharString, ReadingString: str255;
  begin { ShowTranslation }
    WordString := DisplayWords(Ptr^.JapaneseWords, DBJap);
    CharString := WordString;
    TEInsert(Pointer(Ord(@WordString) + 1), Length(WordString), TranslationText);
    ShowInsertion(TranslationWindow, TranslationText, TransLines, vTransSB, hTransSB);
    if (Ptr^.Pronoun.WordBegin > 0) and (Ptr^.Pronoun.WordLength > 0) then
      begin                                  { give the reading  }
        ReadingString := DisplayWords(Ptr^.Pronoun, DBJap);
        WordString := Concat(PronounString, ReadingString, ')');
        TEInsert(Pointer(Ord(@WordString) + 1), Length(WordString), TranslationText);
        ShowInsertion(TranslationWindow, TranslationText, TransLines, vTransSB, hTransSB);
      end
    else
      ReadingString := Null;
    DisplayArrow(TranslationText);      { give the translation  }
    ShowInsertion(TranslationWindow, TranslationText, TransLines, vTransSB, hTransSB);
    WordString := DisplayWords(Ptr^.EnglishWords, DBTrans);
    TEInsert(Pointer(Ord(@WordString) + 1), Length(WordString), TranslationText);
    TEKey(chr(CR), TranslationText);
    ShowInsertion(TranslationWindow, TranslationText, TransLines, vTransSB, hTransSB);
    if (Ptr^.Child^.Son = nil) and (Ptr^.Child^.next = nil) then     { update database if leaf node }
      UpdateArea(CharString, ReadingString);
  end; { ShowTranslation }


{ procedure to traverse a subtree of nodes in post order. If complete  }
{ translation, then all relevant nodes are displayed. Otherwise the top}
{ node translation is returned.                                        }
  procedure PostOrderTraversal;
    var
      LastChild: NodePtr;
      CPtr: ChildPtr;
  begin { PostOrderTraversal }
    if (Ptr <> nil) then
      begin
        PostOrderTraversal(Ptr^.Child^.Son);
        CPtr := Ptr^.Child^.next;
        if (CPtr <> nil) then
```

```
      repeat
        PostOrderTraversal(CPtr^.Son);
        CPtr := CPtr^.next;
      until (CPtr = nil);
     if not FullTranslate then
      begin                                { check if at sentence root node }
       if (Ptr^.Parent = Ptr) and (Ptr^.TEEnd = WordEnd) and (Ptr^.TEStart = WordStart) then
         ShowTranslation(Ptr)              { check if words are hilited      }
       else if (Ptr^.TEStart >= WordStart) and (Ptr^.TEEnd <= WordEnd) then
         if not ((Ptr^.Parent^.TEStart >= WordStart) and (Ptr^.Parent^.TEEnd <= WordEnd)) then
           ShowTranslation(Ptr)
      end
     else if (Ptr^.Child^.Son = nil) and (Ptr^.Child^.next = nil) then
      begin                                { child node                      }
       if (Ptr^.TEEnd >= WordStart) and (Ptr^.TEStart <= WordEnd) then
         ShowTranslation(Ptr)
      end
     else
      begin                                { complete translation }
       LastChild := EndNode(Ptr);
       if (Ptr^.Child^.Son^.TEStart >= WordStart) and (LastChild^.TEEnd <= WordEnd) then
         ShowTranslation(Ptr);
      end;
    end;
  end; { PostOrderTraversal }


{ procedure to find the pronounciation of the charstring pointed by the}
{ nodeptr specified by sentcount and nodecount.                        }

  procedure GetPronoun (SentCount, NodeCount: integer;
            CharString: str255;
            var ReadingString: str255);
   var
     index, i: integer;
     SentPtr: NodePtr;
  begin { GetPronoun }
   index := 0;
   ReadingString := Null;
   SentPtr := ArticlePtr;
   for i := 1 to (SentCount - 1) do
    if (SentPtr <> nil) then
     begin                             { advance to the correct sentence tree }
       index := index + SentPtr^.Child^.Son^.NodeId;
       SentPtr := SentPtr^.Child^.Next^.Son;
     end;
   index := index + NodeCount;         { advance to the child node in the tree  }
   if (index < CurrentNode) and (NewNodePtr[index]^.Pronoun.WordBegin > 0) and
       (NewNodePtr[index]^.Pronoun.WordLength > 0) then
     ReadingString := DisplayWords(NewNodePtr[index]^.Pronoun, SentPtr^.JapInfo);
  end; { GetPronoun }
```

```
{ procedure to select text characters when a double click occurs  }
{ It includes all the text contents pointed by the parent of the  }
{ node that specifies the current selection range. If no prehilited  }
{ text, it just hilites the child node content that encloses the  }
{ selection point.                                                }
 procedure SetSelection;
  var
   found: Boolean;
   Start: integer;
 begin { SetSelection }
  Start := CurSenPtr^.TEStart;          { if there is some text already hilited  }
  if (WordStart >= OldSelStart) and (WordStart < OldSelEnd) and (OldPtr <> nil) then
   begin
    if (OldPtr^.Parent <> OldPtr) then   { check if sentence root       }
     begin                               { hilite all nodes from parent }
      TESetSelect(OldPtr^.Parent^.TEStart + Start, OldPtr^.Parent^.TEEnd + Start + 1, TEText);
      TEText^^.SelStart := OldPtr^.Parent^.TEStart + Start;
      TEText^^.SelEnd := OldPtr^.Parent^.TEEnd + Start + 1;
      OldPtr := OldPtr^.Parent;
     end
   end
  else
   begin                                 { hilite the child node  }
    while (not found) and (Ptr^.Child^.Son <> nil) and (Ptr^.Child^.next <> nil) do
     found := SearchChild(2, Ptr);
    TESetSelect(Ptr^.TEStart + Start, Ptr^.TEEnd + Start + 1, TEText);
    TEText^^.SelStart := Ptr^.TEStart + Start;
    TEText^^.SelEnd := Ptr^.TEEnd + Start + 1;
    OldPtr := Ptr;
   end;
  OldSelStart := TEText^^.SelStart - Start;
  OldSelEnd := TEText^^.SelEnd - Start;
 end; {  SetSelection }


{ procedure to dispose sentence tree list as well as the nodes in newnodeptr }
 procedure DisposeLinkedList;
  var
   ParentPointer, NodePointer: NodePtr;
   i: integer;
 begin { DisposeLinkedList }
  if (ArticlePtr <> nil) then                { quit if no article tree }
   begin
    NodePointer := EndNode(ArticlePtr);      { find the end node       }
    while (NodePointer <> NodePointer^.Parent) do
     if (NodePointer <> nil) then
      begin
       ParentPointer := NodePointer^.Parent;  { save the parent node }
       DisposeInfoContent(NodePointer^.JapInfo);
       DisposeInfoContent(NodePointer^.EngInfo);
       dispose(NodePointer);
       NodePointer := ParentPointer;         { points to its parent  }
      end;
```

```
    if (ArticlePtr <> nil) then
      dispose(NodePointer);                    { dispose article pointer}
    for i := 1 to (CurrentNode - 1) do
     if (NewNodePtr[i] <> nil) then
      begin
        dispose(NewNodePtr[i]);
        NewNodePtr[i] := nil;
      end;
   end;
 end; {  DisposeLinkedList }


{ procedure to dispose the info content of a infoarray pointer  }
 procedure DisposeInfoContent;
  var
   NextPtr,  ContentPtr:  InfoArrayPtr;
 begin { DisposeInfoContent }
  ContentPtr := Root;
  while (ContentPtr <> nil) do                 { quit if no info content}
   begin
    NextPtr := ContentPtr^.Next;
    dispose(ContentPtr);                       { dispose the strings   }
    ContentPtr := NextPtr;
   end;
  Root := nil;
 end; {  DisposeInfoContent }


end.
```

```
unit CAIUtilities;

{*************************************************************}
{                                                             }
{ This unit provides useful utility routines for the Nihongo Tutor. }
{                                                             }
{*************************************************************}

{*************************************************************}
{                                                             }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or }
{ combinations thereof, now known or later developed.         }
{                                                             }
{*************************************************************}

interface

 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog, CAILinkedList;


 function DoubleClick: Boolean;

 procedure FixCursor;

 procedure CheckValidType (CheckType: PhoneticType;
           JapString: str255;
           var TypeValid: Boolean);

 procedure GetJapString (var JapString: str255;
           var Valid: Boolean);

 procedure GetFindReplaceString;

 procedure GetDictName (DictOption: integer;
           JapString: str255;
           var Extension: str255;
           var Valid: Boolean);

 procedure NewText (InHandle: Handle;
           var Text: TEHandle);

 procedure ShowSearchInfo (SearchString: str255;
           var SearchDictString: str255);

 procedure ShowCharInfo (JapString: str255;
           FromArticle: Boolean;
           var InfoFound, Exit: Boolean);
```

```
 procedure GetUserInfo (UserText: TEHandle;
           OffSet: Longint;
           UserName: str255);

 procedure GetUserField (var OldDBCount: integer;
           var UserArea: str255);

 procedure UpdateDatabase (ArticleNode: InfoArrayPtr);

 procedure Find;

 procedure Replace (var teh: TEHandle);

 procedure HandleContent (p: Point;
           var theWindow: WindowPtr;
           Extended: Boolean);

 procedure HandleGrow (var theWindow: WindowPtr;
           hSize, vSize: integer);

 procedure HandleCloseWindow (theWindow: WindowPtr);

 procedure HandleActivate (var teh: TEHandle;
           var vScrollBar, hScrollBar: ControlHandle;
           state: integer);

 procedure HandleZoom (var theWindow: WindowPtr;
           location: point;
           InOut: integer);

 procedure HandleTranslation;

 procedure HandleRadicalInfo;

 procedure HandleAbout;

 procedure HandleMenu;


implementation


{ function to check there occurs a double click.  Two mouse down events within a   }
{ short period of time is considered to a double-click event. One tick is sixtieth of }
{ a second.                                                                        }
 function DoubleClick;
  var
    t, time: longint;
    theEvent: EventRecord;
    DoubleClicked: Boolean;
 begin { DoubleClick }
  time := GetDblTime;          { ticks between a mouse-down and a mouse-up}
  DoubleClicked := false;
```

```
  t := TickCount;                { get current tick count   }
  begin
   while ((TickCount - t) < (time div 2)) and (not DoubleClicked) do
    if GetNextEvent(mDownMask, theEvent) then
     DoubleClicked := true;
  end;
  DoubleClick := DoubleClicked;
 end; { DoubleClick }


{ procedure to select an appropriate pattern for the cursor     }
{ It will be an ibeam if the mouse is within the front window   }
{ content and an arrow if not. It is fixed to be an arrow if    }
{ fixedcursor is true.                                          }
 procedure FixCursor;
  var
   p: Point;
   teh: TEHandle;
 begin { FixCursor }
  FixedCursor := (CurrentFunction <> EditCommand);
  if (FixedCursor) then
   SetCursor(Arrow)
  else if (FrontWindow = TextWindow) or (FrontWindow = TranslationWindow) or (FrontWindow =
      CharacterWindow) then
   begin
    if (FrontWindow = TextWindow) then    { check if in any window  }
     teh := TEText
    else if (FrontWindow = TranslationWindow) then
     teh := TranslationText
    else if (FrontWindow = CharacterWindow) then
     teh := CharacterText;
    GetMouse(p);
    if PtInRect(p, teh^^.ViewRect) then
     SetCursor(IBeam^^)    { check mouse position and set the cursor appropriately }
    else
     SetCursor(Arrow);
   end
  else
   ;                                    { let DA set the cursor }
 end; { FixCursor }


{ procedure to check whether a string of Japanese characters falls in a specific Japanese phonetic type}
{ It checks whether the ordinal value of the Japstring falls in the JIS code range for Kanji, Hiragana,  }
{ or Katagana.                                                                                            }
 procedure CheckValidType (CheckType: PhoneticType;
           JapString: str255;
           var TypeValid: Boolean);
  var
   i, limit: integer;
   OrdinalValue: Longint;
   valid: Boolean;
 begin { CheckValidType }
```

```
  Valid := true;
  if (CheckType <> English) then
   limit := (Length(Japstring) div 2)
  else
   limit := Length(Japstring);
  for i := 1 to limit do                          { check each char in the string   }
   If Valid then
    begin
     if (CheckType <> English) then
      OrdinalValue := JapOrdinal(JapString, (2 * i - 1))
     else
      OrdinalValue := ord(JapString[i]);
     if (CheckType = Kanji) then                  { check if in range  }
      Valid := (OrdinalValue >= KanjiLB) and (OrdinalValue <= KanjiUB)
     else if (CheckType = Katakana) then
      Valid := (OrdinalValue >= KataLB) and (OrdinalValue <= KataUB)
     else if (CheckType = Hiragana) then
      Valid := (OrdinalValue >= HiraLB) and (OrdinalValue <= HiraUB)
     else if (CheckType = English) then
      Valid := (OrdinalValue >= EnglishLB) and (OrdinalValue <= EnglishUB);
    end;
  TypeValid := Valid and (Length(Japstring) > 1);   { at least two bytes long  }
 end; { CheckValidType }


{ procedure to get of a hilited Kanji character from the text    }
{ The maximum size is ten characters long.                       }
 procedure GetHilitedString (var teh: TEHandle;
          var JapString: str255);
  var
   i: integer;
   StringSize: Longint;
   TEChar: CharsHandle;
   OutString: str255;
 begin { GetHilitedString }
  StringSize := 1;
  TEChar := TEGetText(teh);
  for i := 1 to (teh^^.SelEnd - teh^^.SelStart - 1) do
   StringSize := 10 * StringSize;                  { a no. of the same length  }
  NumToString(StringSize, OutString);
  for i := 1 to (teh^^.SelEnd - teh^^.SelStart) do
   OutString[i] := TEChar^^[i + teh^^.SelStart - 1];    { string has hilited chars  }
  JapString := OutString;
 end; { GetHilitedString }


{ procedure to a Kanji string from the front windows and test whether it has Kanji characters  }
 procedure GetJapString (var JapString: str255;
          var Valid: Boolean);
  var
   result: integer;
 begin { GetJapString }
  if (FrontWindow = TextWindow) then          { search for hilited string in front window }
```

```
    GetHilitedString(TEText, JapString)
   else if (FrontWindow = TranslationWindow) then
    GetHilitedString(TranslationText, JapString)
   else if (FrontWindow = CharacterWindow) then
    GetHilitedString(CharacterText, JapString);
   CheckValidType(Kanji, JapString, Valid);        { check if Kanji char   }
   if not Valid then
    DisplayDialog(OkD, 'Sorry character information is only available for Kanji characters !', result);
  end; { GetJapString }


{ procedure to a get the find and replace strings  }
  procedure GetFindReplaceString;
   var
    Findmsg, Replacemsg: str255;
    Cancelled: Boolean;
  begin { GetFindReplaceString }
   FindDialog(Findmsg, Replacemsg, Cancelled);  { get the strings    }
   if (not Cancelled) then
    begin
     FindString := Findmsg;
     ReplaceString := Replacemsg;
    end;
  end; { GetFindReplaceString }


{ procedure to get the extension of the dictionary file where the information          }
{ for the JapString would be obtained. This is provided for the character dictionary.  }
  procedure GetDictName (DictOption: integer;
            JapString: str255;
            var Extension: str255;
            var Valid: Boolean);
   var
    result: integer;
    OrdinalValue: Longint;
  begin { GetDictName }
   Valid := false;
   case DictOption of
    SBKanji:
     begin                                { get the ordinal value of the string }
      OrdinalValue := JapOrdinal(JapString, (Length(JapString) div 2));
      Extension := null;                  { check if fall in range of the dictionary file   }
      if (OrdinalValue >= CDKanji1LB) and (OrdinalValue <= CDKanji1UB) then
       Extension := '1'
      else if (OrdinalValue >= CDKanji2LB) and (OrdinalValue <= CDKanji2UB) then
       Extension := '2'
      else if (OrdinalValue >= CDKanji3LB) and (OrdinalValue <= CDKanji3UB) then
       Extension := '3'
      else if (OrdinalValue >= CDKanji4LB) and (OrdinalValue <= CDKanji4UB) then
       Extension := '4'
      else if (OrdinalValue >= CDKanji5LB) and (OrdinalValue <= CDKanji5UB) then
       Extension := '5'
      else if (OrdinalValue >= CDKanji6LB) and (OrdinalValue <= CDKanji6UB) then
```

```
        Extension := '6'
      else if (OrdinalValue >= CDKanji7LB) and (OrdinalValue <= CDKanji7UB) then
        Extension := '7'
      else if (OrdinalValue >= CDKanji8LB) and (OrdinalValue <= CDKanji8UB) then
        Extension := '8';
      Valid := (Extension <> Null);
      if (not Valid) then
        DisplayDialog(OkD, 'Sorry no character information for this Kanji character !!', result);
    end;
   otherwise
    DisplayDialog(OkD, 'Error -- dictionary option out of range !!', result);
  end;
 end; { GetDictName }


{ procedure to create new text handle with contents defined by          }
{ a handle. A copy of the handle content is stored in the text handle.  }
 procedure NewText (InHandle: Handle;
           var Text: TEHandle);
  var
   result: integer;
   TextHandle: Handle;
 begin { NewText }
  SetPort(CharacterWindow);
  TextHandle := InHandle;
  result := HandToHand(TextHandle);                { handle to handle copy }
  Text := TENew(DestRect, DestRect);
  DisposHandle(Text^^.hText);
  Text^^.hText := TextHandle;                      { read in  text chars.  }
  Text^^.teLength := GetHandleSize(TextHandle);
  TECalText(Text);                                 { update new text       }
 end; { NewText }


{ procedure to show the information of a dictionary search according to the }
{ dictionary file specified. This is provided for the search by Nelson's    }
{ dictionary, radical number, and stroke counts for the character dictionary.}
 procedure ShowGrammar (DictChar: CharsHandle;
           InfoStart, InfoEnd: Integer;
           InString: str255;
           var More, Exit: Boolean);
  var
   OldDBChar: CharsHandle;
   OldDBCount, result, item, Index: integer;
   JapString, EngString, DescString: str255;
   Example: InfoArrayPtr;
 begin { ShowGrammar }
  OldDBChar := DBChar;
  OldDBCount := DBCount;
  DBChar := DictChar;
  DBCount := InfoStart;
  Example := nil;
  EngString := Null;
```

```
  DescString := Null;
  JapString := NextLine(false, 1);
  result := 2;
  if ((GrammarOption = 2) and (Pos(InString, JapString) > 0)) or ((GrammarOption = 1) and (InString =
      JapString)) or (SearchOption = SBEnglish) then
   if (DBCount < InfoEnd) then
    begin
     EngString := NextLine(false, 1);
     Index := 1;
     while (Index <= (Length(EngString))) do
      if (EngString[index] = ';') then
       begin
        Insert(ReturnString, EngString, (Index + 1));
        Index := Index + 2;
       end
      else
       Index := Index + 1;
     if (DBCount < InfoEnd) then
      begin
       DescString := NextLine(false, 1);
       ReadExample(InfoEnd, Example);
      end;
     GrammarDialog(JapString, EngString, DescString, Example, result);
    end;
  Exit := (result = 1);
  More := (result = 2);
  DisposeInfoContent(Example);
  DBChar := OldDBChar;
  DBCount := OldDBCount;
 end; { ShowGrammar }


{ procedure to show the information of a dictionary search according to the  }
{ dictionary file specified. This is provided for the search by Nelson's     }
{ dictionary, radical number, and stroke counts for the character dictionary.}
 procedure ShowSearchInfo;
  var
   found: Boolean;
   result, i: integer;
   OffSet, SearchLength, SearchStart, InfoStart, InfoEnd: Longint;
   HeadingString: str255;
   DictText: TEHandle;
   DictChar: CharsHandle;
 begin { ShowSearchInfo }
  NewText(DictHandle, DictText);
  ShowWindow(CharacterWindow);
  SelectWindow(CharacterWindow);
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows      }
   HandleUpdate(UpdateEvent);
  SearchLength := GetHandleSize(DictHandle);
  case SearchOption of
   SBKanji, SBKatakana, SBHiragana:
    ;
```

```
    SBNelsonDictionary:
     HeadingString := Concat('Nelson"s Dictionary reference number = ', SearchString);
    SBRadicals:
     HeadingString := Concat('radical number = ', SearchString);
    SBStrokeCount:
     HeadingString := Concat('stroke counts = ', SearchString);
    otherwise
   end;                                              { search for the key in the dictionary   }
   SearchString := Concat(ReturnString, SearchString, ' ');
   SearchStart := 0;
   OffSet := Munger(DictHandle, SearchStart, POINTER(ord(@SearchString) + 1), Length(SearchString), nil,
       0);
   found := (OffSet >= 0) and (OffSet <= SearchLength);{ if found }
   if found then
    begin                                            { show heading            }
     HeadingString := Concat('The following character(s) conform with ', HeadingString, ' :');
     TESetSelect(MaxInt, MaxInt, CharacterText);
     TEInsert(Pointer(Ord(@HeadingString) + 1), Length(HeadingString), CharacterText);
     TEKey(chr(CR), CharacterText);                  { readjust char window    }
     ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
     InfoStart := OffSet + Length(SearchString);     { find the end of the information }
     OffSet := Munger(DictHandle, InfoStart, POINTER(ord(@ReturnString) + 1), Length(ReturnString), nil,
       0);
     found := (OffSet >= 0) and (OffSet <= SearchLength);
     InfoEnd := OffSet + 1;
     TESetSelect(InfoStart, InfoEnd, DictText);      { show search information}
     TECopy(DictText);
     TEPaste(CharacterText);
     SearchDictString := '  ';
     If (SearchOption = SBNelsonDictionary) then
      begin
       DictChar := TEGetText(CharacterText);
       SearchDictString[1] := DictChar^^[CharacterText^^.SelEnd - 3];
       SearchDictString[2] := DictChar^^[CharacterText^^.SelEnd - 2];
      end;
     TEKey(chr(CR), CharacterText);                  { readjust char window    }
     ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
    end
   else
    DisplayDialog(OkD, 'Sorry no information available from the dictionary !', result);
   TEDispose(DictText);
  end; { ShowSearchInfo }


{ procedure to show the numbers associated with a Kanji character.      }
{ They are information for the Nelson's dictionary reference number,    }
{ radical number and stroke counts.                                     }
 procedure ShowNumber (ShowString: str255;
           DictChar: CharsHandle;
           var DictText: TEHandle;
           var InfoStart: Longint;
           InfoEnd: Longint);
  var
```

```
    OffSet, InfoCount: Longint;
    i: integer;
    Valid: Boolean;
  begin { ShowCharInfo }
   OffSet := Munger(DictHandle, InfoStart, POINTER(ord(@SpaceString) + 1), Length(SpaceString), nil, 0);
   If (OffSet >= 0) and (OffSet <= InfoEnd) and (OffSet > InfoStart) then
    begin                                          { find the end of the number  }
     InfoCount := Offset + 1;
     TESetSelect(InfoStart, InfoCount, DictText);
     TECopy(DictText);
     Valid := true;
     for i := InfoStart to (OffSet - 1) do         { check if a number }
      If Valid then
       Valid := ((ord(DictChar^^[i]) >= $30) and (ord(DictChar^^[i]) <= $39));
     If Valid then                                 { show the number }
      begin
       TEInsert(POINTER(ord(@ShowString) + 1), Length(ShowString), CharacterText);
       TEPaste(CharacterText);
       InfoStart := InfoCount;
      end;
    end;
  end; { ShowNumber }


{ procedure to show the meaning associated with a Kanji character.      }
{ For the character dictionary, it would be a variable number of readings}
{ followed by the meaning associated with these readings. New readings  }
{ are displayed on separated lines. For the general character, there will }
{ be one single reading, followed by the meaning, for each entry.       }
{ The readings are returned to be stored in the user database if the key  }
{ string is from an article. The readings can only be expressed in      }
{ Hiragana or Katagana.                                                 }
  procedure ShowMeaning (FromArticle: Boolean;
            DictChar: CharsHandle;
            var DictText: TEHandle;
            var Reading: str255;
            var InfoStart: Longint;
            InfoEnd: Longint);
   const
    Delimiter = ';';
   var
    OffSet, InfoCount, index: Longint;
    Valid, found: Boolean;
    CheckString: str255;
  begin { ShowMeaning }
   InfoCount := InfoStart;
   Reading := Null;
   If (CurDictionary = CharacterDictionary) then      { if character dictionary  }
    begin
     If (infoEnd >= InfoStart) then
      begin
       TESetSelect(InfoStart, InfoEnd, DictText);
       TECopy(DictText);
```

```
      CheckString := '  ';
      repeat
       if (FromArticle) then
        begin
         index := InfoCount;
         CheckString[1] := DictChar^^[index];              { check the char after a space}
         CheckString[2] := DictChar^^[index + 1];
         CheckValidType(Katakana, CheckString, Valid); { check if a Katagana char }
         if (not Valid) then
          CheckValidType(Hiragana, CheckString, Valid);{ check if a Hiragana char }
         if Valid then
          repeat
           Reading := Concat(Reading, CheckString);    { store all readings    }
           index := index + 2;
           CheckString[1] := DictChar^^[index];
           CheckString[2] := DictChar^^[index + 1];    { until a space is hit    }
          until (CheckString[1] = SpaceString) or ((index + Length(CheckString)) > InfoEnd);
        end;
       OffSet := Munger(DictHandle, InfoCount, POINTER(ord(@SpaceString) + 1), Length(SpaceString),
      nil, 0);
       found := (OffSet >= 0) and ((OffSet + Length(CheckString)) <= InfoEnd);
       if found then
        begin
         CheckString[1] := DictChar^^[OffSet + 1];
         CheckString[2] := DictChar^^[OffSet + 2];
         CheckValidType(Katakana, CheckString, Valid); { check if a Katagana char }
         if (not Valid) then
          CheckValidType(Hiragana, CheckString, Valid);{ check if a Hiragana char }
         if Valid then
          begin
           if (FromArticle) then                         { if key string is from article }
            Reading := Concat(Reading, Delimiter);       { separate different readings }
           TESetSelect(InfoStart, Offset, DictText);
           TECopy(DictText);
           InfoStart := Offset + 1;
           TEPaste(CharacterText);
           TEKey(chr(CR), CharacterText);                { readjust char window    }
           ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
          end;
         InfoCount := OffSet + 1;
        end;
      until (not found) or (InfoCount > InfoEnd);
     end;
    end
   else if (CurDictionary = GeneralDictionary) then       { if general dictionary }
    begin
     OffSet := Munger(DictHandle, InfoCount, POINTER(ord(@SpaceString) + 1), Length(SpaceString), nil,
       0);
     found := (OffSet >= 0) and (OffSet <= InfoEnd);
     TEInsert(Pointer(Ord(@PronounString) + 1), Length(PronounString), CharacterText);
     TESetSelect(InfoStart, Offset, DictText);
     TECopy(DictText);
     TEPaste(CharacterText);                               { show the reading and meaning  }
```

```
      TEKey(')', CharacterText);
      DisplayArrow(CharacterText);
      InfoStart := Offset + 1;
     end;
   if (InfoEnd > InfoStart) then                    { print last information   }
    begin
     TESetSelect(InfoStart, InfoEnd, DictText);
     TECopy(DictText);
     TEPaste(CharacterText);
    end;
 end; { ShowMeaning }


{ procedure to show the information of a Kanji string from a chosen dictionary.    }
 procedure ShowCharInfo;
  const
   Delimiter = ';';
  var
   found: Boolean;
   result: integer;
   OffSet, SearchLength, InfoStart, InfoEnd, OldInfoStart, Start, index: Longint;
   DictText: TEHandle;
   DictChar: CharsHandle;
   HeadingString, Reading, DoubleReturn: str255;
 begin { ShowCharInfo }
  DoubleReturn := Concat(ReturnString, ReturnString);
  NewText(DictHandle, DictText);
  DictChar := TEGetText(DictText);
  SearchLength := GetHandleSize(DictHandle);
  Exit := false;
  Start := 0;
  repeat                        { check if key string is in the file starting from the location Start }
   OffSet := Munger(DictHandle, Start, POINTER(ord(@JapString) + 1), Length(JapString), nil, 0);
   found := (OffSet >= 0) and (OffSet <= SearchLength);
   if found then                                    { if key string is found }
    begin
     if (SearchOption = SBEnglish) or (SearchOption = SBHiragana) or (SearchOption = SBKatakana) then
      InfoStart := OffSet
     else
      InfoStart := OffSet + Length(JapString) + 1;
     if (InfoStart <= SearchLength) then            { find the end of the information }
      if (CurDictionary = GrammarDictionary) then
       OffSet := Munger(DictHandle, InfoStart, POINTER(ord(@DoubleReturn) + 1), Length(DoubleReturn),
      nil, 0)
      else
       OffSet := Munger(DictHandle, InfoStart, POINTER(ord(@ReturnString) + 1), Length(ReturnString),
      nil, 0);
     found := (OffSet >= 0) and (OffSet <= SearchLength);   { there is a match          }
     if found then
      begin
       InfoEnd := OffSet + 1;
       if (CurDictionary <> GrammarDictionary) then
        begin
```

```
      ShowWindow(CharacterWindow);
      SelectWindow(CharacterWindow);
      TESetSelect(MaxInt, MaxInt, CharacterText);
      while GetNextEvent(updateMask, UpdateEvent) do{ update all windows      }
       HandleUpdate(UpdateEvent);
     end;
    If (CurDictionary = CharacterDictionary) and (SearchOption <> SBEnglish) and (SearchOption <>
SBHiragana) and (SearchOption <> SBKatakana) then
     TEInsert(Pointer(Ord(@JapString) + 1), Length(JapString), CharacterText)
    else if (CurDictionary = GrammarDictionary) or (CurDictionary = GeneralDictionary) or
(CurDictionary = SpecialtyDictionary) or ((CurDictionary = CharacterDictionary) and
((SearchOption = SBEnglish) or (SearchOption = SBHiragana) or (SearchOption = SBKatakana))) then
     begin                                              { print the entire word    }
      If (not InfoFound) and (CurDictionary <> GrammarDictionary) then
       begin                                            { print the heading        }
        HeadingString := Concat('The following phrase(s) has meaning related to ', JapString, ' :',
ReturnString);
        TEInsert(Pointer(Ord(@HeadingString) + 1), Length(HeadingString), CharacterText);
        ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
       end;
      index := InfoStart;
      If ((CurDictionary = GrammarDictionary) and (SearchOption = SBEnglish)) then
       while ((DictChar^^[index] <> ReturnString) or (DictChar^^[index - 1] <> ReturnString)) and
(index > 0) do
        index := index - 1
      else
       while (DictChar^^[index] <> ReturnString) and (index > 0) do
        index := index - 1;
      If (CurDictionary = CharacterDictionary) and ((SearchOption = SBEnglish) or (SearchOption =
SBHiragana) or (SearchOption = SBKatakana)) then
       begin
        InfoStart := Index + 5;
        TESetSelect(Index + 1, Index + 4, DictText);
       end
      else if (CurDictionary = GeneralDictionary) then
       begin
        InfoStart := Index + 6;                         { get Japanese word    }
        TESetSelect(Index + 1, InfoStart, DictText);
       end
      else if (CurDictionary = SpecialtyDictionary) then
       begin
        If (SpecialtyDOption = Communications) then
         TESetSelect(Index + 1, InfoEnd, DictText);
       end
      else if (CurDictionary = GrammarDictionary) then
       If (DictChar^^[index - 1] = ReturnString) then
        ShowGrammar(DictChar, (Index + 1), InfoEnd, JapString, found, Exit);
      if (CurDictionary <> GrammarDictionary) then
       begin
        TECopy(DictText);
        TEPaste(CharacterText);
       end;
     end;
```

```
        OldInfoStart := InfoStart;
        If (CurDictionary = CharacterDictionary) then
         begin
          DisplayArrow(CharacterText);                    { print Japanese info     }
          InfoStart := InfoStart - 1;
          ShowNumber('Number = ', DictChar, DictText, InfoStart, InfoEnd);
          ShowNumber('Radical = ', DictChar, DictText, InfoStart, InfoEnd);
          ShowNumber('Strokes = ', DictChar, DictText, InfoStart, InfoEnd);
         end;
        If (InfoStart > OldInfoStart) then                { if there are numbers    }
         begin
          TEKey(chr(CR), CharacterText);                  { readjust char window    }
          ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
         end;
        If (CurDictionary <> SpecialtyDictionary) and (CurDictionary <> GrammarDictionary) then
         ShowMeaning(FromArticle, DictChar, DictText, Reading, InfoStart, InfoEnd);
        If (CurDictionary = CharacterDictionary) then
         TEKey(chr(CR), CharacterText);                   { readjust char window    }
        ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
        If (CurDictionary = GrammarDictionary) or (CurDictionary = GeneralDictionary) or
       (CurDictionary = SpecialtyDictionary) or ((CurDictionary = CharacterDictionary) and
       ((SearchOption = SBEnglish) or (SearchOption = SBHiragana) or (SearchOption = SBKatakana))) then
          Start := InfoEnd;
       end;
     end;
    InfoFound := InfoFound or found;
   until ((CurDictionary = CharacterDictionary) and (SearchOption <> SBEnglish) and (SearchOption <>
       SBHiragana) and (SearchOption <> SBKatakana)) or (not found) or Exit or (Start > SearchLength);
   If (FromArticle) then
    begin
     RemoveLastChar(Reading, Delimiter);                  { remove last delimiter of reading  }
     RemoveLastChar(JapString, SpaceString);              { remove last space of key string   }
    end;
   If (not found) and (CurDictionary = CharacterDictionary) and (SearchOption <> SBEnglish) and
      (SearchOption <> SBHiragana) and (SearchOption <> SBKatakana) then
    DisplayDialog(OkD, 'Sorry no information for this Kanji character(s) !', result)
   else If (CurDictionary = CharacterDictionary) and (FromArticle) then
    UpdateArea(JapString, Reading);                       { update user database    }
   TEDispose(DictText);
   DisposHandle(DictHandle);
  end; { ShowCharInfo }


{ procedure to get a char handle which contains all content of the user file.  }
{ DBCount is set to locate from Offset and beyond the user name.               }
  procedure GetUserInfo (UserText: TEHandle;
            OffSet: Longint;
            UserName: str255);
  begin { GetUserInfo }
   DBChar := TEGetText(UserText);
   DBCount := LoWord(Offset) + Length(UserName) + 2; { skip delimiter }
  end; { GetUserInfo }
```

```
{ procedure to get a field in the user field. }
 procedure GetUserField (var OldDBCount: integer;
           var UserArea: str255);
 begin { GetUserField }
  DBCount := DBCount + 1;                    { skip leading space    }
  OldDBCount := DBCount;
  UserArea := NextLine(false, 1);            { get next user field   }
 end; { GetUserField }


{ procedure to update the unfamiliar area by decrementing the counter of    }
{ word/char and remove the entry from it and add to the well-known area     }
{ if the counter becomes zero or negative after the subtraction.            }
 procedure MoveToArea (var Bucket, CurrentPtr, OldPtr: InfoArrayPtr;
           Delimiter: str255;
           Counter: integer);
  var
   TempPtr, OldTempPtr: InfoArrayPtr;
   CharString, ReadingString: str255;
   NumCount, BucketNumber: integer;
   Found: Boolean;
 begin { MoveToArea }
  UpdateReading(CurrentPtr, Null, Delimiter, true, Counter, NumCount, CharString, ReadingString);
  if (ReadingString <> Null) then            { update the entries    }
   if (ReadingString[1] = Delimiter) then
    Delete(ReadingString, 1, 1);             { remove leading space}
  RemoveLastChar(ReadingString, Delimiter);
  if ((NumCount - Counter) <= 0) then        { move it to well-known area }
   begin
    RemoveContent(Bucket, CurrentPtr, OldPtr);{ remove old entry}
    GetBucket(WKHashTable, CharString, BucketNumber);
    CheckArea(WKHashTable[BucketNumber], Delimiter, CharString, Found, TempPtr, OldTempPtr);
    if (not Found) then                      { add to well-known area  }
     AddContent(WKHashTable[BucketNumber], CharString, ReadingString, Delimiter, 0, 1);
   end;
 end; { MoveToArea }


{ procedure to update the user information with the article database file }
{ All leaf nodes from the article will be added to well-known area if they }
{ are not found in either area. If it is found in the unfamiliar area, its  }
{ counter will be decremented and if the result is less than or equal to    }
{ zero, it is moved to the well-known area.                                 }
 procedure UpdateDatabase (ArticleNode: InfoArrayPtr);
  const
   Delimiter = ';';
  var
   CurrentPtr, OldPtr: InfoArrayPtr;
   LeafNode, NodeList, LeafPronoun: str255;
   index, NodeCount, NumCount, SentCount, BucketNumber: integer;
   Found: Boolean;
 begin { UpdateDatabase }
```

```
  if (ArticleNode <> nil) then
   begin
    SentCount := 1;
    repeat
     GetNextString(ArticleNode, DelimiterString, 1, NodeList);
     if (NodeList <> Null) then                    { get no of nodes in a sentence    }
      begin
       ReadString(GetSingleArea(NodeList, Delimiter, 1), NodeCount);
       for index := 2 to (NodeCount + 1) do
        begin                                      { get next basic node   }
         LeafNode := GetSingleArea(NodeList, Delimiter, index);
         GetBucket(UnFHashTable, LeafNode, BucketNumber);
         CheckArea(UnFHashTable[BucketNumber], Delimiter, LeafNode, Found, CurrentPtr, OldPtr);
         if (Found) then                           { update all well-known and unfamiliar chars  }
          MoveToArea(UnFHashTable[BucketNumber], CurrentPtr, OldPtr, Delimiter, DecrementCount)
         else
          begin
           GetBucket(WKHashTable, LeafNode, BucketNumber);
           CheckArea(WKHashTable[BucketNumber], Delimiter, LeafNode, Found, CurrentPtr, OldPtr);
           if (not Found) then                     { add to well-known area  }
            begin
             GetPronoun(SentCount, (index - 1), LeafNode, LeafPronoun);
             AddContent(WKHashTable[BucketNumber], LeafNode, LeafPronoun, Delimiter, 0, 1);
            end;
          end;
        end;
      end;
     SentCount := SentCount + 1;
    until (NodeList = Null);                       { repeat until all sentences are processed  }
   end;                                            { now unfamiliar area is updated }
 end; { UpdateDatabase }


{ procedure to a find a string in a text window. It is hilited if found.  }
 procedure Find;
  var
   SearchText: TEHandle;
   SearchString: str255;
   Location, Offset: Longint;
   result: integer;
   found: Boolean;
 begin { GetFindReplaceString }
  SearchString := FindString;
  if (SearchString <> Null) then
   begin
    if (FrontWindow = TextWindow) then             { search through the front window   }
     SearchText := TEText
    else if (FrontWindow = TranslationWindow) then
     SearchText := TranslationText
    else if (FrontWindow = CharacterWindow) then
     SearchText := CharacterText;                  { start with current selection point  }
    if (SearchText^^.SelStart >= 0) and ((SearchText^^.SelEnd - SearchText^^.SelStart) =
      Length(SearchString)) then
```

```
      Offset := SearchText^^.SelStart + Length(SearchString)
     else
      Offset := SearchText^^.SelStart;
     Location := Munger(SearchText^^.hText, Offset, POINTER(ord(@SearchString) + 1),
       Length(SearchString), nil, 0);
     found := (Location >= 0) and (Location <= SearchText^^.teLength);      { if there is a match      }
     if (found) then
      begin
       TESetSelect(Location, Location + Length(SearchString), SearchText);
       if (FrontWindow = TextWindow) then              { hilite the string   }
        ShowInsertion(TextWindow, TEText, TELines, vTextSB, hTextSB)
       else if (FrontWindow = TranslationWindow) then
        ShowInsertion(TranslationWindow, TranslationText, TransLines, vTransSB, hTransSB)
       else if (FrontWindow = CharacterWindow) then
        ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
      end
     else
      DisplayDialog(OkD, Concat('Sorry cannot find ', SearchString, ' from the current selection !'), result)
    end
   else
    DisplayDialog(OkD, 'Error -- invalid search string !!  Cannot proceed.', result)
  end; { Find }


{ procedure to a replace a hilited string in a text window with the replacestring}
  procedure Replace;
   var
    ChangeString: str255;
    Offset: Longint;
    result: integer;
  begin { Replace }
   ChangeString := ReplaceString;
   if (ChangeString <> Null) then
    begin
     Offset := teh^^.SelStart;                    { delete old hilited string      }
     TEDelete(teh);                               { replace with change string   }
     TEInsert(POINTER(ord(@ChangeString) + 1), Length(ChangeString), teh);
     TESetSelect(Offset, Offset + Length(ChangeString), teh);
    end
   else
    DisplayDialog(OkD, 'Error -- invalid replace string !!  Cannot proceed.', result);
  end; { Replace }


{ procedure to handle the content of all hilited characters when a selection is made.}
{ it first checks whether mouse is down in the scroll bar areas, then it checks if    }
{ there occurs a double click in the content of the text window. It goes up the parse}
{ tree of the sentence to hilite all of its children if a double click happens.       }
  procedure HandleContent;
   var
    pt: Point;
    NodePointer: NodePtr;
    control: ControlHandle;
```

```
    error, location: integer;
  begin { HandleContent }
   pt := p;
   GlobalToLocal(pt);
   location := FindControl(pt, theWindow, control);  { check if in scroll bar        }
   if (location <> 0) then
    ScrollContent(control, theWindow, location, pt)
   else if (theWindow = TextWindow) then       { work for the text window       }
    begin
     TEClick(pt, Extended, TEText);
     if TextOpened then                        { check if text database opened  }
      if DoubleClick then                      { check if double click occurs   }
       begin
        LocateWordinSentence(NodePointer, error);
        if (NodePointer <> nil) then           { hilite words of its parent     }
         SetSelection(NodePointer);
       end
      else
       begin
        OldSelStart := 0;                      { reset old settings if not double-click  }
        OldSelEnd := 0;
       end;
    end
   else if (theWindow = TranslationWindow) then
    TEClick(pt, Extended, TranslationText)     { other windows have normal double-clicking  }
   else if (theWindow = CharacterWindow) then
    TEClick(pt, Extended, CharacterText);
  end; { HandleContent }


{ procedure to change the size of a window when its size box is changed  }
  procedure HandleGrow;
   var
    oldPort: GrafPtr;
  begin { HandleGrow }
   GetPort(oldPort);
   SetPort(theWindow);
   SizeWindow(theWindow, hSize, vSize, true);          { resize the window}
   InvalRect(theWindow^.portRect);                     { force update     }
   if (theWindow = TextWindow) then                    { update the window}
    UpdateWindow(theWindow, TEText, TELines, TEWidth, vTextSB, hTextSB)
   else if (theWindow = TranslationWindow) then
    UpdateWindow(theWindow, TranslationText, TransLines, TransWidth, vTransSB, hTransSB)
   else if (theWindow = CharacterWindow) then
    UpdateWindow(theWindow, CharacterText, CharLines, CharWidth, vCharSB, hCharSB);
   SetPort(oldPort);
  end;{ HandleGrow }


{ procedure to close a window.  If the window is a DA window, it is closed.  }
{ If it is a window created by the application, it is just hidden.           }
  procedure HandleCloseWindow;
   var
```

```
    window: WindowPeek;
  begin { HandleCloseWindow }
   if not ((theWindow = TextWindow) or (theWindow = TranslationWindow) or (theWindow =
       CharacterWindow)) then
    begin
     window := WindowPeek(theWindow);
     with window^ do
      if (windowKind < 0) then              { a DA window          }
       CloseDeskAcc(windowKind)             { close the DA         }
      else
       HideWindow(WindowPtr(window))        { an application window}
    end
   else
    HideWindow(thewindow);                  { hide other window    }
  end; { HandleCloseWindow }


{ procedure to activative and deactivative a text and its control handles.}
  procedure HandleActivate;
  begin { HandleActivate }
   case state of
    Active:                                 { activate text and controls  }
     begin
      TEActivate(Teh);
      HiLiteControl(vScrollBar, Active);
      HiLiteControl(hScrollBar, Active);
     end;
    InActive:                               { deactivate text and controls}
     begin
      TEDeactivate(Teh);
      HiLiteControl(vScrollBar, InActive);
      HiLiteControl(hScrollBar, InActive);
     end;
    otherwise
   end;
  end; { HandleActivate }


{ procedure to handle zooming a window when the zoom box is selected   }
  procedure HandleZoom;
  begin { HandleZoom }
   if TrackBox(theWindow, location, InOut) then          { if mouse down in zoom box  }
    begin
     EraseRect(theWindow^.portRect);
     ZoomWindow(theWindow, InOut, false);                { zoom the window in or out  }
     InvalRect(theWindow^.portRect);
     if (theWindow = TextWindow) then                    { update the window text  }
      FixText(theWindow, TEText, vTextSB, hTextSB)
     else if (theWindow = TranslationWindow) then
      FixText(theWindow, TranslationText, vTransSB, hTransSB)
     else if (theWindow = CharacterWindow) then
      FixText(theWindow, CharacterText, vCharSB, hCharSB);
     if (theWindow = TextWindow) then                    { make the window active }
```

```
      HandleActivate(TEText, vTextSB, hTextSB, Active)
     else if (theWindow = TranslationWindow) then
      HandleActivate(TranslationText, vTransSB, hTransSB, Active)
     else if (theWindow = CharacterWindow) then
      HandleActivate(CharacterText, vCharSB, hCharSB, Active);
    end;
 end;{ HandleZoom }


{ procedure to provide translation for some Japanese words. It checks whether }
{ the hilited words fall in a single sentence range and then locates the lowest }
{ level node that includes all the words. If found, translation information is then}
{ given.                                                                       }
 procedure HandleTranslation;
  var
   NodePointer, TranslateRootNode: NodePtr;
   error, result: integer;
 begin { HandleTranslation }
  SetCursor(Watch^^);
  LocateWordinSentence(NodePointer, error);          { locate the sentence     }
  case error of                                      { check for errors        }
   1:
    DisplayDialog(OkD, 'Sorry translation is not available for these characters !', result);
   2:
    DisplayDialog(OkD, 'Error -- translation is provided for each single sentence only!!', result);
   otherwise
    if (NodePointer <> nil) then
     begin
      LocateWordinNode(NodePointer, TranslateRootNode);{ find the parent node }
      ShowWindow(TranslationWindow);
      SelectWindow(TranslationWindow);
      TESetSelect(MaxInt, MaxInt, TranslationText);
      while GetNextEvent(updateMask, UpdateEvent) do { update all windows   }
       HandleUpdate(UpdateEvent);
      PostOrderTraversal(TranslateRootNode);         { print translation       }
      TEKey(chr(CR), TranslationText);               { readjust trans window   }
      ShowInsertion(TranslationWindow, TranslationText, TransLines, vTransSB, hTransSB);
     end;
  end;
 end; { HandleTranslation }


{ procedure to handle showing the radical information dialog    }
 procedure HandleRadicalInfo;
  var
   result: integer;
 begin { HandleRadicalInfo }
  DisplayDialog(RadicalD, Null, result);
 end; { HandleRadicalInfo }


{ procedure to handle showing the about dialog    }
 procedure HandleAbout;
```

```
  var
    result: integer;
  begin { HandleAbout }
   DisplayDialog(AboutD, Null, result);
  end; { HandleAbout }


{ procedure to disable and enable items in each menu under appropriate conditions }
  procedure HandleMenu;
   var
    i: integer;
  begin { HandleMenu }
  {   if EditFunction then}
   if (EditFunction or (CurrentFunction = EditCommand)) then
    begin                                                { function is edit    }
     DisableItem(Menus[FunctionM], TranslateCommand);
     DisableItem(Menus[FunctionM], FullTranslationCommand);
     DisableItem(Menus[FunctionM], CharacterInfoCommand);
     if (CurrentFunction = EditCommand) then
      if (FrontWindow <> TextWindow) then     { only edit with text window }
       begin
        DisableItem(Menus[FileM], RevertCommand);
        DisableItem(Menus[FunctionM], EditCommand);
        DisableItem(Menus[EditM], CutCommand);
        EnableItem(Menus[EditM], CopyCommand);
        DisableItem(Menus[EditM], ClearCommand);
        DisableItem(Menus[EditM], PasteCommand);
        DisableItem(Menus[EditM], SelectAllCommand);
        DisableItem(Menus[EditM], ReplaceCommand);
       end
      else                                        { text window is in front  }
       begin
        EnableItem(Menus[FileM], RevertCommand);
        EnableItem(Menus[FunctionM], EditCommand);
        EnableItem(Menus[EditM], PasteCommand);
        EnableItem(Menus[EditM], SelectAllCommand);
        EnableItem(Menus[EditM], FindWhatCommand);
        EnableItem(Menus[EditM], FindCommand);
        if (TEText^^.SelEnd > TEText^^.SelStart) then     { check if hilited}
         begin
          EnableItem(Menus[EditM], CutCommand);
          EnableItem(Menus[EditM], CopyCommand);
          EnableItem(Menus[EditM], ClearCommand);
          EnableItem(Menus[EditM], ReplaceCommand);
         end
        else
         begin
          DisableItem(Menus[EditM], CutCommand);
          DisableItem(Menus[EditM], CopyCommand);
          DisableItem(Menus[EditM], ClearCommand);
          DisableItem(Menus[EditM], ReplaceCommand);
         end;
       end
```

```
    end
   else
    begin                                    { function is learn text }
     if (FrontWindow <> TextWindow) then
      DisableItem(Menus[FunctionM], EditCommand)
     else
      EnableItem(Menus[FunctionM], EditCommand);
     DisableItem(Menus[FileM], RevertCommand);  { editing functions disabled  }
     DisableItem(Menus[EditM], CutCommand);
     DisableItem(Menus[EditM], CopyCommand);
     DisableItem(Menus[EditM], PasteCommand);
     DisableItem(Menus[EditM], ClearCommand);
     DisableItem(Menus[EditM], SelectAllCommand);
     EnableItem(Menus[EditM], FindWhatCommand);
     EnableItem(Menus[EditM], FindCommand);
     DisableItem(Menus[EditM], ReplaceCommand);
     if (TEText^^.SelEnd > TEText^^.SelStart) and TextOpened then
      begin
       EnableItem(Menus[FunctionM], TranslateCommand);
       EnableItem(Menus[FunctionM], FullTranslationCommand);
      end
     else
      begin
       DisableItem(Menus[FunctionM], TranslateCommand);
       DisableItem(Menus[FunctionM], FullTranslationCommand);
      end;
     DisableItem(Menus[FunctionM], CharacterInfoCommand);
     if ((FrontWindow = TextWindow) and (TEText^^.SelEnd > TEText^^.SelStart)) and ((TEText^^.SelEnd -
       TEText^^.SelStart) <= CharLengthMax) and (CurDictionary = CharacterDictionary) then
      EnableItem(Menus[FunctionM], CharacterInfoCommand)
     else if ((FrontWindow = TranslationWindow) and (TranslationText^^.SelEnd >
       TranslationText^^.SelStart)) and ((TranslationText^^.SelEnd - TranslationText^^.SelStart) <=
       CharLengthMax) and (CurDictionary = CharacterDictionary) then
      EnableItem(Menus[FunctionM], CharacterInfoCommand)
     else if ((FrontWindow = CharacterWindow) and (CharacterText^^.SelEnd > CharacterText^^.SelStart))
       and ((CharacterText^^.SelEnd - CharacterText^^.SelStart) <= CharLengthMax) and (CurDictionary =
       CharacterDictionary) then
      EnableItem(Menus[FunctionM], CharacterInfoCommand);
    end;
   if (FrontWindow <> nil) then
    EnableItem(Menus[FileM], CloseCommand)
   else
    DisableItem(Menus[FileM], CloseCommand);
   if (FrontWindow = TextWindow) or (FrontWindow = TranslationWindow) or (FrontWindow =
      CharacterWindow) then
    EnableItem(Menus[FileM], PrintCommand)
   else
    DisableItem(Menus[FileM], PrintCommand);
   if (CurDictionary <> CharacterDictionary) then      { update pop-up menu items  }
    begin
     for i := SBKanji to SBHiragana do
      EnableItem(OptionsMenu, i);
     for i := SBNelsonDictionary to SBStrokeCount do
```

```
      DisableItem(OptionsMenu, i);
    end
   else
    for i := SBKanji to SBStrokeCount do
     EnableItem(OptionsMenu, i);

{   prevent direct access of files    }
{    DisableItem(Menus[FileM], OpenCommand);}
 end; { HandleMenu }


end.
```

```
unit CAIPrint;

{**************************************************************}
{                                                              }
{ This unit handles the Page Setup routine as well as the Print Routine for the main program. }
{                                                              }
{**************************************************************}

{**************************************************************}
{                                                              }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or }
{ combinations thereof, now known or later developed.          }
{                                                              }
{**************************************************************}

interface

 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog;


 procedure HandlePageSetup;

 procedure HandlePrint;


implementation


{ procedure to handle page setup          }
 procedure HandlePageSetup;
  var
    PrintRecOk, NewStyle: Boolean;
 begin { HandlePageSetup }
  PrOpen;
  if (not PrintStyle) then
    PrintDefault(prRecHandle);                  { default record   }
  PrintRecOk := PrValidate(prRecHandle);        { validate record  }
  NewStyle := PrStlDialog(prRecHandle);         { get new style    }
  PrintStyle := PrintStyle or NewStyle;
  PrClose;
 end; { HandlePageSetup }


{ function to get a copy of the text to be printed }
 function GetText (var teh: TEHandle): Boolean;
  var
    SourceText: TEHandle;
    HandleCopy: Handle;
```

```
    error, result: integer;
  begin { GetText }
   SetCursor(Watch^^);
   if (FrontWindow = TextWindow) then              { find the correct text    }
    SourceText := TEText
   else if (FrontWindow = TranslationWindow) then
    SourceText := TranslationText
   else if (FrontWindow = CharacterWindow) then
    SourceText := CharacterText;
   HandleCopy := SourceText^^.hText;
   error := HandToHand(HandleCopy);                { duplicate the text handle }
   if (error <> noErr) then
    DisplayDialog(OkD, 'Error occurs while printing!! The printing process is cancelled.', result)
   else
    begin                                          { create a new text with the same content }
     teh := TENew(DestRect, DestRect);
     DisposHandle(teh^^.hText);
     teh^^.hText := HandleCopy;
     teh^^.teLength := GetHandleSize(HandleCopy);
     TECalText(teh);
    end;
   GetText := (error = noErr);
  end; { GetText }


{ procedure to get the pages of a document to be printed }
  procedure GetPages (MaxPages: integer;
            var FirstPageNo, LastPageNo: integer);
  begin { GetPages }
   FirstPageNo := prRecHandle^^.prJob.iFstPage;    { save requested page nos.}
   if (prRecHandle^^.prJob.iLstPage <= MaxPages) then
    LastPageNo := prRecHandle^^.prJob.iLstPage
   else
    LastPageNo := MaxPages;                        { set max. print pages     }
   prRecHandle^^.prJob.iFstPage := 1;
   prRecHandle^^.prJob.iLstPage := MaxPages;
  end; { GetPages }


{ procedure to print a document }
  procedure PrDoc (teh: TEHandle;
            PrPort: TPPrPort;
            MaxPages, Margin, FirstPageNo, LastPageNo, linesPerPage, totalLines: integer);
   var
    PageNo, i: Integer;
    Start, SelEnd: Integer;
    PrintText: TEHandle;
  begin { PrDoc }
   Start := 0;
   for PageNo := 1 to MaxPages do                  { loop through all pages   }
    if (Start <= teh^^.teLength) then              { check if out of range    }
     If (PrError <> NoErr) then
      PrSetError(iPrAbort)                         { abort if print errors    }
```

```
    else
     begin
      PrOpenPage(PrPort, nil);                    { start new page          }
      if (PrError <> NoErr) then
       PrSetError(iPrAbort)                       { abort if print errors   }
      else
       begin
        SelEnd := MaxInt;
        if ((PageNo * linesPerPage) <= teh^^.nLines) then
         SelEnd := teh^^.lineStarts[PageNo * linesPerPage];
        if (PageNo >= FirstPageNo) and (PageNo <= LastPageNo) then
         begin                                    { print selected pages    }
          TESetSelect(Start, SelEnd, teh);
          TECopy(teh);
         end;
        Start := SelEnd;
        if (PageNo >= FirstPageNo) and (PageNo <= LastPageNo) then
         begin
          PrintText := TENew(thePort^.portRect, thePort^.portRect);
          TEPaste(PrintText);
          for i := 1 to Margin do                 { adjust the boundary     }
           TEKey(chr(CR), PrintText);
          TEDispose(PrintText);
         end;
       end;
      PrClosePage(PrPort);
     end;
 end; { PrDoc }


{ procedure to handle print routine          }
 procedure HandlePrint;
  const
   MaxPages = 128;
   Margin = 2;
  var
   PrPort: TPPrPort;
   StatusRec: TPrStatus;
   OldPort, PrintPort: GrafPtr;
   PrintRecOk, NewPrintJob: Boolean;
   FirstPageNo, LastPageNo, result: Integer;
   linesPerPage, totalLines: Integer;
   teh: TEHandle;
 begin { HandlePrint }
  PrOpen;
  if (not PrintStyle) then
   PrintDefault(prRecHandle);
  PrintRecOk := PrValidate(prRecHandle);
  NewPrintJob := PrJobDialog(prRecHandle);
  if NewPrintJob then
   if GetText(teh) then
    begin
     GetPort(OldPort);
```

```
      while GetNextEvent(updateMask, UpdateEvent) do  { update all windows      }
       HandleUpdate(UpdateEvent);
      PrPort := PrOpenDoc(prRecHandle, nil, nil);        { open printing grafPort  }
      GetPages(MaxPages, FirstPageNo, LastPageNo);
      PrintPort := @PrPort^.gPort;
      SetPort(PrintPort);
      TextFont(FontNum);
      TextSize(FontSize);
      totalLines := (prRecHandle^^.prInfo.rPage.bottom - prRecHandle^^.prInfo.rPage.top) div lineHt;
      linesPerPage := totalLines - Margin;                { compute printing parameters}
      HLock(teh^^.hText);                                 { print the document      }
      PrDoc(teh, PrPort, MaxPages, Margin, FirstPageNo, LastPageNo, linesPerPage, totalLines);
      HUnLock(teh^^.hText);
      PrCloseDoc(PrPort);
      if (prRecHandle^^.prJob.bjDocLoop = bSpoolLoop) and (PrError = NoErr) then
       PrPicFile(prRecHandle, nil, nil, nil, StatusRec);  { spooled printing        }
      SetPort(OldPort);
      if (PrError <> NoErr) and (PrError <> iPrAbort) then
       DisplayDialog(OkD, Concat('Error ', NumberToString(PrError), ' occurs while printing !!  The
       printing process is cancelled.'), result);
      TEDispose(teh);
    end;
  PrClose;
 end; { HandlePrint }


end.
```

```
unit CAIInit;


{*****************************************************************}
{                                                                 }
{ This unit handles the most of the initialization processes for the main program.  It creates  }
{ the menubar, and sets it up.  It uses standard routines provided by QuickDraw.               }
{                                                                 }
{*****************************************************************}

{*****************************************************************}
{                                                                 }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All     }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or          }
{ combinations thereof, now known or later developed.                                     }
{                                                                 }
{*****************************************************************}


interface

 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog, CAILinkedList, CAIUtilities;

 procedure Initialize;


implementation


{ procedure to create a single menu and append to the menu list }
 procedure CreateMenu (Id: integer;
           var Mhandle: MenuHandle;
           title, items: string);
 begin { CreateMenu }
  Mhandle := NewMenu(Id, title);
  AppendMenu(MHandle, items);
  InsertMenu(MHandle, 0);       { insert to the end  }
 end; { CreateMenu }


{ procedure to set up each menu in the menu bar                   }
 procedure SetUpMenus;
  var
   items, appleTitle: string;
 begin { SetUpMenus }
  ClearMenuBar;

  appleTitle := 'a';
  appleTitle[1] := Chr(AppleMark);
  CreateMenu(AppleM, Menus[AppleM], appleTitle, 'About Nihongo Tutor...;(-');
```

```
  AddResMenu(Menus[AppleM], 'DRVR');

  items := 'New       /N;Open...   /O;Close;(-;Save        /S;Save As...;Revert;(-;Page
      Setup...;Print... /P;(-;Quit      /Q';
  CreateMenu(FileM, Menus[FileM], 'File', items);

  items := 'Undo;(-;Cut      /X;Copy     /C;Paste    /V;Clear;(-;Select All;(-;Find What...  /W;Find
      /F;Replace  /R';
  CreateMenu(EditM, Menus[EditM], 'Edit', items);

  Menus[FunctionM] := GetMenu(FunctionMId);       { get function menu from the resource file    }
  InsertMenu(Menus[FunctionM], 0);
  OptionsMenu := GetMenu(SearchOptionMId);        { get the search pop-up submenu }
  InsertMenu(OptionsMenu, -1);
  FontMenu := GetMenu(FontMId);                   { get the font pop-up submenu    }
  AddResMenu(FontMenu, 'FONT');
  InsertMenu(FontMenu, -1);
  SizeMenu := GetMenu(SizeMId);                   { get the font size pop-up submenu}
  InsertMenu(SizeMenu, -1);

  Menus[DictionaryM] := GetMenu(DictionaryMId); { get dictionary menu from the resource file  }
  InsertMenu(Menus[DictionaryM], 0);
  SpecialtyDMenu := GetMenu(SpecialtyDMId);       { get the specialty dictionary pop-up submenu }
  InsertMenu(SpecialtyDMenu, -1);

  items := 'Text      /1;Translation /2;Dictionary  /3';
  CreateMenu(WindowM, Menus[WindowM], 'Windows', items);

  DrawMenuBar;
 end; { SetUpMenus }


{ procedure to initialize all routines, create the menubar and the windows.                }
{ textdatabase is the file contains all information about the translation nodes of an article.}
 procedure Initialize;
  var
{fInfo: FMetricRec;}
    ErrCode, result: integer;
    String50: str255;
 begin { Initialize }
  FlushEvents(everyEvent, 0);
  InitCursor;

  SetUpMenus;
  IBeam := GetCursor(IBeamCursor);
  Watch := GetCursor(watchCursor);
  SetCursor(Watch^^);
  SetRect(TextRect, 4, 50, 500, 280);
  SetRect(TranslationRect, 4, 70, 500, 300);
  SetRect(CharacterRect, 4, 90, 500, 320);
  with ScreenBits.bounds do
   SetRect(GrowLimitRect, 80, 80, (right - left), (bottom - top - 20));
  FontNum := Geneva;
```

```
  FontSize := 12;
  TextFont(FontNum);
  TextSize(FontSize);
  GetFontInfo(fInfo);
{   FontMetrics(fInfo);}
  with fInfo do
   lineHt := ascent + descent + leading;
  with ScreenBits.bounds do                          { open the windows }
   SetRect(DragRect, 4, 24, right - 4, bottom - 4);
  OpenWindow(TextWindow, vTextSB, hTextSB, TEText, Null, TextRect, TELines, TEWidth, false);
  OpenWindow(TranslationWindow, vTransSB, hTransSB, TranslationText, TransWTitle, TranslationRect,
      TransLines, TransWidth, false);
  OpenWindow(CharacterWindow, vCharSB, hCharSB, CharacterText, CharWTitle, CharacterRect,
      CharLines, CharWidth, false);
  SetPort(TextWindow);

  FullTranslate := false;                  { initialize the gloval variables  }
  TextOpened := false;
  DictOpened := false;
  PrintStyle := false;
  FurRef := '< further reference >';
  PronounString := ' (pronounced as ';
  ArrowString := '  ==>  ';
  CurDictionary := CharacterDictionary;
  CurDictName := CharDictionary;
  DelimiterString := '"';
  SearchString := '"';
  SpaceString := ' ';
  ReturnString := ' ';
  ReturnString[1] := chr(CR);
  FindString := null;
  ReplaceString := null;
  UserName := Null;
  String50 := '                                                  ';
  DisplayString := Concat(String50, String50, String50, String50, String50, '        ');
  SearchOption := SBKanji;
  SpecialtyDOption := Communications;
  FontOption := 1;
  GrammarOption := 1;
  repeat
   GetItem(FontMenu, FontOption, FontName);
   If (FontName <> 'Geneva') then
    FontOption := FontOption + 1;
  until (FontName = 'Geneva');
{   SizeOption := FontSize;}
  SetFont(FontNum);
  DictvRef := 0;
  ArticlePtr := nil;
  EditFunction := false;
  If EditFunction then
   CurrentFunction := editCommand
  else
   CurrentFunction := LearnTextCommand;
```

```
  CurFunction := CurrentFunction;
  Result := GetVol(@DefaultVolName, DictvRef);
  vRef := DictvRef;
  if (Result = nsvErr) then
   DisplayDialog(OkD, 'Warning -- cannot open default volume !', ErrCode);
  WaitDialog := GetNewDialog(WaitDId, nil, nil);

  CheckItem(Menus[FunctionM], CurFunction, true);
  CheckItem(Menus[DictionaryM], CurDictionary, true);
  CheckItem(OptionsMenu, SearchOption, true);
  CheckItem(FontMenu, FontOption, true);
{   CheckItem(SizeMenu, SizeOption, true);}

  ErrCode := TEFromScrap;
  prRecHandle := THPrint(NewHandle(SIZEOF(TPrint)));
  CreateHashTable(WKHashTable);
  CreateHashTable(UnFHashTable);
 end; { Initialize }


end.
```

```
unit CAIFile;

{**************************************************************************}
{                                                                          }
{ This unit provides routines for handling file i/o, especially for opening and saving files. }
{                                                                          }
{**************************************************************************}
```



```
Interface


 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog, CAILinkedList, CAIUtilities, CAIInit;


 function RWFile (RW: RWType;
          origName: Str255;
          fType: OSType;
          var vRef: integer): str255;

 function ReadFile (fName: Str255;
          vRef: integer;
          OpenDB: Boolean): Boolean;

 function WriteFile (fName: Str255;
          vRef: integer;
          UseText: Boolean;
          SaveText: TEHandle): Boolean;

 procedure SaveFile (Save: Boolean;
          var Cancelled: Boolean);

 procedure OpenFieldFile (StartLoc: integer;
          FileName: str255;
          var TechnicalArea: InfoArrayPtr);

 procedure OpenUserFile (var WellArea, UnFamArea: InfoArrayPtr);

 function OpenTextDB (fName: str255;
          vRef: integer): Boolean;

 procedure OpenExistedFile (fName: str255;
          NewWindow: Boolean;
```

```
          var title: str255);

 procedure OpenFile (New: Boolean);


implementation


{ function to call SFGetFile to get the name of the file to be open    }
{ and SFPutFile to get the name of the file to be saved. Standard      }
{ dialogs are displayed. A null string is returned if the process      }
{ fails; otherwise the filename and vref are returned.                 }
 function RWFile;
  var
   where: point;
   reply: SFReply;
   TextType: SFTypelist;
 begin { RWFile }
  where.h := 80;
  where.v := 55;
  TextType[0] := fType;
  if (RW = ReadF) then                              { open a file }
   SFGetFile(where, Null, nil, 1, TextType, nil, reply)
  else                                              { save a file }
   SFPutFile(where, Null, origName, nil, reply);
  with reply do
   if not good then                                 { check if succeed }
    RWFile := Null
   else
    begin
     RWFile := fName;                               { return the file name  }
     vRef := vRefNum;
    end;
 end; { RWFile }


{ function to read the text of a file into a handled buffer.           }
{ It calls appropriate FS operations to handle the task.               }
{ Text file is put into FileHandle; dictionary into DictHandle;        }
{ TextDatabase into DBHandle. It closes the file if there occurs       }
{ an error.                                                            }
 function ReadFile;
  label
   10;
  var
   opened: Boolean;
   fSize: longint;
   fRef, result: integer;
   ErrorCode: OSErr;

  procedure ReadError (fName, errmsg: str255);
  begin { ReadError }
   DisplayDialog(OkD, Concat('Error while reading from "', fName, '": ', errmsg, ' !'), result);
```

```
    ReadFile := false;
    goto 10;
  end; { ReadError }

 begin { ReadFile }
  SetCursor(Watch^^);
  opened := false;
  ErrorCode := FSOpen(fName, vRef, fRef);              { open the file }
  if (ErrorCode <> noErr) then                         { check if error occurs}
   ReadError(fName, Concat('I/O error #', NumberToString(ErrorCode)))
  else
   begin
    opened := true;
    ErrorCode := GetEOF(fRef, fSize);                  { check if size < 32767}
    if (ErrorCode <> noErr) then
     ReadError(fName, Concat('I/O error #', NumberToString(ErrorCode)))
    else if (fSize > MaxInt) then
     ReadError(fName, ' file is too big to be edited')
    else
     begin
      ErrorCode := FSClose(fRef);                      { close file while allocating file buffer  }
      If OpenDB then                                   { check if to open text database  }
       DBHandle := NewHandle(fSize)
      else if (fName = CurDictName) then               { check if to open dictionary  }
       DictHandle := NewHandle(fSize)
      else
       FileHandle := NewHandle(fSize);                 { open a text                  }
      ErrorCode := FSOpen(fName, vRef, fRef);          { buffer is safely allocated   }
      If OpenDB then
       ErrorCode := FSRead(fRef, fSize, DBHandle^)     { read the text into buffer    }
      else if (fName = CurDictName) then
       ErrorCode := FSRead(fRef, fSize, DictHandle^)
      else
       ErrorCode := FSRead(fRef, fSize, FileHandle^);
      If (ErrorCode = noErr) then
       ReadFile := True
      else
       ReadError(fName, Concat('I/O error #', NumberToString(ErrorCode)))
     end;
   end;
10:
  If opened then
   ErrorCode := FSClose(fRef);
 end; { ReadFile }


{ function to write the text of a file into a file.    }
{ It calls appropriate FS operations to handle the task.  }
{ If closes the file if there occurs an error.         }
 function WriteFile;
  label
   20;
  var
```

```
    fSize: longint;
    fRef, result: integer;
    ErrorCode: OSErr;
    duplicated: Boolean;

  procedure WriteError (fName, errmsg: str255);
  begin { WriteError }
    DisplayDialog(OkD, Concat('Error while writing to "', fName, '": ', errmsg, ' !'), result);
    WriteFile := false;
    goto 20;
  end; { WriteError }

begin { WriteFile }
  SetCursor(Watch^^);
  ErrorCode := Create(fName, vRef, 'NELL', 'TEXT');   { create a file              }
  duplicated := (ErrorCode = dupFNErr);
  if (ErrorCode = noErr) or (duplicated) then         { check if duplicated       }
    begin
      ErrorCode := FSOpen(fName, vRef, fRef);
      if (ErrorCode <> noErr) then
        WriteError(fName, Concat('I/O error #', NumberToString(ErrorCode)))
      else
        begin                                         { new file to store the text}
          ErrorCode := SetEOF(fRef, 0);               { make sure it is empty     }
          if (ErrorCode <> noErr) then
            WriteError(fName, Concat('I/O error #', NumberToString(ErrorCode)))
          else
            begin
              if (UseText) and (SaveText <> nil) then     { save a generic text file  }
                begin
                  fSize := SaveText^^.teLength;
                  ErrorCode := FSWrite(fRef, fSize, SaveText^^.hText^);
                end
              else if (FrontWindow = TextWindow) then  {write out the text to file}
                begin
                  fSize := TEText^^.teLength;
                  ErrorCode := FSWrite(fRef, fSize, TEText^^.hText^);
                end
              else if (FrontWindow = TranslationWindow) then
                begin
                  fSize := TranslationText^^.teLength;
                  ErrorCode := FSWrite(fRef, fSize, TranslationText^^.hText^);
                end
              else if (FrontWindow = CharacterWindow) then
                begin
                  fSize := CharacterText^^.teLength;
                  ErrorCode := FSWrite(fRef, fSize, CharacterText^^.hText^);
                end;
              if (ErroRCode = noErr) then
                WriteFile := true
              else
                WriteError(fName, Concat('I/O error #', NumberToString(ErrorCode)))
            end;
```

```
      end;
    end
   else
    WriteError(fName, Concat('I/O error #', NumberToString(ErrorCode)));
20:
  ErrorCode := FSClose(fRef);
  ErrorCode := FlushVol(nil, fRef);
 end; { WriteFile }


{ procedure to save the text into a file with some file name other than    }
{ window titles.                                                          }
 procedure SaveFile;
  var
   result: integer;
   title, fName: str255;
 begin { SaveFile }
  SetCursor(Watch^^);
  if (FrontWindow = TextWindow) then
   GetWTitle(TextWindow, fName)                    { get window title      }
  else if (FrontWindow = TranslationWindow) then
   fName := TransWTitle
  else if (FrontWindow = CharacterWindow) then
   fName := CharWTitle;
  title := fName;                                  { if save as command    }
  if (not save) or (fName = SysTitle) or (fName = TransWTitle) or (fName = CharWTitle) then
   fName := RWFile(WriteF, fName, 'TEXT', vRef); { get new title          }
  Cancelled := (fName = Null);                     { check if cancelled    }
  if not Cancelled then
   if (fName = SysTitle) or (fName = TransWTitle) or (fName = CharWTitle) then
    DisplayDialog(OkD, Concat('Sorry! Cannot save a file named as ', fName, '. Please choose another
      name.'), result)
   else if (WriteFile(fName, vRef, false, nil)) then
    if (not save) or (title = SysTitle) then
     SetWTitle(TextWindow, fName);                 { rename the window     }
 end; { SaveFile }


{ procedure to open the file containing all information about technical areas and their files.   }
{ It reads in the information from startloc into technical area.                                 }
 procedure OpenFieldFile (StartLoc: integer;
           FileName: str255;
           var TechnicalArea: InfoArrayPtr);
  var
   result: integer;
   AreaText: TEHandle;
   OldPort: GrafPtr;
 begin { OpenFieldFile }
  TechnicalArea := nil;
  if (ReadFile(FileName, DictvRef, false)) then    { open technical field file  }
   begin
    GetPort(OldPort);
    NewText(FileHandle, AreaText);                 { create a text handle      }
```

```
      DBChar := TEGetText(AreaText);
      DBCount := StartLoc;
      SetPort(WaitDialog);
      ReadStr(true, AreaText^^.teLength, TechnicalArea);          { read file information    }
      TEDispose(AreaText);
      SetPort(OldPort);
     end
   else if (StartLoc = 0) then
    DisplayDialog(OkD, 'Sorry -- cannot select an article for you !!', result);
  end; { OpenFieldFile }


{ procedure to get the user information on his well-known area and   }
{ unfamiliar area.                                                  }
  procedure OpenUserFile (var WellArea, UnFamArea: InfoArrayPtr);
   var
    PersonalFile: str255;
  begin { OpenUserFile }
   if (UserName <> Null) then
    begin
     WellArea := nil;
     UnFamArea := nil;
     PersonalFile := Concat(UserName, PersonId);
     if (ReadFile(PersonalFile, DictvRef, false)) then      { open user personal database    }
      begin
       OpenFieldFile(0, PersonalFile, WellArea);           { get well-known area }
       OpenFieldFile(DBCount, PersonalFile, UnFamArea); { get unfamiliar area   }
      end;
     if (WellArea <> nil) then                             { check if no content    }
      if (WellArea^.InfoContent = Null) then
       DisposeInfoContent(WellArea);
     if (UnFamArea <> nil) then
      if (UnFamArea^.InfoContent = Null) then
       DisposeInfoContent(UnFamArea);
    end;
  end; { OpenUserFile }


{ procedure to open a file and build up translation data structure  }
{ for it if the database file is found.                             }
  function OpenTextDB (fName: str255;
            vRef: integer): Boolean;
   const
    Left = 20;
    Top = 65;
    Bottom = 85;
    Right = 210;
   var
    result: integer;
    ArticleNode, WellArea, UnFamArea: InfoArrayPtr;
    DBText: TEHandle;
    EncloseRect: Rect;
    OldPort: GrafPtr;
```

```
 begin { OpenTextDB }
  OpenTextDB := false;
  if not ReadFile(ConCat(fName, IDString), vRef, true) then { open text database   }
   DisplayDialog(OkD, Concat('Error -- cannot open database of ', fName, '; no translation available!!'),
      result)
  else                                          { build up trans. data structure  }
   begin
    GetPort(OldPort);
    ShowWindow(WaitDialog);
    SelectWindow(WaitDialog);
    DrawDialog(WaitDialog);
    SetPort(WaitDialog);
    SetRect(EncloseRect, Left, Top, Right, Bottom);
    FrameRect(EncloseRect);
    DisposeLinkedList;                          { dispose old data base        }
    OldSelStart := 0;                           { reset database parameters }
    OldSelEnd := 0;
    CurrentNode := 1;
    SearchOption := 1;
    FindString := null;
    ReplaceString := null;
    ArticlePtr := nil;
    OldPtr := nil;
    CreateSentenceList(ArticlePtr, DBText);     { rebuild data structures  }
    SetDText(WaitDialog, 1, 'Loading User" s Database ');
    EraseRect(EncloseRect);
    FrameRect(EncloseRect);
    CurSenPtr := ArticlePtr;
    OpenTextDB := true;
    if (UserName = Null) then                   { extract database info    }
     begin
      WellArea := nil;
      UnfamArea := nil;
     end
{   else if (WellArea = nil) and (UnfamArea = nil) then}
    else
     OpenUserFile(WellArea, UnFamArea);
    ClearHashTable(WKHashTable);                { clear old hash tables    }
    ClearHashTable(UnFHashTable);
    HashArea(WellArea, WKHashTable);
    HashArea(UnfamArea, UnFHashTable);
    HideWindow(WaitDialog);
    while GetNextEvent(updateMask, UpdateEvent) do
     HandleUpdate(UpdateEvent);
    ArticleNode := nil;                         { open article database file}
    if (DQCount >= 0) then                      { find double delimiter }
     begin
      OpenFieldFile((LoWord(DQCount) + Length(Concat(DoubleDQ, ReturnString))), Concat(fName,
      IDString), ArticleNode);
      if (ArticleNode <> nil) then
       UpdateDatabase(ArticleNode); { update database from translation info }
     end;
    DisposeInfoContent(ArticleNode);
```

```
    TEDispose(DBText);
    SetPort(OldPort);
   end;
 end; { OpenTextDB }


{ procedure to open either an existing file or a new file  }
 procedure OpenExistedFile (fName: str255;
           NewWindow: Boolean;
           var  title:  str255);
 begin { OpenExistedFile }
  if ReadFile(fName, vRef, false) then
   begin
    title := fName;
    while GetNextEvent(updateMask, UpdateEvent) do
     HandleUpdate(UpdateEvent);
    if NewWindow then                          { if to give a new window}
     begin
      NewTEHandle(TEText, fName, TextWindow, true);
      InvalRect(TextWindow^.portRect);         { force updating          }
      ShowInsertion(TextWindow, TEText, TELines, vTextSB, hTextSB);
      while GetNextEvent(updateMask, UpdateEvent) do
       HandleUpdate(UpdateEvent);
      if (CurrentFunction = LearnTextCommand) then
       TextOpened := OpenTextDB(title, vRef);  { open text data base     }
      while GetNextEvent(updateMask, UpdateEvent) do{ update all windows  }
       HandleUpdate(UpdateEvent);
     end;
   end
  else
   title := Null;
 end; { OpenExistedFile }


{ procedure to open either an existing file or a new one. If new is     }
{ chosen, the content of the text window will be lost.                  }
 procedure OpenFile;
  var
   Result: integer;
   title, fName: str255;
   Cancelled: Boolean;
 begin { OpenFile }
  SetCursor(Watch^^);
  Result := Ok;
  if (Result = Ok) then
   if New then              { to open a new file}
    begin
     Result := 3;
     GetWTitle(TextWindow, fName);
     if (fName = SysTitle) and (TEText^^.teLength > 0) then{ if an edit file    }
      DisplayDialog(YesNoD, Concat('Do you want to save ', fName, ' before opening a new file ?!'),
      result);
     while GetNextEvent(updateMask, UpdateEvent) do  { update all windows      }
```

```
        HandleUpdate(UpdateEvent);
      Cancelled := (result = Cancel);
      If (result = Ok) then                          { check if canceled    }
       SaveFile(false, Cancelled);                   { save the file        }
      If (not Cancelled) then
       begin
        fName := Null;
        NewTEHandle(TEText, fName, TextWindow, true);
        InvalRect(TextWindow^.portRect);             { force updating       }
        ShowInsertion(TextWindow, TEText, TELines, vTextSB, hTextSB);
        TextOpened := false;
       end
     end
    else                  { to open an existing file  }
     begin
      fName := RWFile(ReadF, Null, 'TEXT', vRef);    { get the file name  }
      If (fName <> Null) then                        { skip dictionary files}
       OpenExistedFile(fName, true, title)
     end;
 end; { OpenFile }


end.
```

```
unit CAIHighLevel;

{*************************************************************************}
{                                                                         }
{ This unit provides high level routines which require file i/o for the main program. These  }
{ routines will involve using dictionary files, user database, selecting article, etc. Usually, }
{ they will invoke utility routines to handle their task once the file is successfully opened.  }
{                                                                         }
{*************************************************************************}

{*************************************************************************}
{                                                                         }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All  }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or  }
{ combinations thereof, now known or later developed.                     }
{                                                                         }
{*************************************************************************}

interface


 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog, CAILinkedList, CAIUtilities, CAIPrint, CAIInit, CAIFile;


 procedure HandleQuit (var quit: Boolean);

 procedure SelectFile;

 procedure HandleCharInfo (UseString, FromArticle: Boolean;
           InString: str255);

 procedure HandleSearchDictionary;

 procedure HandleRevert;

 procedure HandleStartUp;


implementation


{ procedure to save the user level of Japanese proficiency on his personal database.  }
 procedure SaveUserDB;
  var
   result: integer;
   SaveUserFile, Cancelled: Boolean;
   WellArea, UnFamArea: InfoArrayPtr;
   UserText: TEHandle;
 begin { HandleQuit }
  HideAll;
```

```
  while (FrontWindow <> nil) do                    { close all active windows }
    HandleCloseWindow(FrontWindow);
  if (CurrentFunction <> EditCommand) then         { no work under edit mode }
    begin
      if (UserName = Null) then
        begin                                      { get the user name     }
          EditDialog('your user name', EditD, UserName, Cancelled);
          if (UserName = Null) then
            DisplayDialog(OkD, 'Sorry -- cannot save your Japanese proficiency level !', result);
        end;
      if (UserName <> Null) and (not Cancelled) then
        begin
          UserText := TENew(DestRect, DestRect);   { create new text handle  }
          TableToArea(WKHashTable, WellArea);
          PutInfo(UserText, WellArea);             { store well-known area info }
          TableToArea(UnFHashTable, UnFamArea);
          PutInfo(UserText, UnFamArea);            { store unfamiliar area info}
          SaveUserFile := (WriteFile(Concat(UserName, PersonId), DictvRef, true, UserText));
          TEDispose(UserText);
        end;
    end;
end; { SaveUserDB }


{ procedure to handle quitting.  The user is asked whether  }
{ to save any changes to the text file being edited.        }
procedure HandleQuit;
  var
    result:  integer;
    fName:  str255;
    Cancelled: Boolean;
begin { HandleQuit }
  GetWTitle(TextWindow, fName);
  if (fName = SysTitle) and (TEText^^.teLength > 0) then { if an edit file      }
    DisplayDialog(YesNoD, Concat('Do you want to save any changes to ', fName, ' before closing ?!'),
        result)
  else
    result := 3;
  while GetNextEvent(updateMask, UpdateEvent) do   { update all windows   }
    HandleUpdate(UpdateEvent);
  if (result = Ok) then                            { check if canceled  }
    SaveFile(false, Cancelled);                    { save the file      }
  SaveUserDB;
  quit := (result = 3) or ((result = Ok) and (not Cancelled));
end; { HandleQuit }


{ procedure to open a dictionary file. If not found, the user will be asked  }
{ to locate the dictionary.                                                  }
procedure OpenDictFile;
  var
    fName:  str255;
    result:  integer;
```

```
 begin { OpenDictFile }
  DictOpened := ReadFile(CurDictName, DictvRef, false);  { open selected dictionary }
  if not DictOpened then
   begin
    DisplayDialog(OkCanD, Concat('Error -- cannot open ', CurDictName, '; do you want to locate ',
      CurDictName, '?!'), result);
    if (Result = Ok) then
     begin
      fName := RWFile(ReadF, Null, 'TEXT', vRef);
      if (fName <> null) then                          { check if canceled        }
       begin
        CurDictName := fName;                          { update dictionary name   }
        DictOpened := ReadFile(CurDictName, DictvRef, false);
       end;
      if not DictOpened then
       DisplayDialog(OkD, Concat('Error -- cannot open ', CurDictName, '; no dictionary information
      available !!  Please try to select other dictionaries.'), result);
     end
    else
     begin
      DictOpened := false;                             { dictionary not found     }
      if (Result = 3) then
       DisplayDialog(OkD, 'Sorry no dictionary information for this character (dictionary not found) !',
      result);
     end;
   end;
 end; { OpenDictFile }


{ procedure to choose an article for the user according to his technical   }
{ area of interest as well as his proficiency level on Japanese. The      }
{ name of the article assigned to the user is store in the selectionfile.  }
{ The article will be opened once is found in the selectionfile associated }
{ with the user name.                                                      }
 procedure SelectFile;
  var
   SearchName, ArticleFile, title: str255;
   result, OldDBCount: integer;
   SaveSelection, Cancelled: Boolean;
   SelectionText: TEHandle;
   Offset: Longint;
 begin { SelectFile }
  EditDialog('your user name', EditD, UserName, Cancelled);          { get the user name }
  if ((UserName <> Null) and (not Cancelled)) then
   begin
    if (not ReadFile(SelectionFile, DictvRef, true)) then     { open selection file}
     DisplayDialog(OkD, 'Sorry -- cannot select an article for you !', result)
    else
     begin
      NewText(DBHandle, SelectionText);                       { find the user name}
      SearchName := Concat(ReturnString, UserName, DelimiterString);
      OffSet := Munger(SelectionText^^.hText, 0, POINTER(ord(@SearchName) + 1), Length(SearchName),
      nil, 0);
```

```
      If (OffSet >= 0) then
       begin
        GetUserInfo(SelectionText, OffSet, UserName);      { get file name      }
        GetUserField(OldDBCount, ArticleFile);
        If (ArticleFile <> Null) then
         OpenExistedFile(ArticleFile, true, title);        { open the document}
       end
      else
       DisplayDialog(OkD, 'Sorry -- cannot select an article for you !', result);
      TEDispose(SelectionText);
     end;
   end
  else
   begin                                                   { user name unknown}
    UserName := Null;
    If (not Cancelled) then
     DisplayDialog(OkD, 'Sorry cannot select an article for you !', result);
   end;
 end; { SelectFile }


{ procedure to handle showing character information. It attempts to open the selected }
{ dictionary. If failed, the user will be prompted to locate it. Succeeding in finding  }
{ the dictionary file, it will print out character information. The key string for       }
{ searching the dictionary will be InString if UseString is true. Otherwise, it will use }
{ the edit dialog with initial contents of hilited text from the front window.          }
 procedure HandleCharInfo;
  var
   JapString, Extension, OldDictName: str255;
   Valid, found, InfoFound, Info1Found, Exit: Boolean;
   ExtenNumber: Longint;
   result, limit: integer;
 begin { HandleCharInfo }
  SetCursor(Watch^^);
  Valid := UseString;
  If UseString then                                        { get the search string }
   JapString := InString                                   { use input string      }
  else
   GetJapString(JapString, Valid);                         { get string from edit dialog  }
  If Valid then
   begin
    OldDictName := CurDictName;
    If (CurDictionary = CharacterDictionary) and (SearchOption <> SBEnglish) and (SearchOption <>
      SBKatakana) and (SearchOption <> SBHiragana) then
     begin                                                 { from character dictionary  }
      GetDictName(SBKanji, JapString, Extension, found);
      If found then                                        { select the dictionary file that contains the string
       }
       begin
        CurDictName := Concat(CDKanji, Extension);
        OpenDictFile;                                      { open dictionary file  }
        JapString := Concat(JapString, SpaceString);
        If DictOpened then                                 { show character information }
```

```
        ShowCharInfo(JapString, FromArticle, InfoFound, Exit);
      end
    end
  else if (CurDictionary = GrammarDictionary) or (CurDictionary = GeneralDictionary) or
    (CurDictionary = SpecialtyDictionary) or ((CurDictionary = CharacterDictionary) and
    ((SearchOption = SBEnglish) or (SearchOption = SBKatakana) or (SearchOption = SBHiragana))) then
    begin                                    { from general or specialty dictionary }
     Extension := '1';
     InfoFound := false;
     Info1Found := false;
     if (CurDictionary = CharacterDictionary) then { search limit for dictionary }
      limit := CDFiles
     else if (CurDictionary = GeneralDictionary) then
      limit := GDFiles
     else if (CurDictionary = SpecialtyDictionary) then
      begin
       if (SpecialtyDOption = Communications) then
        limit := SDCommFiles
      end
     else if (CurDictionary = GrammarDictionary) then
      limit := GramFiles;
     repeat
      if (CurDictionary = CharacterDictionary) then{ go thru all dictionary files }
       CurDictName := Concat(CDKanji, Extension)
      else if (CurDictionary = GeneralDictionary) then
       CurDictName := Concat(GDKanji, Extension)
      else if (CurDictionary = SpecialtyDictionary) then
       case SpecialtyDOption of
        Communications:
         CurDictName := Concat(SDComm, Extension);
       end
      else if (CurDictionary = GrammarDictionary) then
       CurDictName := Concat(Grammar, Extension);
      OpenDictFile;
      if DictOpened then                     { show word information }
       ShowCharInfo(JapString, FromArticle, InfoFound, Exit);
      Info1Found := Info1Found or InfoFound;
      StringToNum(Extension, ExtenNumber);   { to get next file index }
      Extension := NumberToString(succ(ExtenNumber));
      StringToNum(Extension, ExtenNumber);
     until (ExtenNumber > limit) or Exit;    { check if no more files }
     if not Exit then
      if (not Info1Found) then
       DisplayDialog(OkD, Concat('Sorry no information for these characters from the ', OldDictName,
    '!'), result)
      else if (CurDictionary = GrammarDictionary) then
       DisplayDialog(OkD, Concat('Sorry no more information for these characters from the ',
    OldDictName, '!'), result)
      else
       begin
        TEKey(chr(CR), CharacterText);       { readjust char window }
        ShowInsertion(CharacterWindow, CharacterText, CharLines, vCharSB, hCharSB);
       end
```

```
      end;
     CurDictName := OldDictName;
    end;
  end; { HandleCharInfo }


{ procedure to search the string or the number to be searched in the dictionary.}
{ Type and range checking is enforced so that only valid values are searched.   }
  procedure GetSearchValue (var ValueValid, Cancel: Boolean;
            var ResultString: str255;
            var ResultNumber: integer);
   var
    Valid, Cancelled: Boolean;
    SearchNumber: Longint;
    i, strlength: integer;
    msg, editmsg: str255;
  begin { GetSearchValue }
   case SearchOption of
    SBKanji, SBKatakana, SBHiragana:                    { display heading message }
     begin
      msg := 'a string of characters';
      if (CurDictionary = CharacterDictionary) then
       strlength := 1
      else if (CurDictionary = GeneralDictionary) then
       strlength := 2;
      if ((CurDictionary = CharacterDictionary) or (CurDictionary = GeneralDictionary)) and
       (SearchOption = SBKanji) then
        msg := Concat('a string of Kanji (only ', NumberToString(strlength), ' character(s))');
     end;
    SBNelsonDictionary:
     msg := Concat('a Nelson''s Dictionary Ref. number (1 - ', NumberToString(NelsonInputMax), ') ');
    SBRadicals:
     msg := Concat('a radical number (1 - ', NumberToString(RadicalInputMax), ') ');
    SBStrokeCount:
     msg := Concat('a number of stroke counts (1 - ', NumberToString(StrokeInputMax), ') ');
    SBEnglish:
     msg := 'a string of characters';
    otherwise
   end;
   Valid := (SearchOption >= SBKanji) and (SearchOption <= SBEnglish);
   SearchNumber := 0;
   if (CurDictionary = GrammarDictionary) then
    EditDialog(msg, GrammarD, editmsg, Cancelled)          { get the search key       }
   else
    EditDialog(msg, EditD, editmsg, Cancelled);
   if (not Cancelled) then
    begin
     case SearchOption of
      SBKanji:
       if ((Length(editmsg) <= CharLengthMax) and (CurDictionary = CharacterDictionary)) or
       ((Length(editmsg) <= WordLengthMax) and (CurDictionary = GeneralDictionary)) then
         CheckValidType(Kanji, editmsg, Valid);            { check if Kanji character }
      SBHiragana:
```

```
        CheckValidType(Hiragana, editmsg, Valid);          { check if Hiragana character }
       SBKatakana:
        CheckValidType(Katakana, editmsg, Valid);          { check if Katakana character }
       SBNelsonDictionary, SBRadicals, SBStrokeCount:
        If (Length(editmsg) >= 1) and (Length(editmsg) <= Length(NumberToString(NelsonInputMax))) then
         begin                                             { if length within limits    }
          for i := 1 to Length(editmsg) do
           If Valid then                                   { if all chars are digits    }
            Valid := ((ord(editmsg[i]) >= $30) and (ord(editmsg[i]) <= $39));
          If Valid then
           begin
            StringToNum(editmsg, SearchNumber);            { if in valid ranges         }
            Valid := (SearchNumber >= 1) and (((SearchNumber <= NelsonInputMax) and (SearchOption =
       SBNelsonDictionary)) or ((SearchNumber <= RadicalInputMax) and (SearchOption = SBRadicals)) or
       ((SearchNumber <= StrokeInputMax) and (SearchOption = SBStrokeCount)));
           end;
         end;
       SBEnglish:
        CheckValidType(English, editmsg, Valid);           { check if English character  }
       otherwise
      end;
      If Valid then
       begin
        ResultString := editmsg;                           { return search key if Ok  }
        ResultNumber := SearchNumber;
       end;
     end;
    ValueValid := Valid;
    Cancel := Cancelled;
  end; { GetSearchValue }


{ procedure to search the dictionary according to the specified search option    }
{ SBKanji              : search dictionary file by Kanji characters             }
{ SBHiragana           : search dictionary file by Hiragana characters          }
{ SBKatakana           : search dictionary file by Katakana characters          }
{ SBNelsonDictionary   : search dictionary file by Nelson's Dictionary entry    }
{ SBStrokeCount        : search dictionary file by stroke counts                }
{ SBRadicals           : search dictionary file by radical number               }
{ SBEnglish            : search dictionary file by English translation          }
{ The Katagana and Hiragana must be converted to Kanji characters before the    }
{ searching can be proceeded. Only a string of one character long is allowed    }
{ for the character dictionary and two characters for the compound dictionary. }
  procedure HandleSearchDictionary;
   var
    Result, SearchDictNumber: integer;
    SearchDictString, OldDictName, KanjiString: str255;
    Valid, Cancel, found: Boolean;
  begin { HandleSearchDictionary }
   If (CurDictionary < CharacterDictionary) and (CurDictionary > SpecialtyDictionary) then
    DisplayDialog(OkD, 'Sorry search dictionary is not available for this dictionary !', result)
   else
    begin                                                  { get the search string    }
```

```
    GetSearchValue(Valid, Cancel, SearchDictString, SearchDictNumber);
    if (not Cancel) then
     If Valid then
      begin
       SetCursor(Watch^^);
       while GetNextEvent(updateMask, UpdateEvent) do
        HandleUpdate(UpdateEvent);                    { update all windows      }
       case SearchOption of
        SBKanji, SBKatakana, SBHiragana, SBEnglish:   { invoke charinfo with search key provided }
         begin
          If (SearchOption = SBEnglish) then
           SearchDictString := Concat(SpaceString, SearchDictString);
          HandleCharInfo(true, false, SearchDictString);
         end;
        SBNelsonDictionary, SBRadicals, SBStrokeCount:
         begin
          OldDictName := CurDictName;
          case SearchOption of                        { use selected dictionary file }
           SBNelsonDictionary:
            CurDictName := CDNelson;                  { change current dict name}
           SBRadicals:
            CurDictName := CDRadical;
           SBStrokeCount:
            CurDictName := CDStroke;
          end;
          OpenDictFile;                               { open the dictionary file  }
          If DictOpened then
           ShowSearchInfo(SearchDictString, KanjiString);{ show the information    }
          CurDictName := OldDictName;                 { reset dictionary name   }
          If (SearchOption = SBNelsonDictionary) then
           HandleCharInfo(true, false, KanjiString);
         end;
        otherwise
         DisplayDialog(OkD, 'Sorry no such search option is available !!', result);
       end;
      end
     else
      DisplayDialog(OkD, Concat('Sorry the search input is invalid !!  No information available from the ',
      CurDictName), result);
   end;
 end; { HandleSearchDictionary }


{ procedure to revert the text window contents to the last save version     }
 procedure HandleRevert;
  var
   result: integer;
   fName, title: str255;
 begin { HandleRevert }
  If (FrontWindow <> TextWindow) then                 { only the text window }
   DisplayDialog(OkD, 'Sorry only the text window can be reverted !', result)
  else
   begin
```

```
    GetWTitle(TextWindow, fName);
    If (fName = SysTitle) then
     DisplayDialog(OkD, Concat('Sorry cannot revert ', fName, ' to the last saved version !'), result)
    else
     begin
      DisplayDialog(OkCanD, Concat('Revert ', fName, ' to the last saved version ?!'), result);
      while GetNextEvent(updateMask, UpdateEvent) do { update all windows}
       HandleUpdate(UpdateEvent);
      If (result = Ok) then                          { check if canceled  }
       OpenExistedFile(fName, true, title);          { get the old file content  }
      If (title = Null) then
       DisplayDialog(OkD, Concat('Sorry cannot revert ', fName, ' to the last saved version !'), result);
     end;
   end;
 end; { HandleRevert }


{ procedure to handle openning up any text file when startup (by Mike Hussey)  }
{ It handles printing the selected file (and then quit) and open up a file from  }
{ the Finder if the user double clicks on the file icon to launch the Nihongo tutor. }
{ If the tutor is started up instead, it will automatically select a file for the  }
{ user based on the information stored in the selection file by the administrator.}
 procedure HandleStartUp;
  var
   FinderMessage, nDocs, result: integer;
   DocInfo: AppFile;
   title:  str255;
 begin { HandleStartUp }
  FixCursor;                                { set the correct cursor  }
  SystemTask;                               { handle system task      }
  TEIdle(TEText);                           { make the insertion blink }
  CountAppFiles(FinderMessage, nDocs);
  If (FinderMessage = appPrint) then        { to print a file         }
   begin
    If (nDocs > 0) then                     { if a document is selected}
     begin
      GetAppFiles(nDocs, DocInfo);          { get the last file       }
      with DocInfo do
       If (fType = 'TEXT') then
        begin
         vRef := vRefNum;                   { specify the volume      }
         CurrentFunction := EditCommand;
         OpenExistedFile(fName, true, title);{ open the document     }
         HandlePrint;                       { print the last document  }
         ClrAppFiles(nDocs);                { tell Finder it is processed}
        end;
     end;
    ExitToShell;
   end
  else If (FinderMessage = appOpen) then    { to open a document      }
   If (nDocs > 0) then                      { if a document is selected}
    begin
     GetAppFiles(nDocs, DocInfo);           { get the last file       }
```

```
      with DocInfo do
       if (fType = 'TEXT') then
         begin
          vRef := vRefNum;                    { specify the volume      }
          CurrentFunction := EditCommand;
          CheckItem(Menus[FunctionM], CurFunction, false);
          CurFunction := CurrentFunction;     { reset the check mark    }
          CheckItem(Menus[FunctionM], CurFunction, true);
          OpenExistedFile(fName, true, title);{ open the document       }
          ClrAppFiles(nDocs);                 { tell Finder it is processed}
         end;
     end
    else                                      { start up from application}
     SelectFile;                              { select file for the user  }
  end; { HandleStartUp }


end.
```

```
program CAIMain;

{*****************************************************************}
{                                                                 }
{ This program contains the main evnt loop and all top level routines of the editor/translator.}
{ It also uses some routines from the  adjust, dialog, linked list, utilities, init, print, file,  }
{ and highlevel  unit.                                            }
{                                                                 }
{*****************************************************************}

{*****************************************************************}
{                                                                 }
{ © Copyright, 1989 by Purdue Research Foundation, West Lafayette, Indiana 47907. All  }
{ Rights Reserved. Unless permission is granted, this material shall be copied, reproduced or}
{ coded for reproduction by any electrical, mechanical or chemical processes, or      }
{ combinations thereof, now known or later developed.             }
{                                                                 }
{*****************************************************************}

 uses
  PrintTraps, CAIGlobals, CAIAdjust, CAIDialog, CAILinkedList, CAIPrint, CAIInit, CAIUtilities, CAIFile,
      CAIHighLevel;


{ procedure to handle the command when an item in a menu is selected      }
{ there will be a check mark beside the current selected item of the function,  }
{ and the dictionary  menu.                                               }
 procedure DoCommand (MenuChoice: longint;
           var done: Boolean);
  var
   theMenu, theItem, EditItem, temp, result: Integer;
   fName, name, SizeName: Str255;
   Cancelled: Boolean;
 begin { DoCommand }
  If menuChoice <> 0 then          { a menu item is selected  }
   begin
    theMenu := HiWord(menuChoice);
    theItem := LoWord(menuChoice);

    case theMenu of
     AppleId:                      { in apple menu      }
      begin
       GetItem(Menus[AppleM], theItem, name);
       If (theItem = AboutItem) then
        HandleAbout
       else If (theItem > 2) then
        temp := OpenDeskAcc(name);
      end;

     FileId:                       { in file menu       }
      case theItem of
       NewCommand:
        OpenFile(true);
```

```
      OpenCommand:
       OpenFile(false);
      CloseCommand:
       HandleCloseWindow(FrontWindow);
      SaveCommand:
       SaveFile(true, Cancelled);
      SaveAsCommand:
       SaveFile(false, Cancelled);
      RevertCommand:
       HandleRevert;
      PageSetUpCommand:
       HandlePageSetUp;
      PrintCommand:
       HandlePrint;
      QuitCommand:          { quit the program     }
       HandleQuit(done);
      otherwise
     end;

    EditId:
     begin                  { set Edititem correctly  }
      EditItem := theItem;
      if not (SystemEdit(theItem - 1)) then
       case EditItem of
        CutCommand:
         TECut(TEText);
        CopyCommand:
         if (FrontWindow = TextWindow) then
          TECopy(TEText)
         else if (FrontWindow = TranslationWindow) then
          TECopy(TranslationText)
         else if (FrontWindow = CharacterWindow) then
          TECopy(CharacterText);
        PasteCommand:
         begin
          TEPaste(TEText);
          if (TEText^^.teLength > 31000) then
           DisplayDialog(OkD, Concat('WARNING: The ', SysTitle, ' window is almost full! Try to avoid
adding more context.'), result);
         end;
        ClearCommand:
         TEDelete(TEText);
        SelectAllCommand:
         TESetSelect(0, MaxInt, TEText);
        FindWhatCommand:
         GetFindReplaceString;
        FindCommand:
         Find;
        ReplaceCommand:
         Replace(TEText);
        otherwise
       end;
      ShowInsertion(TextWindow, TEText, TELines, vTextSB, hTextSB);
```

```
  end;

FunctionMId:            { choose the appropriate function }
  begin
   CheckItem(Menus[FunctionM], CurFunction, false);
   case theItem of
    EditCommand:
     begin
      CurrentFunction := theItem;
      TextOpened := false;
     end;
    LearnTextCommand:
     begin
      CurrentFunction := theItem;
      GetWTitle(TextWindow, fName);
      if (fName <> SysTitle) and (not TextOpened) then
       TextOpened := OpenTextDB(fName, DictvRef);
     end;
    TranslateCommand:
     begin
      FullTranslate := false;
      HandleTranslation;
      if (TranslationText^^.teLength > 31000) then
       DisplayDialog(OkD, Concat('WARNING: The ', TransWTitle, ' window is almost full! Try to
avoid adding more context.'), result);
     end;
    FullTranslationCommand:
     begin
      FullTranslate := true;
      HandleTranslation;
      if (TranslationText^^.teLength > 31000) then
       DisplayDialog(OkD, Concat('WARNING: The ', TransWTitle, ' window is almost full! Try to
avoid adding more context.'), result);
     end;
    CharacterInfoCommand:
     begin
      HandleCharInfo(false, true, null);
      if (CharacterText^^.teLength > 31000) then
       DisplayDialog(OkD, Concat('WARNING: The ', CharWTitle, ' window is almost full! Try to avoid
adding more context.'), result);
     end;
    SearchDictCommand:
     ;
    RadicalInfoCommand:
     HandleRadicalInfo;
    FontCommand:
     ;
    SizeCommand:
     ;
    otherwise
   end;
   CurFunction := theItem;
   CheckItem(Menus[FunctionM], CurFunction, true);
```

```
    end;

  DictionaryMId:      { select the appropriate dictionary }
    begin
      if (CurDictionary <> SpecialtyDictionary) then
        CheckItem(Menus[DictionaryM], CurDictionary, false)
      else
        CheckItem(SpecialtyDMenu, SpecialtyDOption, false);
      case theItem of
        CharacterDictionary:
          CurDictName := CharDictionary;
        GeneralDictionary:
          CurDictName := GenDictionary;
        SpecialtyDictionary:
          ;
        GrammarDictionary:
          CurDictName := GramDictionary;
        otherwise
      end;
      if (CurDictName <> CharDictionary) and (SearchOption >= SBNelsonDictionary) and (SearchOption
    <= SBStrokeCount) then
        begin       { only search by Kanji, Hiragana, and Katagana are provided for these dictionaries }
          DisplayDialog(OkD, 'Search option is now reset to be "Search by Kanji" !', result);
          CheckItem(OptionsMenu, SearchOption, false);
          SearchOption := SBKanji;                          { reset the search option     }
          CheckItem(OptionsMenu, SearchOption, true);
        end;
      CurDictionary := theItem;
      CheckItem(Menus[DictionaryM], CurDictionary, true);
    end;

  WindowId:           { select the appropriate window }
    case theItem of
      TextW:
        begin         { make it the front active window}
          ShowWindow(TextWindow);
          SelectWindow(TextWindow);
        end;
      TranslationW:
        begin
          ShowWindow(TranslationWindow);
          SelectWindow(TranslationWindow);
        end;
      CharacterW:
        begin
          ShowWindow(CharacterWindow);
          SelectWindow(CharacterWindow);
        end;
      otherwise
    end;

  SearchOptionMId:    { select search options          }
    begin
```

```
      CheckItem(OptionsMenu, SearchOption, false);
      SearchOption := theItem;
      HandleSearchDictionary;
      CheckItem(OptionsMenu, SearchOption, true);
    end;

  SpecialtyDMId:      { select specialty dictionary     }
    begin
      if (CurDictionary <> SpecialtyDictionary) then
        CheckItem(Menus[DictionaryM], CurDictionary, false)
      else
        CheckItem(SpecialtyDMenu, SpecialtyDOption, false);
      SpecialtyDOption := theItem;
      CurDictionary := SpecialtyDictionary;
      CurDictName := SpecDictionary;
      if (SearchOption >= SBNelsonDictionary) and (SearchOption <= SBStrokeCount) then
        begin       { only search by Kanji, Hiragana, and Katagana are provided for specialty dictionary
        }
          DisplayDialog(OkD, 'Search option is now reset to be "Search by Kanji" !', result);
          CheckItem(OptionsMenu, SearchOption, false);
          SearchOption := SBKanji;                    { reset the search option     }
          CheckItem(OptionsMenu, SearchOption, true);
        end;
      case theItem of
        Communications:
          ;
        otherwise
      end;
      CheckItem(SpecialtyDMenu, SpecialtyDOption, true);
    end;

  FontMId:            { select font type   }
    begin
      CheckItem(FontMenu, FontOption, false);
      FontOption := theItem;
      GetItem(FontMenu, FontOption, fontName);
      GetFNum(fontName, FontNum);
      ChangeFont(FontNum, FontSize, TextWindow, TEText, TELines, vTextSB);
      ChangeFont(FontNum, FontSize, TranslationWindow, TranslationText, TransLines, vTransSB);
      ChangeFont(FontNum, FontSize, CharacterWindow, CharacterText, CharLines, vCharSB);
      SetFont(FontNum);
      CheckItem(FontMenu, FontOption, true);
    end;

  SizeMId:            { select font size   }
    begin
      CheckItem(SizeMenu, SizeOption, false);
      SizeOption := theItem;
      GetItem(SizeMenu, SizeOption, SizeName);
      ReadString(SizeName, FontSize);
      ChangeFont(FontNum, FontSize, TextWindow, TEText, TELines, vTextSB);
      ChangeFont(FontNum, FontSize, TranslationWindow, TranslationText, TransLines, vTransSB);
      ChangeFont(FontNum, FontSize, CharacterWindow, CharacterText, CharLines, vCharSB);
```

```
        CheckItem(SizeMenu, SizeOption, true);
       end;

      otherwise
     end;
     HiLiteMenu(0);        { Complement the menu title  }
    end;
  end; { DoCommand }


{ procedure to deal the case when the mouse is down on the screen        }
  procedure DealWithMouseDowns (theEvent: EventRecord;
            var done: Boolean);
   var
    windowLoc, result: Integer;
    windowChoice: WindowPtr;
    anEvent: EventRecord;
    newWindowSize, menuChoice: Longint;
    Extended: Boolean;
  begin { DealWithMouseDowns }
   windowLoc := FindWindow(theEvent.where, windowChoice);

   case windowLoc of
    inSysWindow:                    { mouse is down in system window }
     SystemClick(theEvent, windowChoice);

    inDrag:                         { mouse is down in drag bar area     }
     DragWindow(windowChoice, theEvent.where, dragRect);

    inDesk:
     ;                              { mouse is down in desktop           }

    inMenuBar:                      { mouse is down in menubar           }
     begin
      menuChoice := MenuSelect(theEvent.Where);
      DoCommand(menuChoice, done); { select an item from the menu       }
     end;

    otherwise
     if (windowChoice <> FrontWindow) then
      SelectWindow(Windowchoice)
     else
      case windowLoc of
       inGoAway:                    { mouse is down in go away region   }
        if (TrackGoAway(windowChoice, theEvent.where)) then
         HandleCloseWindow(FrontWindow);

       inGrow:                      { mouse is down in grow box          }
        begin
         newWindowSize := Growwindow(windowChoice, theEvent.where, GrowLimitRect);
         if (newWindowSize <> 0) then
          HandleGrow(windowChoice, LoWord(newWindowSize), HiWord(newWindowSize));
        end;
```

```
        inZoomIn:                       { mouse is down in zoom in box      }
          HandleZoom(windowChoice, theEvent.Where, inZoomIn);

        inZoomOut:                      { mouse is down in zoom out box     }
          HandleZoom(windowChoice, theEvent.Where, inZoomOut);

        inContent:                      { mouse is down in window content   }
          begin
            Extended := (BitAnd(theEvent.modifiers, ShiftKey) <> 0);
            HandleContent(theEvent.Where, windowChoice, Extended);
          end;

        otherwise
          begin
            SysBeep(10);                { system beep if not in any of the above areas }
            repeat
            until GetNextEvent(MUpMask, anEvent);
          end;
      end;
    end;
  end; { DealWithMouseDowns }


{ procedure to deal the case when a key is pressed. If the command key is    }
{ pressed, it is equivalent to a menu item selection.                        }
  procedure DealWithKeyDowns (theEvent: EventRecord;
            var done: Boolean);
    var
      CharCode, result: Integer;
      TabString: str255;
  begin { DealWithKeyDowns }
    CharCode := BitAnd(theEvent.Message, charCodeMask);
    { check if if a command key is pressed                    }
    if (BitAnd(theEvent.modifiers, cmdKey) = cmdKey) then
      DoCommand(MenuKey(chr(CharCode)), done)
    else if (CurrentFunction = EditCommand) and (FrontWindow = TextWindow) then
      if ((CharCode >= SP) or (CharCode = CR) or (CharCode = BS) or (CharCode = Tab)) or
         ((BitAnd(theEvent.modifiers, optionKey) = optionKey)) then
        begin                                   { check if the option key is pressed  }
          if (CharCode = Tab) then
            begin
              TabString := '        ';
              TEInsert(Pointer(Ord(@TabString) + 1), Length(TabString), TEText);
            end
          else
            TEKey(chr(CharCode), TEText);       { enter only printable characters}
          ShowInsertion(TextWindow, TEText, TELines, vTextSB, hTextSB);
          if (TEText^^.teLength > 31000) then
            DisplayDialog(OkD, Concat('WARNING: The ', SysTitle, ' window is almost full! Try to avoid adding
            more context.'), result);
        end
      else
```

```
    SysBeep(10);
  end; { DealWithKeyDowns }


{ procedure to deal the case when windows become active and inactive      }
  procedure DealWithActivateEvents (theEvent: EventRecord);
    const
      ChangeFlag = $0002;                    { mask for event modifier }
    var
      window: WindowPtr;
      result: integer;
  begin { DealWithActivateEvents }
    window := WindowPtr(theEvent.message);
    SetPort(window);
    if (BitAnd(theEvent.modifiers, activeFlag) <> 0) then
      begin                                  { activate window  }
        FixedCursor := false;
        if (window = TextWindow) then
          HandleActivate(TEText, vTextSB, hTextSB, Active)
        else if (window = TranslationWindow) then
          HandleActivate(TranslationText, vTransSB, hTransSB, Active)
        else if (window = CharacterWindow) then
          HandleActivate(CharacterText, vCharSB, hCharSB, Active);
        if (BitAnd(theEvent.modifiers, ChangeFlag) <> 0) then
          result := TEFromScrap;             { get content from scrap  }
        DisableItem(Menus[EditM], UndoCommand);
      end
    else
      begin                                  { deactivate window   }
        if (window = TextWindow) then
          HandleActivate(TEText, vTextSB, hTextSB, InActive)
        else if (window = TranslationWindow) then
          HandleActivate(TranslationText, vTransSB, hTransSB, InActive)
        else if (window = CharacterWindow) then
          HandleActivate(CharacterText, vCharSB, hCharSB, InActive);
        if (BitAnd(theEvent.modifiers, ChangeFlag) <> 0) then
          begin
            result := ZeroScrap;             { write out scrap contents}
            result := TEToScrap;
          end;
        EnableItem(Menus[EditM], UndoCommand);
      end;
    if (window = TextWindow) or (window = TranslationWindow) or (window = CharacterWindow) then
      DrawGrowIcon(window);                  { draw grow size box  }
  end; { DealWithActivateEvents }


{ procedure to handle events when a disk is inserted }
  procedure DealWithDiskEvents (theEvent: EventRecord);
    var
      mousePt: point;
      result: integer;
  begin { DealWithDiskEvents }
```

```
  if (HiWord(theEvent.message) <> noErr) then
    begin
      mousePt.h := 50;
      mousePt.v := 50;
      DILoad;                                  { initialize the disk }
      result := DIBadMount(mousePt, theEvent.message);
      DIUnload;
    end;
 end; { DealWithDiskEvents }


{ procedure to wrap up when the program quits   }
 procedure WrapUp;
  var
    result: integer;
 begin { WrapUp }
  TEDispose(TEText);                           { dispose text contents}
  TEDispose(TranslationText);
  TEDispose(CharacterText);
  CloseDialog(WaitDialog);                     { dispose dialog        }
  DisposeWindow(TextWindow);                   { dispose windows       }
  DisposeWindow(TranslationWindow);
  DisposeWindow(CharacterWindow);
  DisposeLinkedList;                           { dispose data structures}
  if (TEGetScrapLen > 0) then                  { write out scrap content}
    begin
      result := ZeroScrap;
      result := TEToScrap;
    end;
 end; { WrapUp }


{ procedure to call the right procedure for handling the event  }
{ in the event queue. The event loop continues until the user   }
{ quits the program.                                            }
 procedure EventLoop;
  var
    theEvent: EventRecord;
    done: Boolean;
 begin { EventLoop }
  done := false;
  repeat
    FixCursor;                          { set the correct cursor  }
    SystemTask;                         { handle system task      }
    TEIdle(TEText);                     { make the insertion blink }

    if GetNextEvent(EveryEvent, theEvent) then
      begin
        HandleMenu;
        case theEvent.What of
          nullEvent:
            ;
          mouseDown:
```

```
        DealWithMouseDowns(theEvent, done);
      keyDown, autoKey:
        DealWithKeyDowns(theEvent, done);
      activateEvt:
        DealWithActivateEvents(theEvent);
      updateEvt:
        HandleUpdate(theEvent);
      diskEvt:
        DealWithDiskEvents(theEvent);
      otherwise
    end;
  end;
 until done;
 WrapUp;                          { wrap up the program }
end; { EventLoop }


{ *** THE MAIN PROGRAM *** }
begin { main }
 Initialize;
 HandleStartUp;
 EventLoop;
end. { main }
```