- *Clear_Tune* : Clears all the variables that are used in the optimisation process.
- *Change_Optimise* : Performs optimisation on the magnitude or phase of a node voltage, branch current or branch admittance.
- *Optimise_Component* : Performs optimisation on a component.
- *Tune* : The main optimising (tuning) subprogram that is called by several other subprograms and does the maximising, minimising, setting, etc. The next two subprograms are declared within this one.
- *Optimum_Analysis* : Adjusts the netlist for optimising.
- *Optimising* : Optimises using the Simplex algorithm.
- *Clear* : Clears all the optimisation variables if the user agrees.
- *Find_Bandwidth* : Finds the bandwidth, upper, centre and lower frequencies, as well as maximum gain and the Q-factor of a bandpass response circuit.
- *Insert* : Defines new variables that are to be used in the optimisation process : the component's value, it's magnitude or phase, or the operating frequency.
- *Maximise* : Maximises the response of a specific node pair by tuning selected component values and / or frequencies.
- *Minimise* : Minimises the response of a specific node pair by tuning selected component values and / or frequencies.
- *Set* : Tunes component values and / or frequency so that the response of the selected node pair will be set at a specific value.
- *Show* : Displays the list of variables that are to be used in the optimisation.

4.16 Plotter

Needed Units (Interface)	Needed Units (Body)	Units Using This Unit
Analysers (Declared within)	Graphs Unchecked_Deallocation	Scheduler

4.16.1 Purpose

It holds all the analysers that display their results on the graphic screen. Thus they are separated from the other analysers.

4.16.2 Data Types and Structures

The HARMONIC_ACCESS_TYPE is a dynamic access type which points to a record HARMONIC_TYPE. This record consists of five array types which hold the *Fourier* plot values. The Y_ACCESS_TYPE is a dynamic access type which points to a 2D array Y_VALUE_TYPE. This array type holds the

Circuit_Response plot values. The POLAR_ACCESS_TYPE is a dynamic access type which points to a record POLAR_TYPE. This record consists of two array types which hold the *Polar* plot values.

```
ABS_MAX_HARMONICS : constant := 1000;

type HARMONIC_ARRAY_TYPE is array (0..ABS_MAX_HARMONICS) of REAL_TYPE;

type HARMONIC_TYPE is
   record
      MAGNIL,
      MAGNOP,
      PHASEL,
      PHASEOP,
      Y_VALUES   : HARMONIC_ARRAY_TYPE;
   end record;

type HARMONIC_ACCESS_TYPE is access HARMONIC_TYPE;

HARMONICS : HARMONIC_ACCESS_TYPE;


type Y_VALUE_TYPE  is array (0..NX_PIXELS, 1..5) of REAL_TYPE;
type Y_ACCESS_TYPE is access Y_VALUE_TYPE;

Y_VALUES : Y_ACCESS_TYPE;


MAX_POINTS : constant := 2500;

type POLAR_ARRAY is array (1..MAX_POINTS) of REAL_TYPE;

type POLAR_TYPE is
   record
      REAL,
      IMAG : POLAR_ARRAY;
   end record;

type POLAR_ACCESS is access POLAR_TYPE;

POLAR_VOLT : POLAR_ACCESS;
```

4.16.3 Methods

- *Fouriers* : Plots a steady state, time domain graph of node voltage against time for a specified node pair.
- *Circuit_Response* : Determines and plots the circuit response of the circuit. There are five different options :
  - Plotting frequency response on a linear axis.
  - Plotting frequency response on a logarithmic axis.
  - Plotting parametric response on a linear axis when a component value is varied.
  - Plotting parametric response on a logarithmic axis when a component value is varied.
  - Plotting node response on a linear axis when frequency and the component value are varied.

The nodes for which the response is calculated are assumed to be already defined. The next seven subprograms are declared within this method.

- *Get_X_Component* : Gets a range of values which are used when varying a component for plotting parametric graphs.
- *Graph_Set_Up* : Draws the X and Y axis, displays the nodes that are involved, and labels the graph.
- *Generate_Point* : Solves the AC circuit at a particular frequency and then stores up to five points.
- *Plot_Point* : Plots up to five points as well as different symbols at certain intervals.
- *Normal_Plot* : Plots the frequency response on a linear or logarithmic axis.
- *Parametric_Plot* : Plots the parametric response on a linear or logarithmic axis when a component value is varied.
- *Parametric_Linear_Plot3* : Plots the node response on a linear axis when the frequency and a component value are varied.
- *Phasors* : Plots a phasor diagram for the defined node pairs.
- *Polar* : Plots a polar diagram for the defined node pairs.

4.17 Scheduler

Needed Units (Interface)	Needed Units (Body)	Units Using This Unit
	Ada_Extensions Analysers Central_Solver Drawer Graphs Menus Monte Netlists Netlist_Supporter Optimiser Screens Tutorials	

4.17.1 Purpose

It does the scheduling of all the processes in the program. It also instructs the MENUS object which menu options to make selectable or non-selectable.

4.17.2 Methods

- *Save_Present_Circuit* : Saves the present circuit to disk.
- *Valid_Monte_Operations* : Makes monte operations selectable or non-selectable.
- *Valid_Optimisation_Operations* : Makes optimisation operations selectable or non-selectable.
- *Valid_Operations* : Makes certain menu operations selectable or non-selectable, depending on whether there is a circuit or not.
- *Initialize* : Initialises the SCHEDULER object and any other object that needs initialising.
- *Scheduling* : Schedules the different processes and determines which menu options are available and which are not.

4.18 Screen_Objects

Needed Units (Interface)	Needed Units (Body)	Units Using This Unit
Ada_Extensions Dos_Environment Inputter	Unchecked_Deallocation	Help Screens

4.18.1 Purpose

It contains a collection of objects from which the user interface is constructed. (See Chapter 8)

4.19 Screens

Needed Units (Interface)	Needed Units (Body)	Units Using This Unit
Ada_Extensions	Dos_Environment Help Inputter Screen_Objects Text_IO	Analysers Central_Solver Drawer Graphs Monte Netlists_Supporter Netlists.Outer Netlists.Screen_Supporter Optimiser Tutorial

4.19.1 Purpose

It contains low-level user interface objects - the Subscreens. (See Chapter 10)

4.20 Tutorials

Needed Units (Interface)	Needed Units (Body)	Units Using This Unit
	Ada_Extensions Dos_Environment Screens Sequential_IO	Scheduler

4.20.1 Purpose

It displays the tutorial screens on the main screen. (See Chapter 12)

4.20.2 Construction

A generic package is instantiated for the reading of the tutorials image file.

```
package Tutorial_IO is new Sequential_IO (TUTORIAL_ITEM_TYPE);
```

4.20.3 Data Types and Structures

The TUTORIAL_ITEM_TYPE holds the image for the tutorial screen and the index used to locate each image. The image consists of 3404 locations - 23 rows * 74 columns of 1 character and 1 attribute.

```
type ASCII_ARRAY_TYPE is array (1..3404) of ASCII_CHARACTERS;

type TUTORIAL_ITEM_TYPE is
   record
      INDEX    : INTEGER_TYPE;
      IMAGE    : ASCII_ARRAY_TYPE;
   end record;
```

4.20.4 Methods

- *Find_Tutorial* : Searches the disk file for the tutorial item. If the item, or the disk file containing the tutorial images is not found, an appropriate error message appears.
- *Get_Tutorial_Option* : Gets a tutorial option from the user.
- *Initialize_Tutorial_Path* : Gets the initial default path to the Tutorial file.
- *Show* : Displays the tutorial image on the main screen.
- *Tutorial_To_Screen* : Copies the tutorial image from the file to the main screen.

CHAPTER 5

DATAFLOW DIAGRAMS AND PDL

5.1 Introduction

All the major processes of Elector can be represented by dataflow diagrams. Program Description Language (PDL) describes in greater detail each dataflow diagram. The design of Elector can be broken down into three levels.

5.2 The First Level of the Elector Design

The simulation and optimisation processes are specified.

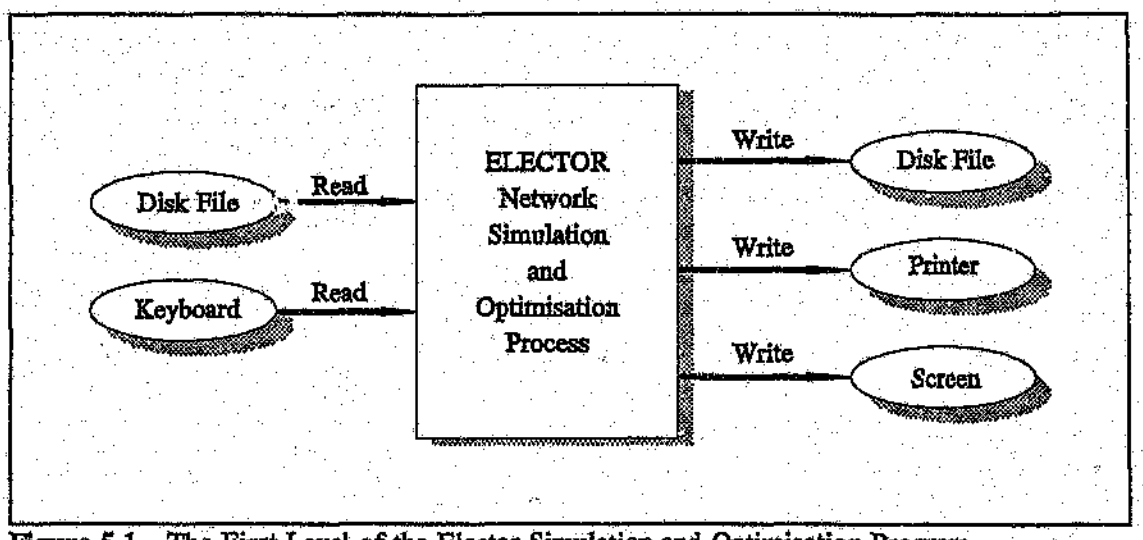

Figure 5-1 The First Level of the Elector Simulation and Optimisation Program

```
begin
  loop
    Get an option from the menu specifying the next operation.
      The option can be selected by mouse or keyboard.
    case (menu option) is
      when (editing) =>
        Execute the edit process.
        if (the printer device is active) then
          Write the text that was written to the screen also to the printer.
        end if;
        if (the disk device is active) then
          Write the text that was written to the screen also to the disk.
        end if;
      when (analysing or optimising) =>
        if (a valid circuit exists) then
          Execute the specified process.
```

```
            if (the printer device is active) then
              Write the text that was written to the screen also to the printer.
            end if;
            if (the disk device is active) then
              Write the text that was written to the screen also to the disk.
            end if;
          else
            Display an appropriate error message.
          end if;
      end case;
    end loop;
end Elector;
```

5.3 The Second Level of the Elector Design

The first level of the design can be broken down into three separate processes.

- The Editing process.
- The Analysis process.
- The Optimisation process.

The Editing Process

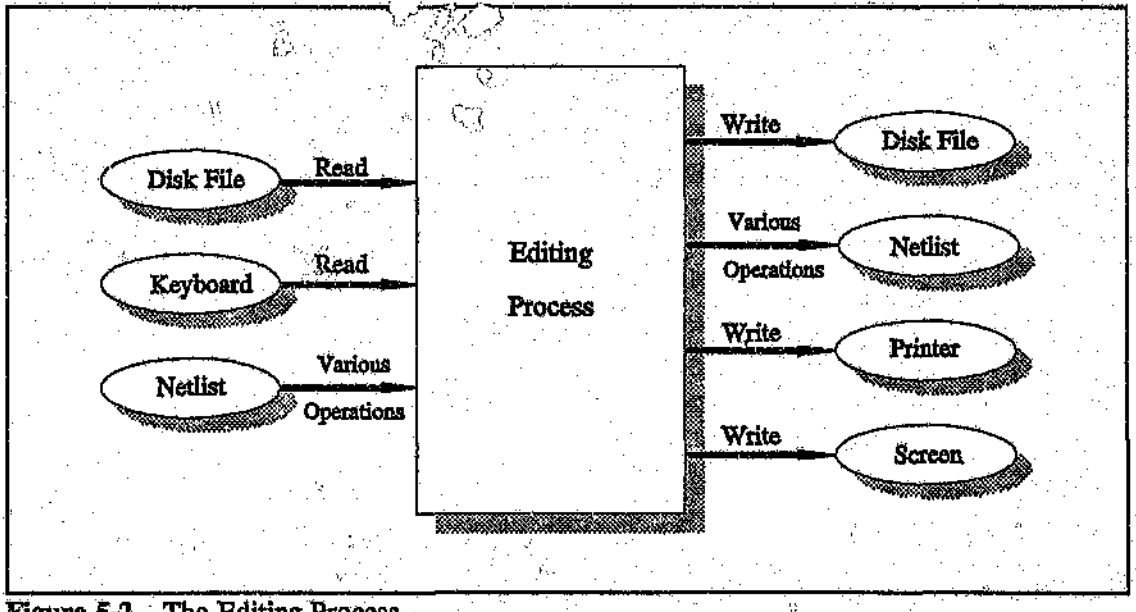

Figure 5-2 The Editing Process

```
begin
  loop
    Get a specific editing option from the menu.
    Perform the desired operations on the netlist.
```

Display the updated netlist components in text and graphical form on the screen.
end loop;
end Editing Process;

The Analysis Process

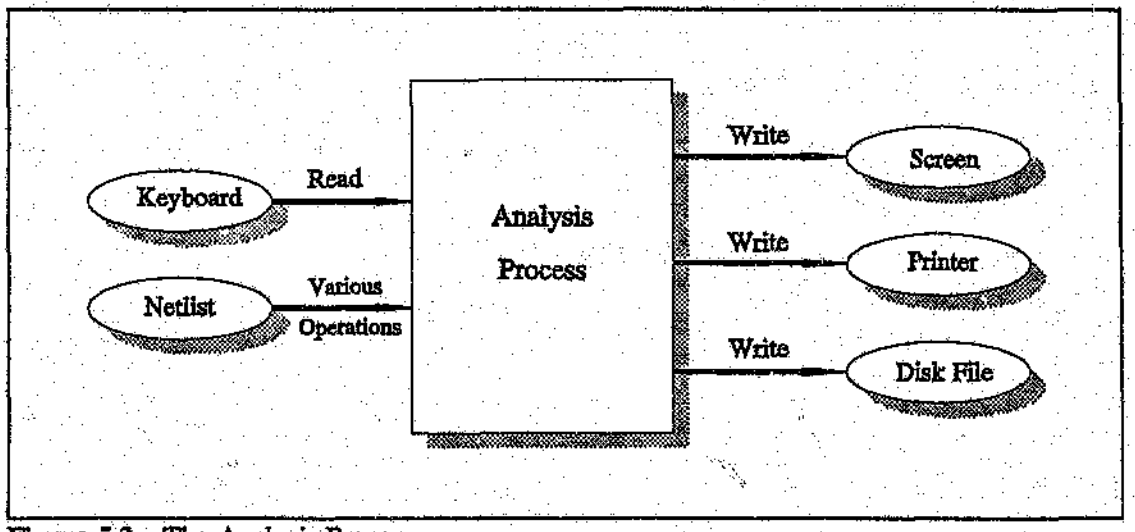

Figure 5-3 The Analysis Process

```
begin
  loop
    Get a specific analysis option from the menu.
    Extract circuit information from the netlist.
    Get the operating frequency from the keyboard.
    Deduce and solve a set of simultaneous linear equations for all node voltages with respect
      to the reference (0) node.
    Find the relative node voltages.
    Display the results in a specified output format.
  end loop;
end Analysis Process;
```

The Optimisation Process

```
begin
  loop
    Get a specific optimisation option from the menu.
    Get the components to be optimised and their standard deviations from the keyboard.
    if (the selected component exists in the netlist) then
      Store the information.
    else
      Display an appropriate error message.
    end if;
    Get the operating frequency from the keyboard.
    Perform the desired type of optimisation.
```

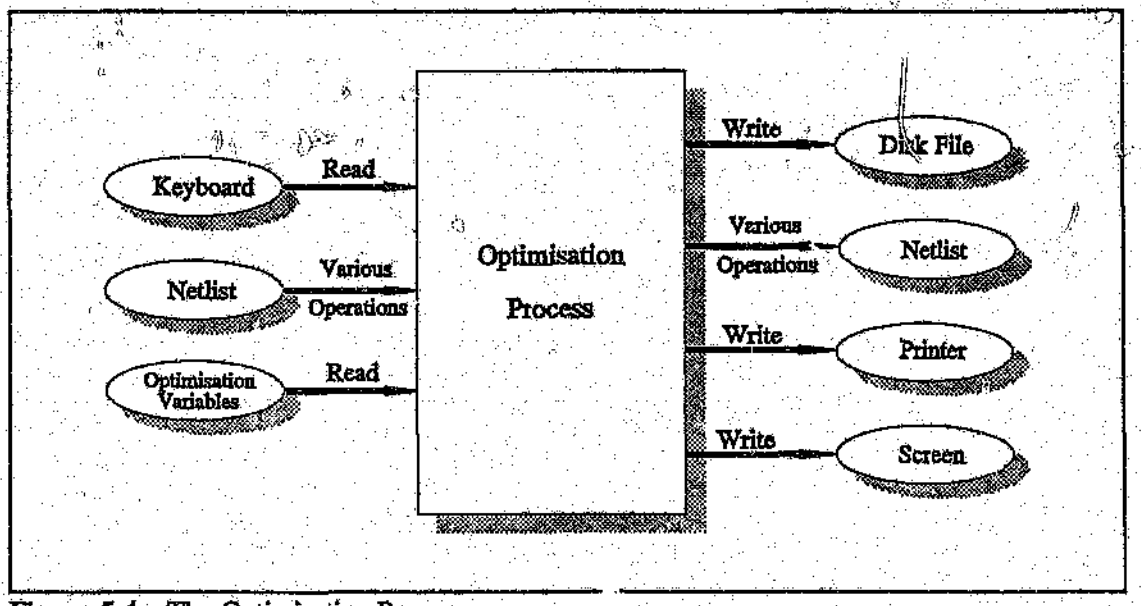

Figure 5-4 The Optimisation Process

```
    if (the optimisation is successful) then
      Update the netlist with the tuned component values.
    else
      Display an appropriate error message.
    end if;
  end loop;
end Optimisation Process;
```

5.4 The Third Level - Editing Processes

The editing process of the second level of the design can be broken down into six separate processes.

- The Change Magnitude and/or Phase process.
- The Insert process.
- The List process.
- The Schematic Drawing process.
- The Show Component process.
- The Show Circuit Branch process.

The Change Magnitude and Phase Process

```
begin
  if (a circuit exists) then
    Get the type of independent source to change and whether the magnitude or the phase is to
```

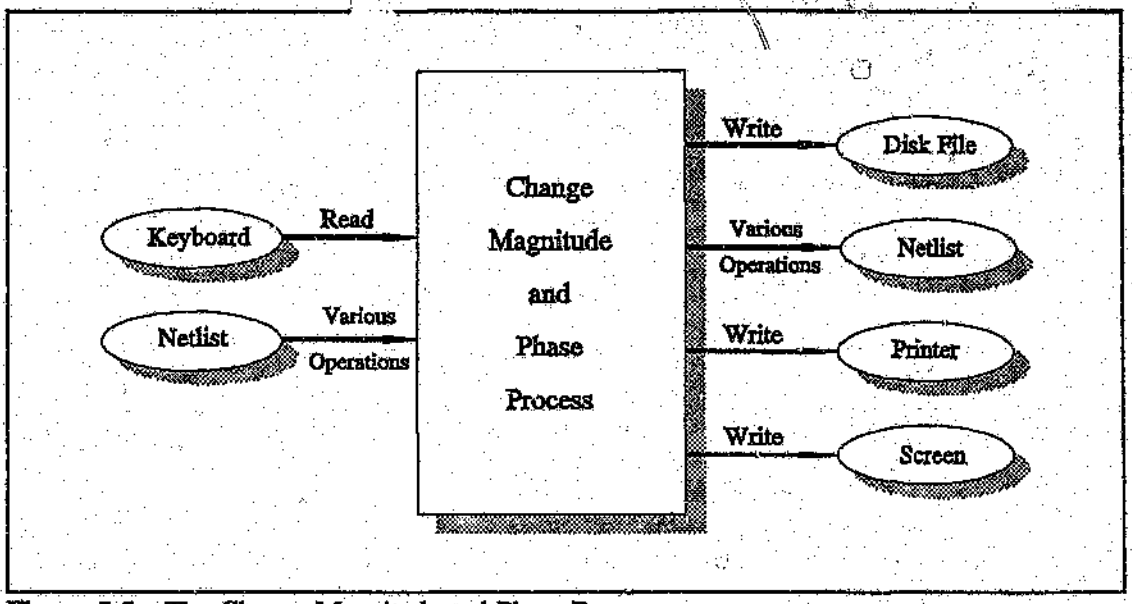

Figure 5-5 The Change Magnitude and Phase Process

```
      be changed.
    Get the node data for this component.
    Determine if this component exists in the netlist.
    if (this component exists) then
      Get the new magnitude or phase.
      Calculate the new real and imaginary parts of the component.
      Update the netlist with these new values.
    else
      Display an appropriate error message.
    end if;
  else
    Display an appropriate error message.
  end if;
end Change Magnitude and/or Phase Process;
```

The Insert Process

```
begin
  loop
    Initialize a component and associated node data.
    Get a component and it's value from the keyboard.
    if (the component value > maximum for that component) then
      Set the value of the component to the maximum allowed.
    elsif (the component value < minimum for that component) then
      Set the value of the component to the minimum allowed.
    end if;
    loop
      Get the first higher and first lower nodes from the keyboard.
      if (the first higher node = the first lower node) then
        Display an appropriate error message.
      else
```

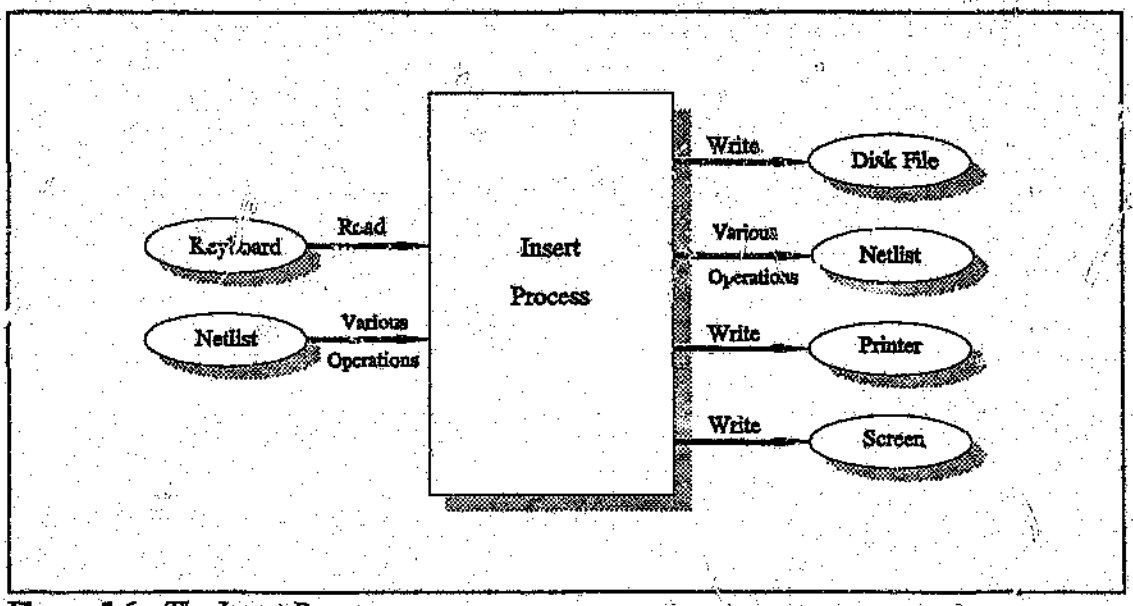

Figure 5-6 The Insert Process

```
      Exit from this loop.
    end if;
  end loop;
  if (coupling or controlling nodes are needed) then
    loop
      Get the second higher and second lower nodes from the keyboard.
      if (the second higher node = the second lower node) then
        Display an appropriate error message.
      else
        Exit from this loop.
      end if;
    end loop;
  end if;
  if (the component is a current or voltage source) then
    loop
      Get the phase from the keyboard.
      if (the phase = 0) then
        Display an appropriate error message.
      else
        Exit from this loop
      end if;
    end loop;
  end if;
  Find out if such a component exists in the circuit.
  if (the component already exists) then
    Update it's value with the new value.
  else
    Find the branch where the new component will be put using it's node data.
    Insert the component values into the netlist.
  end if;
  if (the component value = 0.0 and no other component exists in this branch) then
    Delete this empty branch from the netlist.
  end if;
  Exit from this loop when no more components are to be inserted.
```

```
  end loop;
end Insert Process;
```

The List Process

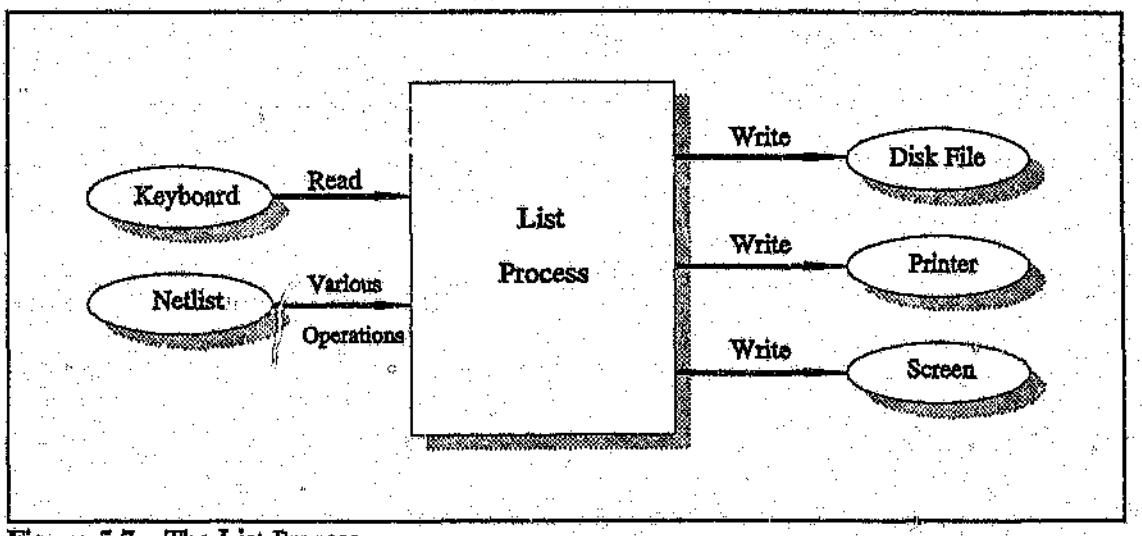

Figure 5-7 The List Process

```
begin
  if (a circuit exists) then
    Get the mode of listing ; Admittance, Equivalent Inductors or RLC.
    Select the first branch of the circuit.
    loop
      Select the first branch component type.
      loop
        if (the component type is compatible with the mode of listing) then
          Display the component.
        end if;
        Select the next component.
        Exit from this loop when all components have been selected.
      end loop;
      Move to the next branch in the circuit.
      Exit from this loop when all the branches have been selected.
    end loop;
  else
    Display an appropriate error message.
  end if;
end List Process;
```

The Schematic Drawing Process

```
begin
  Initialise the drawing variables.
  Get node placement plan or accept the default node placement.
```

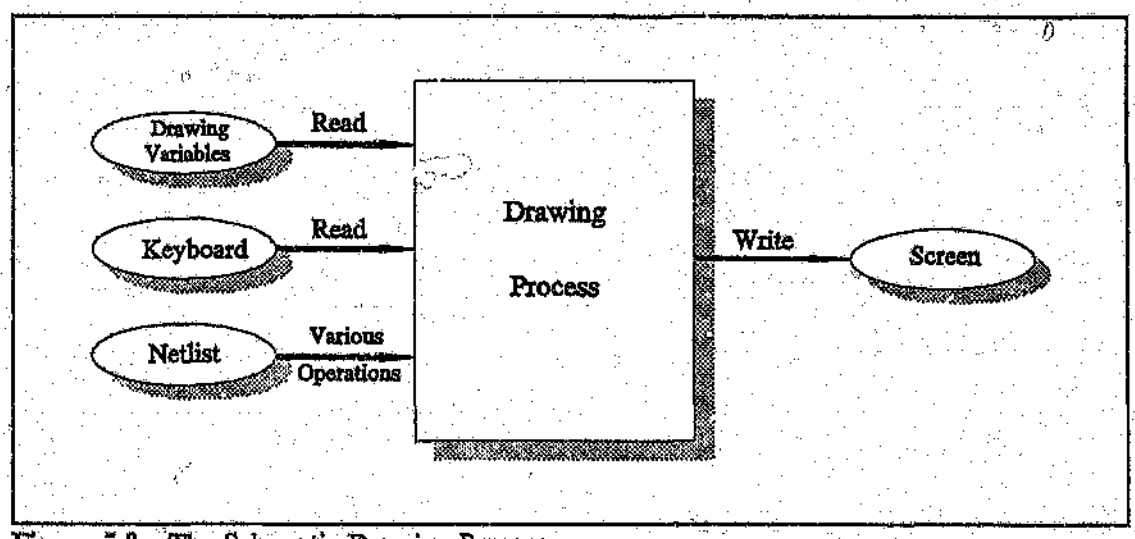

Figure 5-8 The Schematic Drawing Process

```
  Get mode of drawing (fast - inaccurate or slow - accurate).
  Draw the circuit in the default position and size.
  loop
    Get input from the user.
    case (of key pressed) is
      when (Left)        => Decrement horizontal origin.
      when (Right)       => Increment horizontal origin.
      when (Up)          => Decrement vertical origin.
      when (Down)        => Increment vertical origin.
      when (Home)        => Reset the size of the circuit.
      when (End)         => Move to the end of the circuit.
      when (Page up)     => Zoom into the circuit.
      when (Page down)   => Zoom out of the circuit.
      when (Escape)      => Exit from this loop.
    end case;
    if (the circuit must be redrawn) then
      Clear the main screen.
      Draw the circuit on the main screen.
    end if;
  end loop;
end Schematic Drawing Process;
```

The Show Component Process

```
begin
  if (a circuit exists) then
    Get the type of component from the keyboard.
    Select the first branch of the circuit.
    loop
      if (the component type selected exists in this branch) then
        Display the component data.
      end if;
      Move to the next branch in the circuit.
      Exit from this loop if all the branches have been selected.
```

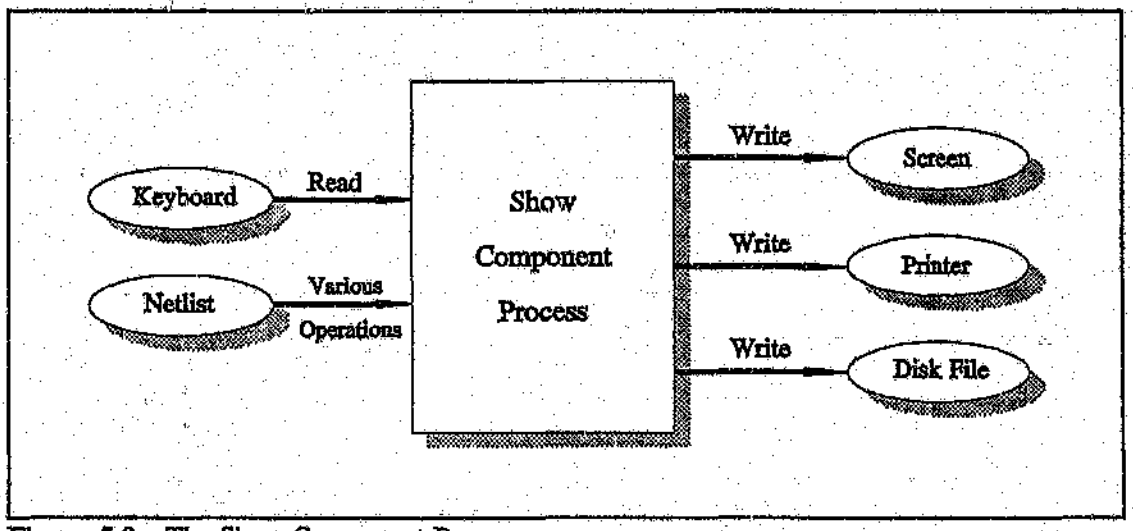

Figure 5-9 The Show Component Process

```
    end loop;
  else
    Display an appropriate error message.
  end if;
end Show Component Process;
```

The Show Circuit Branch Process

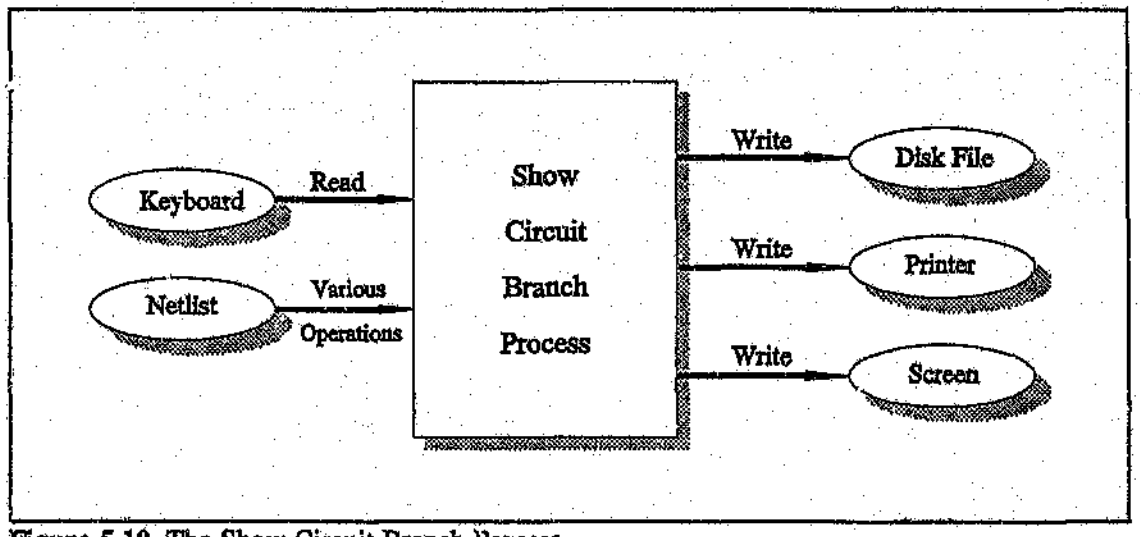

Figure 5-10 The Show Circuit Branch Process

```
begin
  if (a circuit exists) then
    Get a node pair to show from the keyboard.
    if (the node pair is invalid) then
      Display an appropriate error message.
    else
      Select the first branch in the circuit.
```

```
      loop
        if (this branch corresponds to the selected branch) then
          Display all "general" components in this branch.
        end if;
        Moves to the next branch in the circuit.
        Exit from this loop when all the branches have been selected.
      end loop;
    end if;
  else
    Display an appropriate error message.
  end if;
end Show Circuit Branch Process;
```

5.5 The Third Level - Analysis Processes

The analysis process of the second level of the design can be broken down into nine separate processes.

- The Admittance Matrix process.
- The Circuit Response process.
- The Fourier Analysis process.
- The Monte-Carlo Statistical Analysis process.
- The Node Voltage process.
- The Phasor Diagram process.
- The Polar Plotting process.
- The Power Computation process.
- The Sensitivity process.

The Admittance Matrix Process

```
begin
  loop
    Extract circuit information from the netlist.
    Get the operating frequency from the keyboard.
    Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the
      susceptance matric.
    Display each entry of the G and B matrices.
  end loop;
end Admittance Matrix Process;
```

The Circuit Response Process

```
begin
  Get the type of plot required from the keyboard.
```

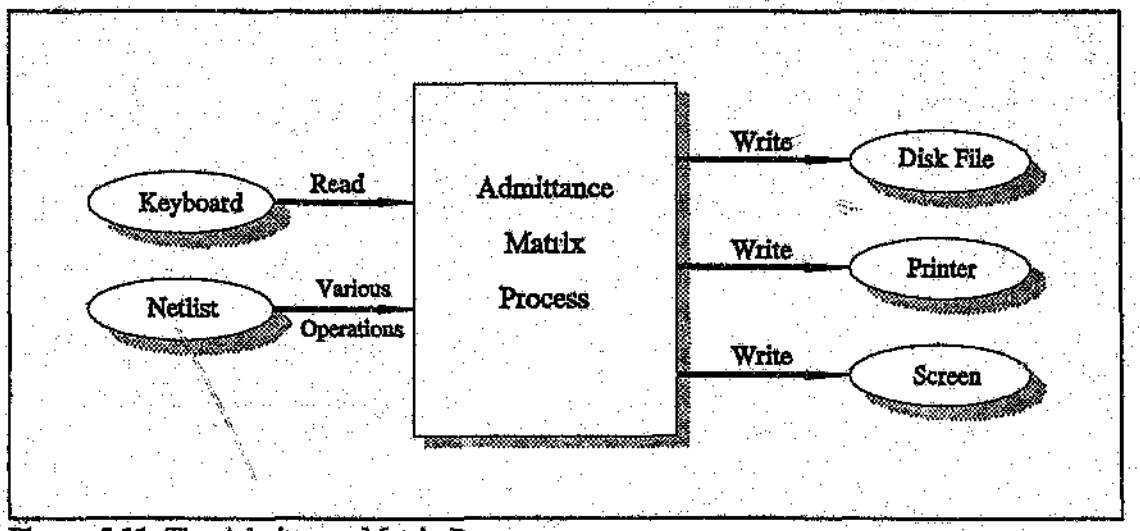

Figure 5-11 The Admittance Matrix Process

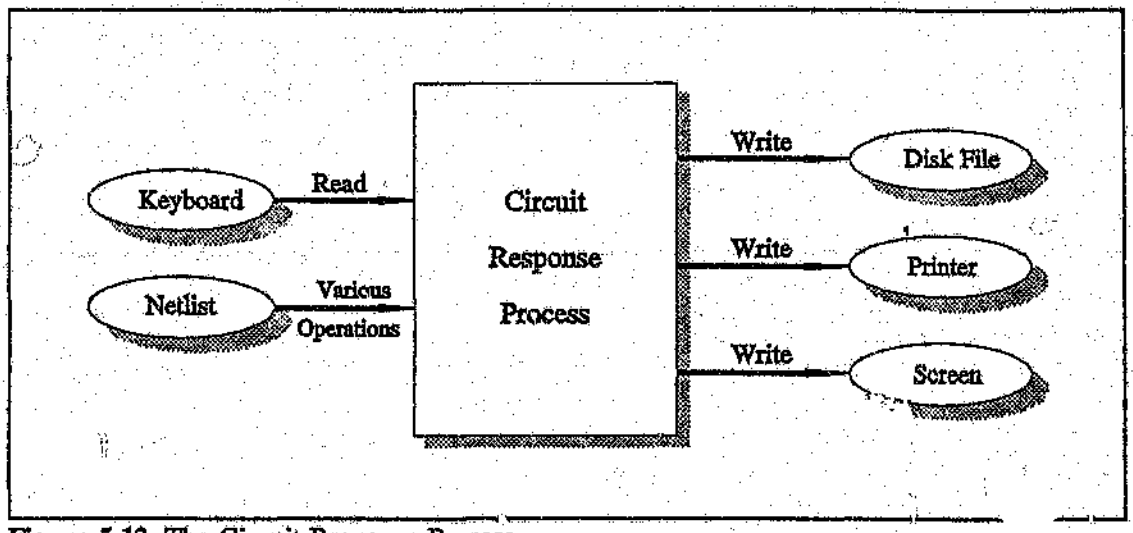

Figure 5-12 The Circuit Response Process

```
case (plot type) is
  when (linear frequency) = >
    Get minimum and maximum frequencies, type of vertical scale, and whether to autoscale
      or not, from the keyboard.
    if (no autoscaling is required) then
      Get the minimum and maximum scale values.
    end if;
    Get the plot increment from the keyboard.
    Select the minimum frequency.
    Label the horizontal frequency axis.
    if (no autoscaling is required) then
      Calculate the scales, and label the vertical axis.
    end if;
    Calculate the frequency increment.
    loop
      Find the relative node voltages at the current frequency and convert it to the selected
        vertical plot type.
      if (no autoscaling is required) then
```

```
        Scale, offset and plot.
      else
        Update the minimum and maximum values.
        Store the results (plot values).
      end if;
      Increment the frequency by the frequency increment.
      Exit from this loop when the frequency > the maximum frequency.
    end loop;
    if (autoscaling is selected) then
      Select the minimum frequency.
      loop
        Scale, offset and plot using the derived minimum and maximum vertical scale values.
        Increment the frequency by the frequency increment.
        Exit from this loop when the frequency > maximum frequency.
      end loop;
    end if;
  when (logarithmic frequency) =>
    Same as for linear frequency except that logarithmic frequency is used.
  when (linear parametric) =>
    The frequency is constant.
    Get the operating frequency and a component.
    if (the component exists in the netlist) then
      Save the component value.
      Plot a graph as above - component on horizontal axis.
      Restore the value of the component to it's original value in the netlist.
    end if;
  when (logarithmic parametric) =>
    Same as for linear parametric except that logarithmic component is used.
  end case;
end Circuit Response Process;
```

The Fourier Analysis Process

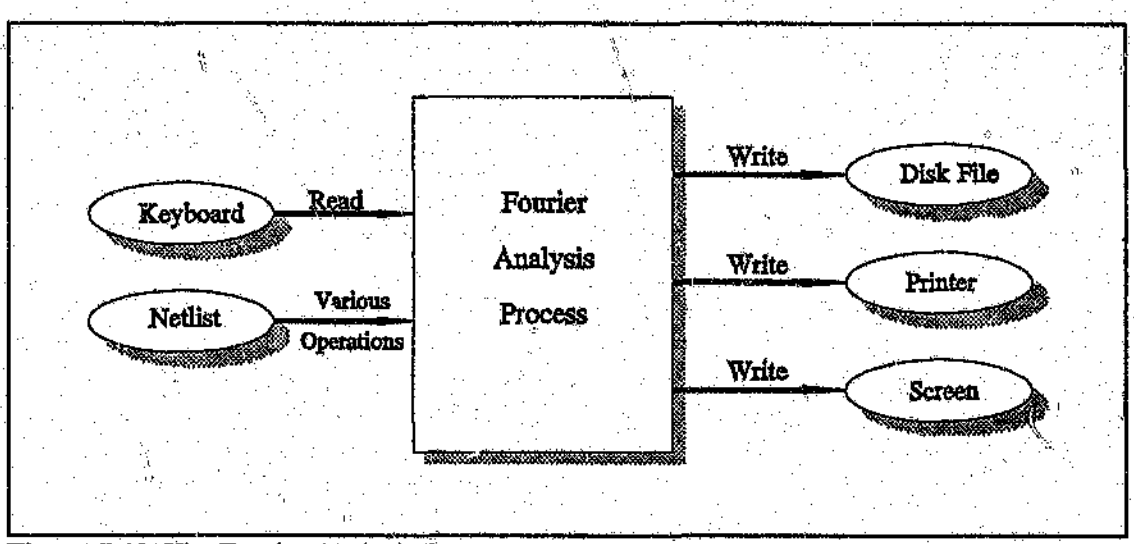

Figure 5-13 The Fourier Analysis Process

```
begin
  loop
    Get the operating frequency from the keyboard.
    Get the shape of the desired time domain waveform from the keyboard.
    Get the number of harmonics this waveform is to be constructed from.
    Compute the magnitude and phase of each spectral component.
    Get the node for output and calculate the response of the circuit to each spectral component.
    Get the time domain window and the sampling increment from the keyboard.
    Draw and label the horizontal axis.
    Select the initial time for the time window.
    Set the maximum magnitude to zero.
    loop
      Find the time domain signal magnitude at the present time.
      Store the magnitude.
      if (this magnitude > maximum magnitude) then
        Set the maximum magnitude = magnitude.
      end if;
      Increment the time by the time step.
      Exit from this loop when the time > end time of the window.
    end loop;
    Label the vertical axis.
    Select the initial time of the time window.
    loop
      Scale, offset and plot the time domain magnitude.
      Increment the time by the time increment.
      Exit from this loop when the time > end time of the window.
    end loop;
  end loop;
end Fourier Analysis Process;
```

The Monte-Carlo Statistical Analysis Process

```
begin
  Get a Monte-Carlo optimisation option from the keyboard.
  case (Monte-Carlo option) is
    when (recover) =>
      Check for a Monte-Carlo file on disk.
      if (the Monte-Carlo file exists) then
        Recover the Monte-Carlo file.
      end if;
    when (save) =>
      if (a Monte-Carlo list exists) then
        Save the Monte-Carlo lists.
      end if;
    when (clear) =>
      if (Monte-Carlo lists exist) then
        Delete each branch of the Monte-Carlo lists.
      end if;
    when (insert) =>
      Get a valid component from the keyboard.
      Find it's position (branch) in the netlist.
      Get the standard deviation and type of distribution (Gaussian / Uniform) from the keyboard.
```

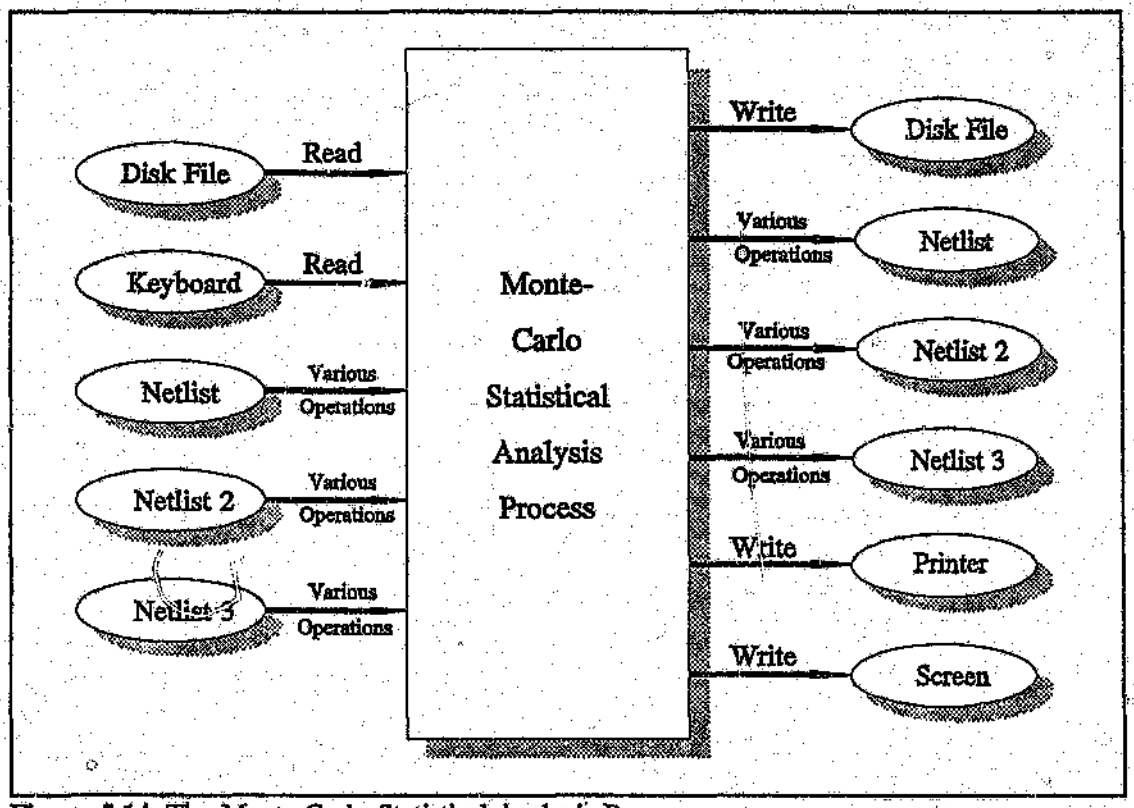

Figure 5-14 The Monte-Carlo Statistical Analysis Process

```
        Store the information in the Monte-Carlo lists. (Same branch position for each netlist).
      when (show) =>
        Scan through both Monte-Carlo lists.
        if (the component exists) then
          Display the component, standard deviation, and the type of distribution, on the screen.
        end if;
      when (perform) =>
        Get the number of simulations, the node pair to be analysed, the seed for the random
          number generator, and whether interim results are required, from the keyboard.
        loop
          Vary all selected components.
          Find the relative node voltages selected.
          Store the relative node voltages.
          Exit from this loop when the number of simulations has been reached.
        end loop;
      when (test random) =>
        Perform a frequency distribution and Chi-squared test on the random number generator,
          displaying the results in numerical and bar graph form.
      when (worst) =>
        Get the number of simulations, the node pair to be analysed, the seed for the random
          number generator, plotting increment, type of output, and frequency range from the
          keyboard.
        Vary components.
        Plot minimum and maximum values.
    end case;
end Monte_Carlo Statistical Analysis Process;
```

The Node Voltage Process

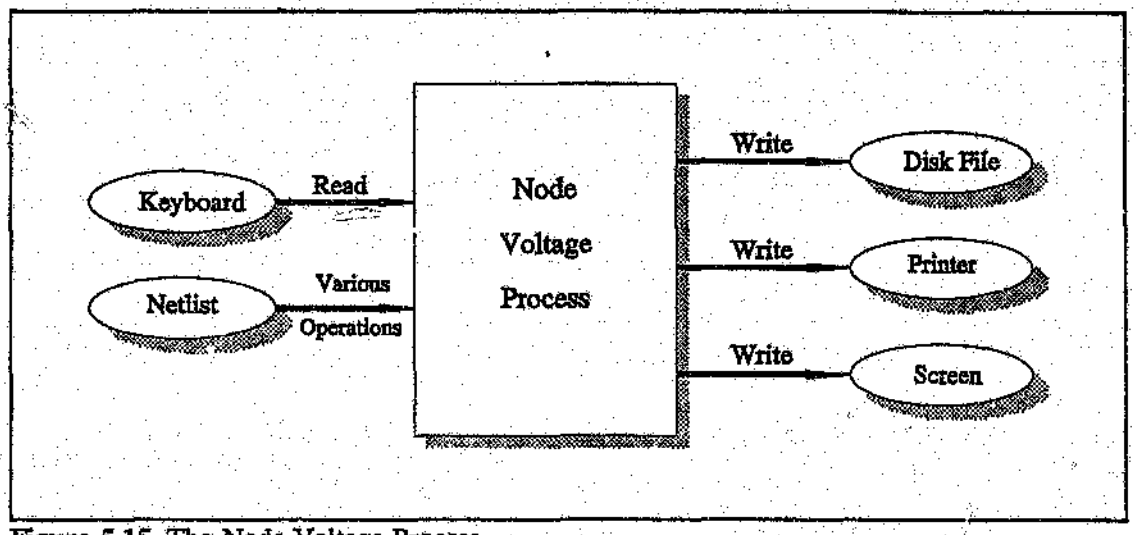

Figure 5-15 The Node Voltage Process

```
begin
  loop
    Extract circuit information from the netlist.
    Get the operating frequency from the keyboard.
    Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the
      susceptance matrix.
    Solve the two sets of simultaneous linear equations (of the form YV = I, (where V is the
      node voltages and I is the branch currents) by equating the real and imaginary parts
      for all node voltages (with respect to the reference node 0).
    Find the relative node voltages.
    Calculate the real and imaginary parts, magnitude, log magnitude and phase of the relative
      node voltages.
    Display the results.
  end loop;
end Node Voltage Process;
```

The Phasor Diagram Process

```
begin
  loop
    Extract circuit information from the netlist.
    Get the operating frequency from the keyboard.
    Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the
      susceptance matrix.
    Solve the two sets of simultaneous linear equations (of the form YV = I, (where V is the
      node voltages and I is the branch currents) by equating the real and imaginary parts
      for all node voltages (with respect to the reference node 0).
    Find the relative node voltages.
```

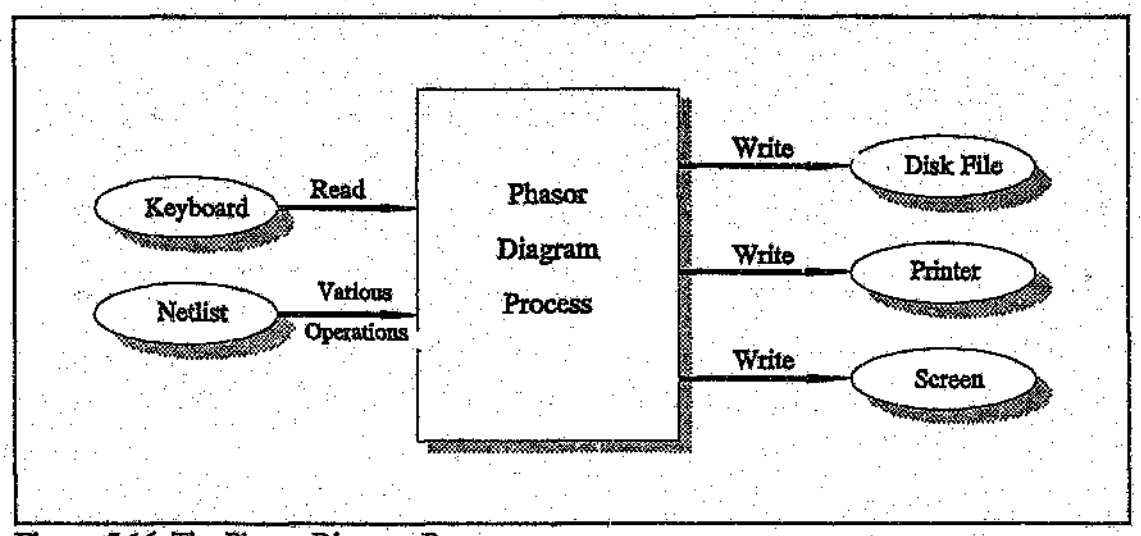

Figure 5-16 The Phasor Diagram Process

Convert the relative node voltages to real-imaginary form.
Plot each phasor on the graphics screen.
end loop;
end Phasor Diagram Process;

The Polar Plotting Process

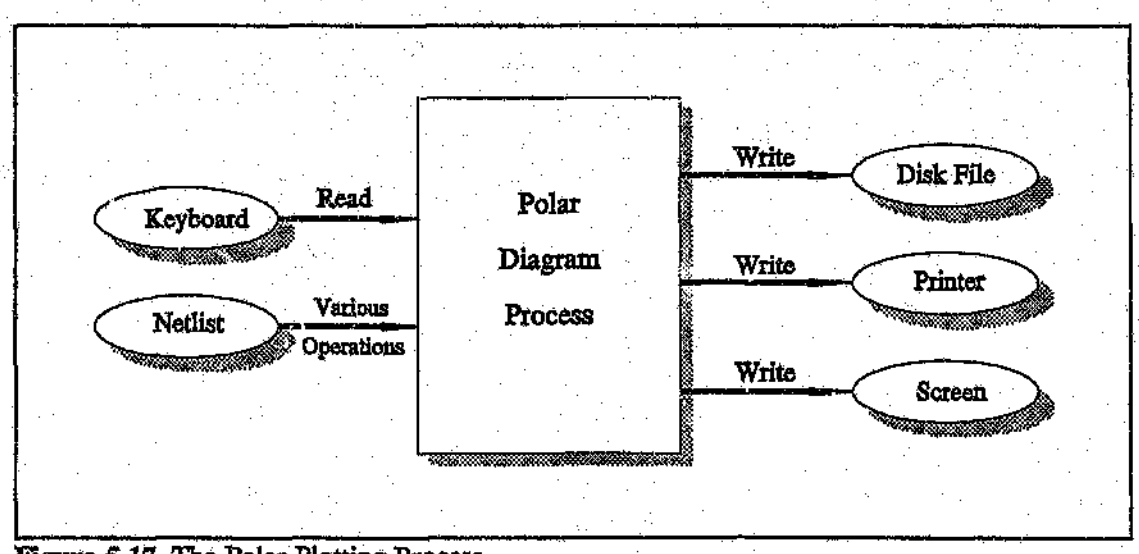

Figure 5-17 The Polar Plotting Process

begin
Extract circuit information from the netlist.
Get the initial, final and incremental operating frequency parameters from the keyboard.
Draw the polar plot axes.
Select the initial operating frequency.
loop

Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the susceptance matrix.
Solve the two sets of simultaneous linear equations (of the form YV = I, (where V is the node voltages and I is the branch currents) by equating the real and imaginary parts for all node voltages (with respect to the reference node 0).
Find the relative node voltages.
Convert the relative node voltages to real-imaginary form and magnitude-phase form.
Store the real-imaginary parts of the relative node voltages.
if (the magnitude > maximum magnitude) then
Set the maximum magnitude = magnitude.
end if;
Increment the operating frequency.
Exit from this loop when the operating frequency > the final operating frequency.
end loop;
loop
Scale and offset the relative node voltages and plot it (a point) on the graphics screen.
Increment the operating frequency.
Exit from this loop when the operating frequency > final operating frequency.
end loop;
end Polar Plotting Process;

The Power Computation Process

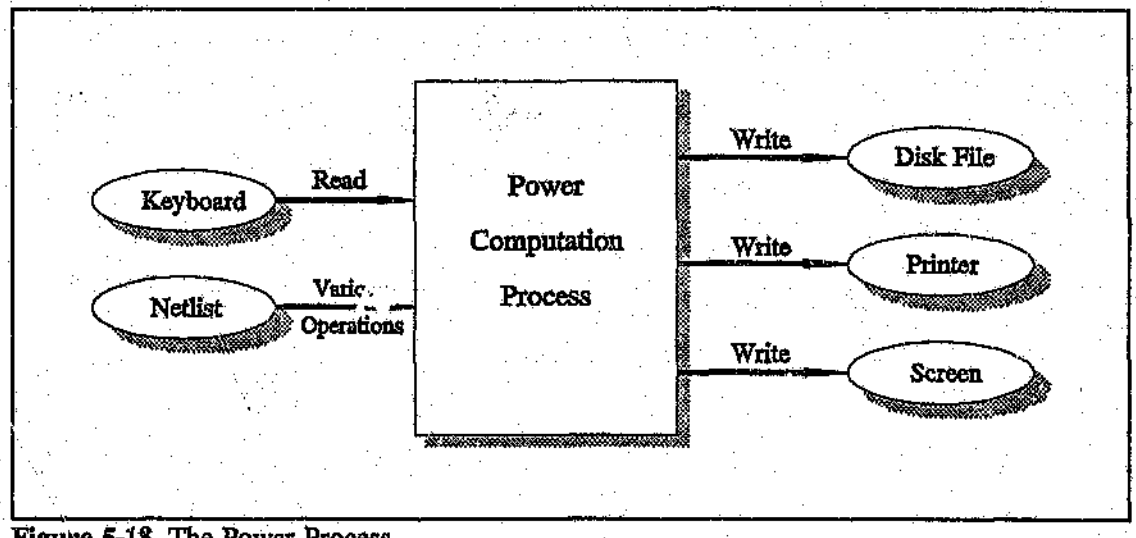

Figure 5-18 The Power Process

begin
loop
Extract circuit information from the netlist.
Get the operating frequency from the keyboard.
Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the susceptance matrix.
Solve the two sets of simultaneous linear equations (of the form YV = I, (where V is the node voltages and I is the branch currents) by equating the real and imaginary parts for all node voltages (with respect to the reference node 0).
Find the relative node voltages.
Calculate the real, reactive (capacitive and inductive) and apparent power absorbed in each

passive component, and calculate the power factor.
Display the results.
end loop;
end Power Computation Process;

The Sensitivity Analysis Process

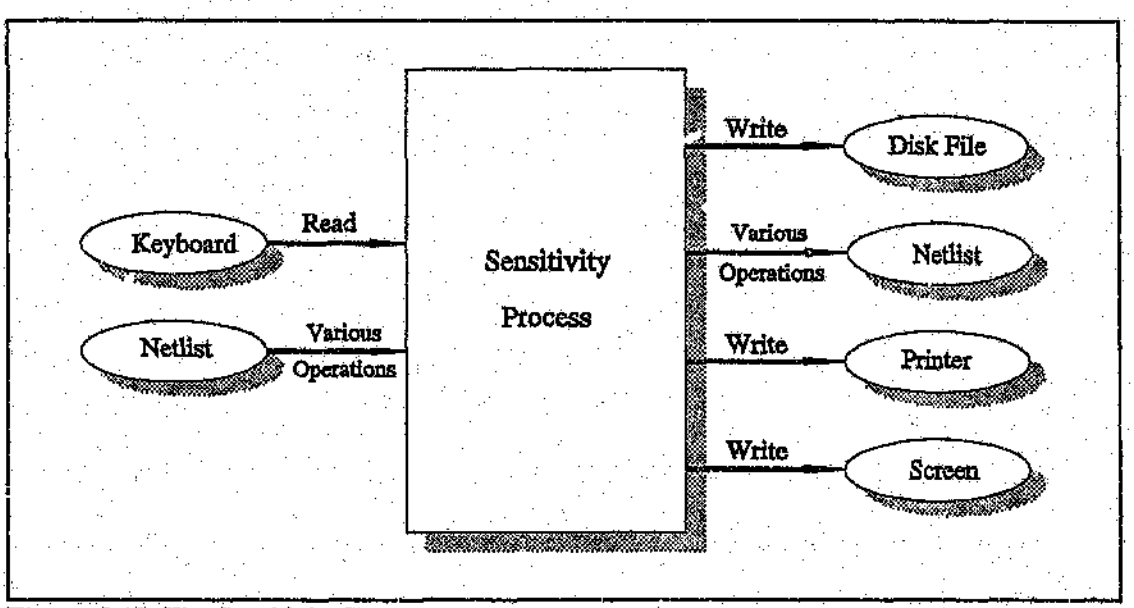

Figure 5-19 The Sensitivity Process

begin
Get the operating frequency from the keyboard.
Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the susceptance matrix.
Solve the two sets of simultaneous linear equations (of the form YV = I, (where V is the node voltages and I is the branch currents) by equating the real and imaginary parts for all node voltages (with respect to the reference node 0)
Find the relative node voltages.
Get the sensitivity perturbation from the keyboard.
Select the first passive component in the network.
Save the component's value.
loop
Vary the component's netlist value by the chosen perturbation.
Deduce the admittance matrix (Y = G + jB), where G is the conductance and B is the susceptance matrix.
Solve the two sets of simultaneous linear equations (of the form YV = I, (where V is the node voltages and I is the branch currents) by equating the real and imaginary parts for all node voltages (with respect to the reference node 0).
Find the relative node voltages.
Calculate the sensitivity factor = (dV/V) / (dcomp/comp).
Display the component that varied and the sensitivity factors.
Restore the original value of the component that varied in the netlist.
Select the next passive component in the netlist.
Exit from this loop when all the netlist components have been varied.
end loop;

end Sensitivity Analysis Process;

5.6 The Third Level - Optimisation Processes

The optimisation process can be broken down into seven processes.

- The Clear process.
- The Find process.
- The Insert process.
- The Maximisation process.
- The Minimisation process.
- The Set process.
- The Show Optimisation process.

The Clear Optimisation Process

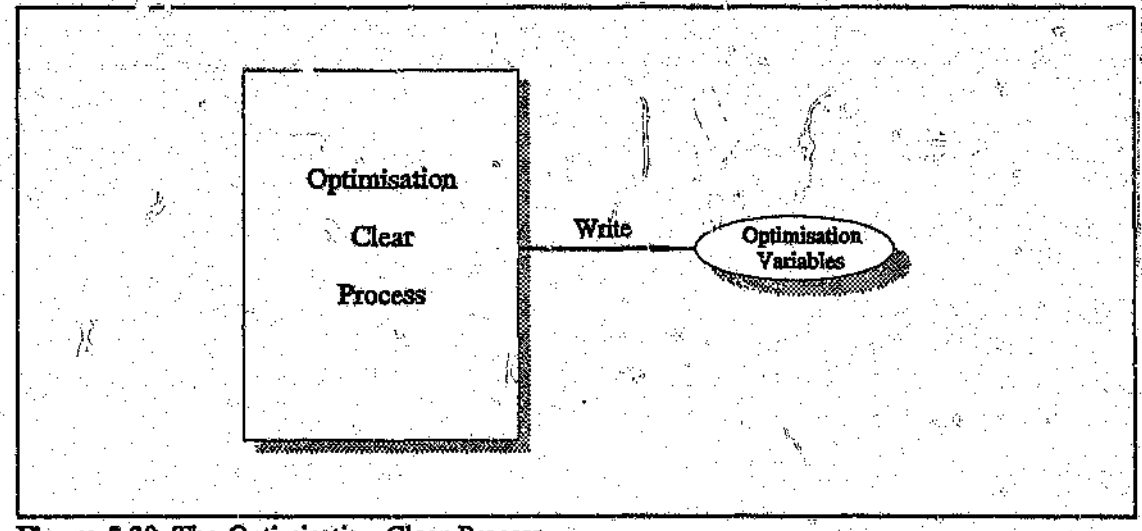

Figure 5-20 The Optimisation Clear Process

```
begin
   Set all optimisation variables to zero.
   Set the number of optimisation variables to zero.
   Reset the frequency selected flag.
end Clear Optimisation Process;
```

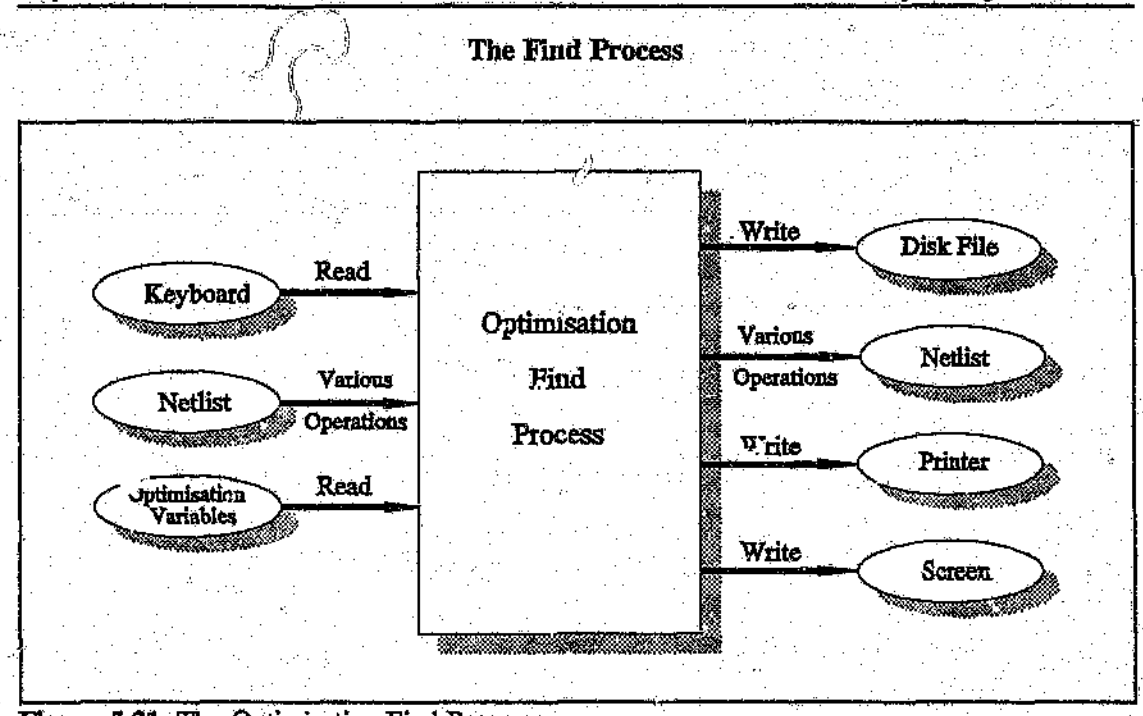

Figure 5-21 The Optimisation Find Process

```
begin
  Get a valid node pair.
  Get the accuracy required.
  Initialize the optimisation variables.
  Save the current operating frequency.
  Select maximisation, and the result type of decibels to find the centre frequency (where the
    response is a maximum).
  loop
    loop
      Set the iteration count to one.
      loop
        Find the relative node voltages and perform multidimensional optimisation using the
          Simplex algorithm, varying the operating frequency.
        if (the relative error is within the desired accuracy) then
          The current optimisation is complete.
          Save the current frequency.
        end if;
        Increment the iteration count.
        Exit from this loop when the circuit is optimised or the iteration count > maximum
          iterations allowed.
      end loop;
      if (the optimisation was not successful) then
        Get response whether to continue or not.
      end if;
      Exit from this loop when the optimisation is successful or no more iterations required.
    end loop;
    if (maximisation was selected) then
      Select to find the lower corner frequency (set the response to 3 dB lower than the
        maximum response by varying the operating frequency).
    end if;
    if (set was selected) then
      Continue with set to find the upper corner frequency.
```

```
    end if;
    Exit from this loop when the centre, upper and lower corner frequencies have been found.
  end loop;
  Calculate the bandwidth and the Q factor.
  Display the centre frequency, the upper and lower corner frequencies, the Q factor, the
    bandwidth, and the midband gain, on the screen.
end Find Process;
```

The Insert Optimisation Process

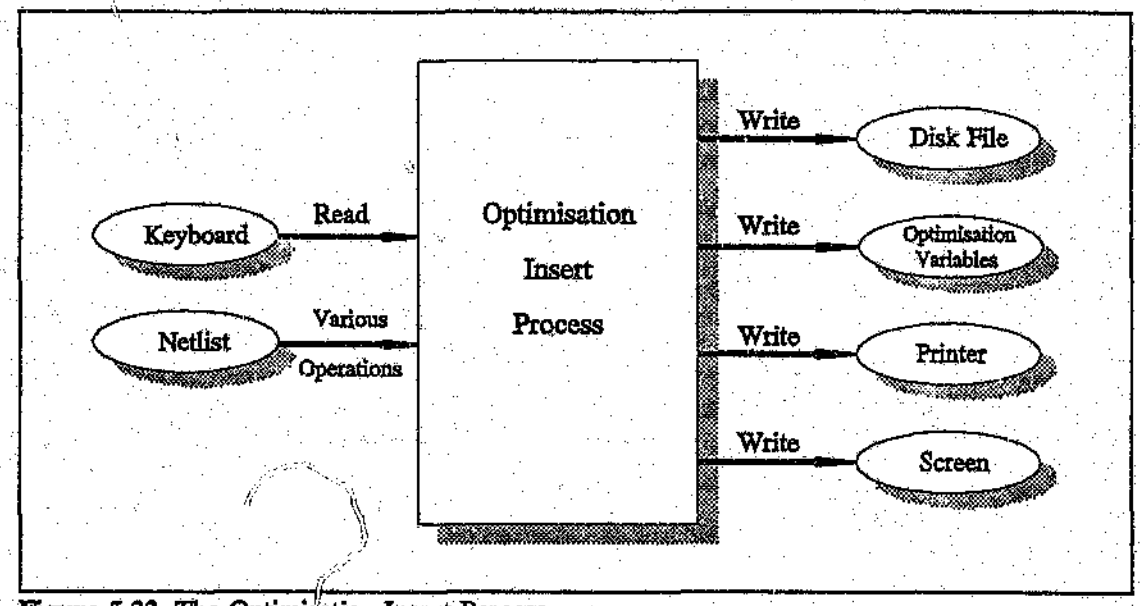

Figure 5-22 The Optimisation Insert Process

```
begin
  if (the maximum number of tuned variables already exists) then
    Display an appropriate error message.
  else
    Get the type of optimisation variable.
    case (optimisation variable type) is
      when (magnitude | phase) =>
        Get the component type and the node data.
        Get whether magnitude or phase option.
        if (the component exists in the netlist) then
          Store the information.
          Display the component.
        else
          Display an appropriate error message.
        end if;
      when (component) =>
        Get the component type and the node pair.
        if (the component exists in the netlist) then
          Store the information.
          Display the component.
        else
          Display an appropriate error message.
```

```
        end if;
      when (frequency) =>
        if (the frequency is already selected) then
          Display an appropriate error message.
        else
          Get the starting frequency.
          Store and display this frequency.
        end if;
    end case;
  end if;
end Insert Optimisation Process;
```

The Maximisation, Minimisation and Set Process

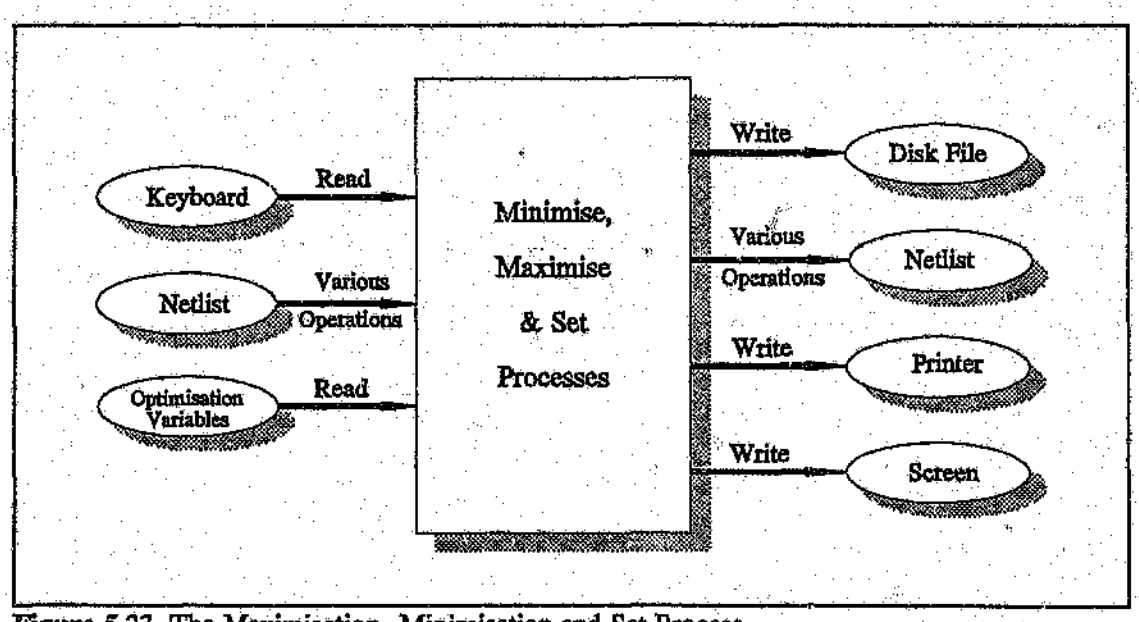

Figure 5-23 The Maximisation, Minimisation and Set Process

```
begin
  if (no tune variables exist) then
    Display an appropriate error message.
  else
    Get a valid node pair from the keyboard.
    Get the accuracy required.
    Get the result type (real, imaginary, magnitude, phase or decibels).
    Initialise for optimisation.
    loop
      Set the iteration count to one.
      loop
        Find the relative node voltages and perform multidimensional optimisation using the
          Simplex algorithm, varying the selected components.
        Update the netlist with the new component values.
        if (the relative error is within the desired accuracy) then
          The current optimisation is complete.
        end if;
        Increment the iteration count.
```

```
        Exit from this loop when the circuit is optimised or the iteration count > maximum
          iterations allowed.
      end loop;
      if (the optimisation was not successful) then
        Get response whether to continue or not.
      end if;
      Exit when the optimisation is successful or no more iterations are required.
    end loop;
    Display the tuned components new values.
  end if;
end Maximisation, Minimisation and Set Process;
```

The Show Optimisation Variables Process

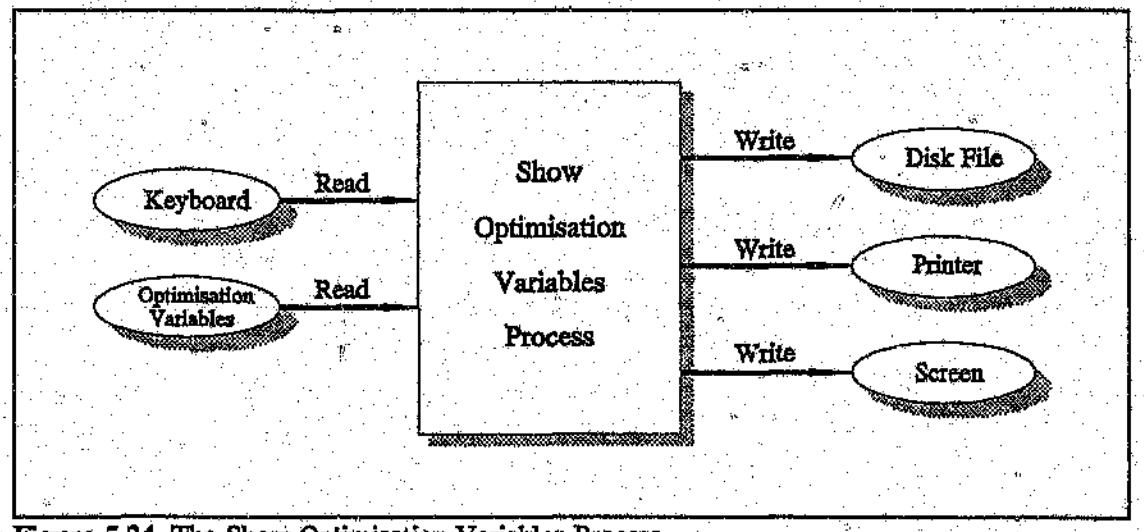

Figure 5-24 The Show Optimisation Variables Process

```
begin
  if (the optimisation variables exist) then
    Select the first optimisation variable.
    loop
      Display the optimisation variable which may be a component value, a magnitude, or a
        phase of an independent source, and/or the operating frequency.
      Select the next optimisation variable.
      Exit from this loop when all the optimisation variables have been displayed.
    end loop;
  end if;
end Show Optimisation Variables Process;
```

CHAPTER 6

THE NETLISTS ABSTRACT DATA TYPE

6.1 Introduction

The Netlists is the core application object of Elector. It holds all the information about all the components in the circuit. The Netlists package contains the NETLIST_TYPE data declarations and three smaller packages. These packages contain methods which operate on instantiations of the NETLIST_TYPE. These three smaller packages are :

- *Screen_Supporter* : It interfaces the Netlists package to the user interface. It is in the Netlists package, but makes use of the user interface features.
- *Inner* : This package contains all the methods that only deal with the netlist object.
- *Outer* : This package contains all the methods that deal with NETLIST_TYPE objects and the screen object. The LOAD_SAVE_RECORD type is used for saving and restoring the NETLIST_TYPE objects. The SEQUENTIAL_IO generic package is also instantiated.

```
type LOAD_SAVE_RECORD (LOAD : BOOLEAN := TRUE) is
   record
      case LOAD is
         when TRUE =>
            TITLE       : STRING (1..64);
            FREQUENCY   : REAL_TYPE;
         when FALSE =>
            COMPS       : COMPONENTS;
      end case;
   end record;

package Netlist_IO is new Sequential_IO (LOAD_SAVE_RECORD);
```

6.2 Behaviour

6.2.1 Introduction

The NETLIST_TYPE is a large dynamic data type that is used whenever a component value is needed or used in a calculation. The NETLIST_TYPE's backbone is a single linked list. Because of it's dynamic nature, it can grow as large as the computer's memory will allow. This means that it must check if there is memory for it to grow larger. If there is no memory, it will raise a STORAGE_ERROR exception.

6.2.2 Exceptions

Exception raised during abnormal circuit Simulation	Raising Subprogram
Attempted_Reference_Node_Removal : There is an attempt to delete the reference node (node 0).	Remove_Node
Branch_Out_Of_Range : There is an attempt to access a component value in a branch that does not exist. ie. The LIST pointer in the NETLIST_TYPE base structure is pointing to null.	Empty_Branch Read_Netlist Update_Mutuals_and_Coupling Write_Netlist
Cannot_Convert_With_Mutual_Inductances : There are mutual inductances in the circuit which would affect the conversion of netlist components.	Convert_Admittance_To_RLC
Delete_Node_Does_Not_Exist : The node to be deleted does not exist in the circuit.	Remove_Node
Meaningless_Component : The actual inductors used in the calculation of mutual inductance cannot be found.	Calculate_Mutual_Inductance Update_Mutuals_and_Coupling
Meaningless_Component_Connection : The high and low node numbers point to the same node.	Calculate_Mutual_Inductance Find_Branch Upda..._Mutuals_and_Coupling
New_Node_Already_Exists : The new node value that will renumber the old node value already exists and so it cannot be used.	Renumber_Nodes
No_Circuit : There is no circuit and so analysis cannot be done.	Check_Topology First_Branch Renumber_Nodes Remove_Node Save_Netlist Show_Node
No_More_Branches : The list of existing branches has come to an end and there are no more.	Next_Branch
No_Vccs_Controlling_Node : There is no defined controlling node pair for an existing voltage controlled current source.	Check_Topology
Node_Pair_Does_Not_Exist : The specified node pair does not exist in the circuit.	Find_Branch Show_Node
Old_Node_Does_Not_Exist : The node to be renumbered in the circuit does not exist.	Renumber_Nodes
Reference_Node_Missing : The circuit is floating and is not tied down to the reference node by a branch.	Check_Topology
Specific_Node_Missing : A specific node is missing from the circuit. This happens when the nodes have not been numbered consecutively and so there is a gap in the numbering.	Check_Topology

Exception raised during abnormal circuit simulation	Raising Subprogram
Storage_Error : This occurs when there is not enough memory to do analysis or to execute the subprogram. (See Chapter 2)	BR_Inductance_Matrix Convert_Admittance_To_RLC Find_Branch Find_Dependent_Source Find_Mutuals_and_Coupling
Too_Many_Branches : There are too many branches in the circuit.	BR_Inductance_Matrix Convert_Admittance_To_RLC

Table 6-1 Exceptions raised by the Netlists Package

6.3 Components

6.3.1 Circuit Components

Internal Representation	Circuit Components
CAPS	Capacitors
INDUCT	Equivalent Inductors
GR_ADMIT	General Real Admittance
GI_ADMIT	General Imaginary Admittance
REAL_VOLT	Real Voltage
IMAG_VOLT	Imaginary Voltage
REAL_CRNT	Real Current
IMAG_CRNT	Imaginary Current
ACT_INDCT	Actual Inductor
CR_ADMIT	Control Real Admittance
CI_ADMIT	Control Imaginary Admittance
CM_ADMIT	Control Magnitude Admittance
POLE_FREQ	Pole Frequency
MUT_INDCT	Mutual Inductor
CUPLING	Coupling Factor

Table 6-2 Circuit Components and Elector's Internal Representation

The components that are used in circuits which can be entered into Elector are defined in Table 6.2. The component's internal representation (used in the Elector Ada source code) is also listed. The COMPONENT_UNIT_TYPE and *Set_Component_Values* method are used to set the units, the display name, minimum and maximum values for each component type.

```
type COMPONENT_UNIT_TYPE is
   record
      UNITS       : STRING (1..4);
      DISPLAY     : STRING (1..4);
      MIN,
      MAX         : REAL_TYPE;
   end record;

type COMPONENT_LIST_TYPE is array (CAPS..RESISTOR) of
     COMPONENT_UNIT_TYPE;
```

6.3.2 Linked List Structure

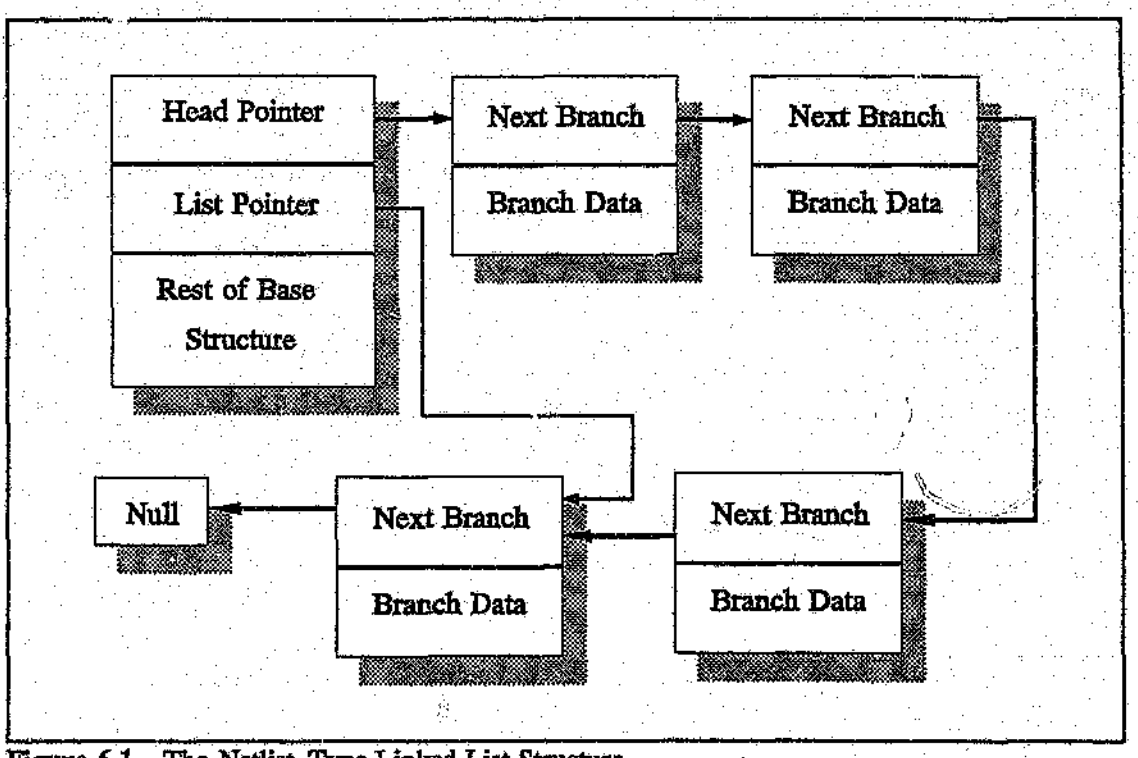

Figure 6-1 The Netlist_Type Linked List Structure

```
type NETLIST;
type LINK is access NETLIST;

type NETLIST is
   record
      NEXT : LINK;
      DATA : COMPONENTS;
   end record;
```

6.3.3 Base Structure

- *HEAD* : Points to the head of the list which corresponds to the first branch in the NETLIST_TYPE. When there is no circuit, the pointer points to NULL.
- *LIST* : Points to the current branch in the netlist.
- *LOCATION* : The values of the pointer to the current branch in the NETLIST_TYPE. This is needed in the optimication process.
- *NODE_MAXIMUM* : The value of the highest node in the netlist.
- *FREQUENCY* : Default frequency.
- *TITLE* : The title of the present circuit which can be a maximum length of 64 characters.

```
type NETLIST_ARRAY_TYPE is array (1..MAX_OPTIMUM_VARIABLE) of LINK;

type NETLIST_TYPE is
   record
      LIST           : LINK;
      HEAD           : LINK;
      LOCATION       : NETLIST_ARRAY_TYPE;
      NODE_MAXIMUM   : NODE_TYPE        := 0;
      FREQUENCY      : REAL_TYPE        := 1000.0;
      TITLE          : STRING (1..64)   := (1..64 => ' ');
   end record;
```

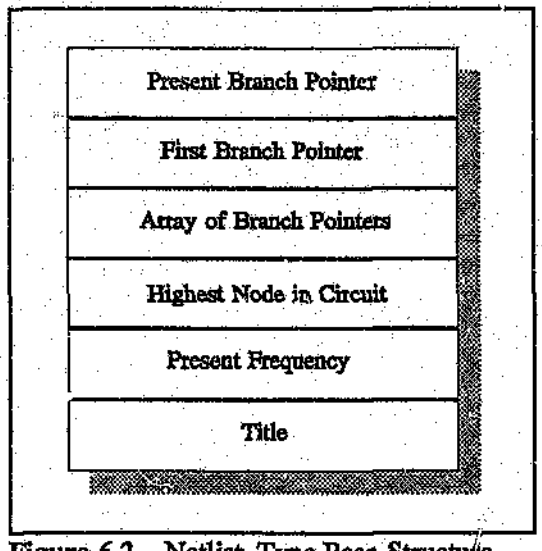

Figure 6-2 Netlist_Type Base Structure

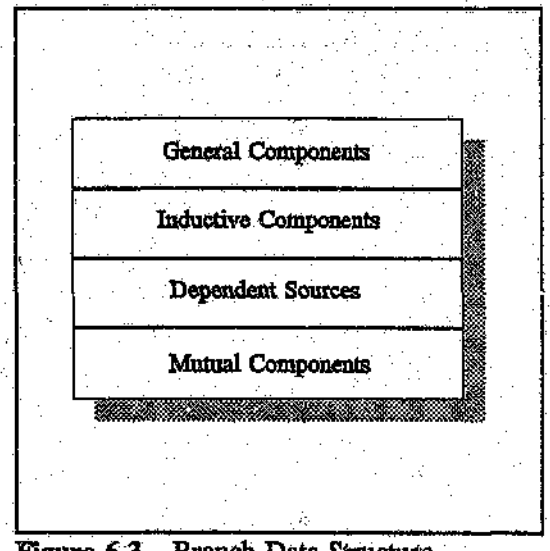

Figure 6-3 Branch Data Structure

6.3.4 Branch Data Structure

```
type COMPONENTS is
   record
      GENERAL        : GENERALS;
      INDUCTIVE      : INDUCTORS;
      DEPEND_SOURCE  : VOLTAGE_CONTROL;
      MUTUAL_INDUCT  : MUTUAL_INDUCTORS;
   end record;
```

6.3.5 General Component Branch Structure

```
type GENERALS is
   record
      CAPACITOR,
      EQUIV_INDUCTOR,
      REAL_ADMITTANCE,
      IMAG_ADMITTANCE,
      REAL_VOLTAGE,
      IMAG_VOLTAGE,
      REAL_CURRENT,
      IMAG_CURRENT    : REAL_TYPE := 0.0;
      HIGHER_NODE,
      LOWER_NODE      : NODE_TYPE := 0;
   end record;
```

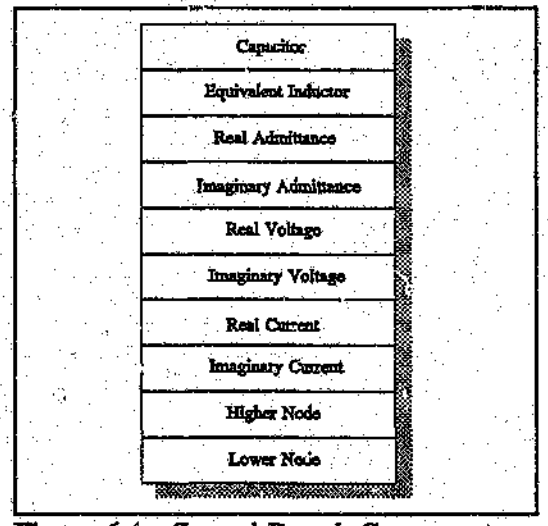

Figure 6-4 General Branch Components

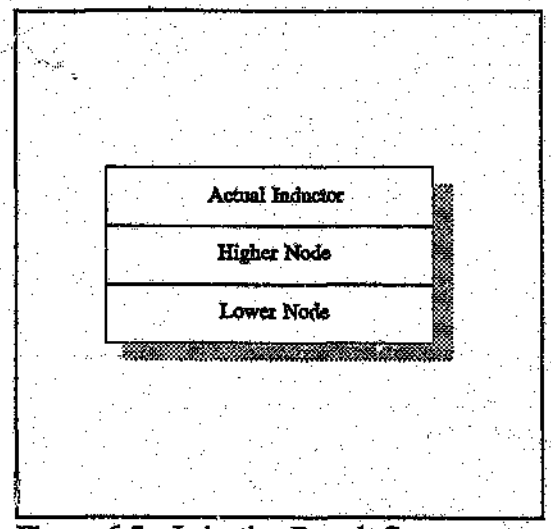

Figure 6-5 Inductive Branch Components

6.3.6 Inductive Component Branch Structure

```
type INDUCTORS is
   record
      ACTUAL_INDUCTOR : REAL_TYPE := 0.0;
      HIGHER_NODE,
      LOWER_NODE      : NODE_TYPE := 0;
   end record;
```

6.3.7 Mutual Inductive Component Branch Structure

```
type MUTUAL_INDUCTORS is
   record
      MUTUAL_INDUCTOR,
      COUPLING        : REAL_TYPE := 0.0;
      HIGHER_NODE1,
      LOWER_NODE1,
      HIGHER_NODE2,
      LOWER_NODE2     : NODE_TYPE := 0;
   end record;
```

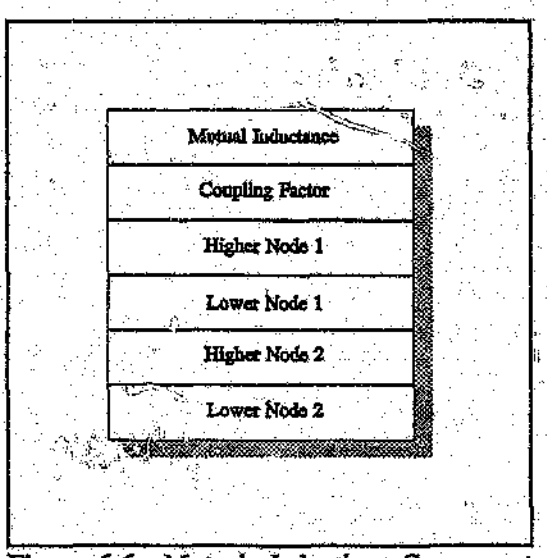

Figure 6-6 Mutual Inductive Component Branch Structure

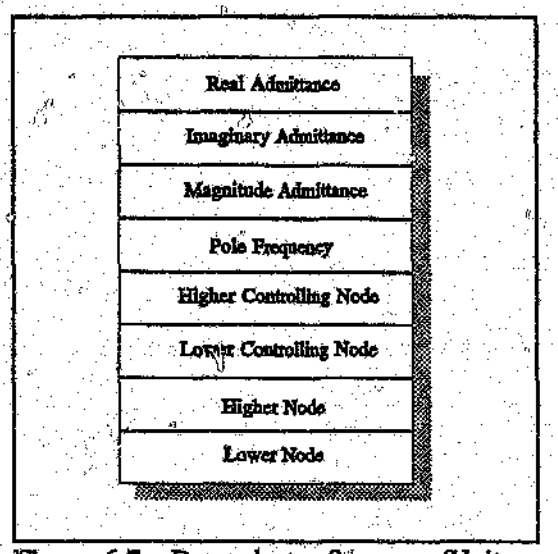

Figure 6-7 Dependent Source (Voltage Controlled Current Source)

6.3.8 Dependent Source Component Branch Structure

```
type VOLTAGE_CONTROL is
   record
      REAL_ADMITTANCE,
      IMAG_ADMITTANCE,
      MAG_ADMITTANCE,
      POLE_FREQUENCY          : REAL_TYPE := 0.0;
      HIGHER_CONTROL_NODE,
      LOWER_CONTROL_NODE,
      HIGHER_NODE,
      LOWER_NODE              : NODE_TYPE := 0;
   end record;
```

6.4 Methods

6.4.1 Services Provided

The NETLIST_TYPE is an abstract data type (ADT). Therefore of this, any access to the data structure of an object (instantiation of the netlist ADT) can only be done by using a method of this object. There are many different methods because other objects that use this object need many different services from it. Methods are grouped according to services they provide in Table 6.3.

Table 6-3 Services provided by the Netlists Package

Type of Service	Method
Accessing the Netlist Base	Read_Frequency Set_Circuit_Title Write_Frequency
Accessing the Current Branch	Clear_Branch Empty_Branch Read_Netlist Write_Netlist
Branch Creation and Insertion	Find_Branch Find_Dependent_Source Find_Mutuals_and_Coupling New_Branch
Checking the Netlist Integrity	Check_Topology
Conversion of the Netlist	Convert_Admittance_To_RLC Convert_RLC_To_Admittance Renumber_Nodes
Deletion of a Branch or Node	Delete_Branch Remove_Node
Displaying Component Values	List_Branch Show_Node
Extracting and Calculating Properties of the Netlist	BR_Inductance_Matrix Calculate_Mutual_Inductance Count_Components Find_Mutuals_and_Coupling Find_Node_Maximum Update_Mutuals_and_Coupling
Initialization of the Netlist	Initialize
Location of a Branch	Find_Branch Find_Dependent_Source Find_Mutuals_and_Coupling
Movement Through the Netlist	First_Branch Next_Branch
Optimisation Methods	Clear_Locations Equal_Locations Retrieve_Location Save_Location
Saving and Restoring the Netlist	Load_Netlist Load_Monte_Netlist Save_Netlist Save_Monte_Netlist
User Interface Communication	Display_Component Get_Component

6.4.2 Screen_Supporter Methods

Display_Component

Displays the component symbol and data on the main screen.

```
begin
  Display the component and the nodes to which it is connected.
  Display it's value and it's units,
  if (the component in CR_ADMIT..CUPLING) then
    if (the component in CR_ADMIT..POLE_FREQ) then
      Display the controlling nodes of the component.
    else
      Display the coupled nodes of the component.
    end if;
  end if;
  if (the component in CRNT_SOURCE..VOLT_SOURCE) then
    Display the phase of the component.
  end if;
end Display_Component;
```

Get_Component

Displays a component list on the PROMPT subscreen and gets the choice of component from the user.

```
begin
  Display a Prompts subscreen and get the chosen component from the user.
  for (COUNTER in the length of the list) loop
    if (this option on the list was chosen) then
      case (of the option chosen) is
        Get the corresponding component.
      end case;
      Exit from this loop.
    end if;
  end loop;
end Get_Component;
```

6.4.3 Inner Methods

BR_Inductance_Matrix

Finds the branch and reciprocal inductance matrices. A branch inductance matrix is generated

from the data in the netlist. It is converted to an indefinite reciprocal inductance matrix from which are extracted the equivalent inductors to be used in the nodal analysis solution. For further information, see : L.N.M. Edward, "BIMRIM : A Mutual Inductance Algorithm for use with Nodal Analysis Programs", International Journal of Electrical Engineering Education, Vol 20, No 2, p 341 - 350, February 1983

```
subtype VECTOR1 is POSITIVE range 1..MAX_BRANCHES;
subtype VECTOR2 is POSITIVE range 1..(MAX_NODES + 1);
subtype VECTOR3 is POSITIVE range 1..DOUBLE_BRANCHES;

type EXPAND_TYPE          is array (VECTOR2, VECTOR1) of REAL_TYPE;
type RIM_TYPE             is array (VECTOR2, VECTOR2) of REAL_TYPE;
type WORK_TYPE            is array (VECTOR1, VECTOR3) of REAL_TYPE;
type NODE_CONNECT_TYPE    is array (VECTOR1) of NODE_TYPE;
type NODE_LIST_TYPE       is array (VECTOR2) of NODE_TYPE;

type MATRIX_TYPE is
  record
    EXPAND   : EXPAND_TYPE;
    WORK     : WORK_TYPE;
  end record;

type LIST_TYPE is
  record
    NODE_CONNECT1  : NODE_CONNECT_TYPE;
    NODE_CONNECT2  : NODE_CONNECT_TYPE;
    NODES          : NODE_LIST_TYPE;
  end record;

type MATRIX_ACCESS   is access MATRIX_TYPE;
type RIM_ACCESS      is access RIM_TYPE;
type LIST_ACCESS     is access LIST_TYPE;

MATRIX       : MATRIX_ACCESS;
MATRIX_RIM   : RIM_ACCESS;
LIST         : LIST_ACCESS;

procedure Dispose_Matrix is new
  Unchecked_Deallocation (MATRIX_TYPE, MATRIX_ACCESS);

procedure Dispose_Matrix_Rim is new
  Unchecked_Deallocation (RIM_TYPE, RIM_ACCESS);

procedure Dispose_List is new
  Unchecked_Deallocation (LIST_TYPE, LIST_ACCESS);

begin
  Count the number of mutual, inductive and general components in the circuit.
  if (there are more branches than the maximum) then
    Generate a branch inductance matrix from the data in the netlist.
    Generate the reciprocal inductance coefficient matrix which is obtained from inverting the
      branch inductance matrix.
    Generate a node-column reference table. This is done by calculating the number of
      columns (NO_OF_COLUMNS) in the expand matrix (MATRIX.EXPAND).
```

```
    Generate an expanded reciprocal inductance coefficient matrix, which, when used with the
      node voltage solution, will yield all the inductive branch currents.
    Find the reciprocal inductance matrix from the expand matrix.
    Update the equivalent inductors that have been generated.
  else
    Raise the exception TOO_MANY_BRANCHES;
  end if;
exception
  when (there is not enough memory to create the matrices) =>
    Raise the exception STORAGE_ERROR;
end BR_Inductance_Matrix;
```

Calculate_Mutual_Inductance

Calculates the mutual inductance for the whole netlist. The LIST pointer is not affected.

```
begin
  if (the high node1 = the low node1 or the high node2 = the low node2) then
    Raise the exception MEANINGLESS_COMPONENT_CONNECTION.
  end if;
  if (the high node1 < the low node1) then
    Swop the high node1 with the low node1.
  end if;
  if (the high node2 < the low node2) then
    Swop the high node2 with the low node2.
  end if;
  while (there is still a branch in the circuit and both inductance values are zero) loop
    if (the nodes are in this branch) then
      if (a flag is set) then
        Set the second inductance value to the actual inductor in this branch.
      else
        Set the first inductance value to the actual inductor in this branch.
        Select the first branch in the circuit.
        Set a flag.
      end if;
    else
      Move to the next branch in the circuit.
    end if;
  end loop;
  if (either inductance value is zero) then
    Raise the exception MEANINGLESS_COMPONENT.
  end if;
  Calculate the mutual inductance value.
end Calculate_Mutual_Inductance;
```

Check_Topology

Checks the topology of the netlist. The LIST pointer is not affected.

```
begin
  if (the highest node = 0) then
    Raise the exception NO_CIRCUIT.
  end if;
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (there is voltage controlled current source in the circuit) then
      Select the first branch in the circuit.
      while (there is still a branch in the circuit and the controlling nodes have not been
        found) loop
        if (the controlling nodes of the voltage controlled current source are in this branch)
          then
          The controlling nodes of the voltage controlled current source have been found.
        else
          Move to the next branch in the circuit.
        end if;
      end loop;
      if (a voltage controlled current source exists and the controlling nodes have not been
        found) then
        Raise the exception NO_VCCS_CONTROLLING_NODE;
      elsif (the controlling nodes have been found) then
        Move to the next branch in the circuit.
      end if;
    else
      Move to the next branch in the circuit.
    end if;
  end loop;
  loop
    Increment the node number.
    Select the first branch in the circuit.
    while (there is still a branch in the circuit and the circuit is floating) loop
      if (there is a branch between this node and the reference node) then
        The circuit is not floating.
      else
        Move to the next branch in the circuit.
      end if;
    end loop;
    Exit this loop when the highest node in the circuit has been reached or the circuit is not
      floating.
  end loop;
  if (the circuit is floating) then
    Raise the exception REFERENCE_NODE_MISSING.
  end if;
  Select the reference node.
  loop
    Increment the node number.
    Select the first branch in the circuit.
    while (there is still a branch in the circuit and this node has not been found) loop
      if (this node is in the branch) then
        This node has been found.
      else
        Move to the next branch in the circuit.
      end if;
    end loop;
    if (this node has not been found) then
      Raise the exception SPECIFIC_NODE_MISSING;
```

```
    end if;
    Exit this loop when the highest node in the circuit has been reached.
  end loop;
end Check_Topology;
```

Clear_Branch

Sets all the component values in the branch pointed to by the LIST pointer to zero.

```
begin
  Clear this branch in the netlist.
end Clear_Branch;
```

Clear_Locations

Clears all the LOCATIONS of the NETLIST_TYPE object so that they do not point to any branch.

```
begin
  Set all the LOCATIONS to null.
end Clear_Locations;
```

Convert_Admittance_To_RLC

Converts the representation of components from admittance to RLC form for the whole netlist. The LIST pointer is not affected.

```
begin
  if (there are mutual components in the circuit) then
    Raise the exception CANNOT_CONVERT_WITH_MUTUAL_INDUCTANCES.
  end if;
  while (there is still a branch in the circuit) loop
    if (there is capacitive admittance) then
      if (there is an actual inductor) then
        Subtract it's impedance from the capacitive impedance.
      end if;
      Calculate the new capacitor value.
    end if;
    if (there is inductive admittance) then
      Subtract capacitive admittance from the inductive admittance.
      if (there is still inductive admittance) then
        Calculate the actual inductor values.
      else
        Set the actual inductor values to zero.
      end if;
      Calculate the branch inductance.
```

```
    end if;
    if (there is voltage controlled current source) then
      if (there is an imaginary admittance) then
        Calculate the pole frequency and the magnitude of the admittance from the real and
          imaginary admittance, and the frequency.
      else
        Set the magnitude of the admittance to the real admittance.
      end if;
    end if;
    Move to the next branch in the circuit.
  end loop;
end Convert_Admittance_To_RLC;
```

Convert_RLC_To_Admittance

Converts the representation of components from RLC to admittance form for the whole netlist. The LIST pointer is not affected.

```
begin
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    Calculate the capacitive reactance.
    if (an inductor exists in this branch) then
      Subtract the inductive reactance.
    end if;
    if (a voltage controlled current source exists in this branch) then
      Calculate the real and imaginary admittance from the magnitude of the admittance,
        omega frequency and the 1st order pole frequency.
    end if;
    Move to the next branch in the circuit.
  end loop;
end Convert_RLC_To_Admittance;
```

Count_Components

Counts the number of a specified component type in the whole netlist. The LIST pointer is not affected.

```
begin
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    case (of the type of search component) is
      when (a general component) =>
        Set the high node = general component's high node.
      when (an inductive component) =>
        Set the high node = inductive component's high node.
      when (mutual component) =>
        Set the high node = mutual inductive component's high node.
```

```
    end case;
    if (the high node /= 0) then
      Increment the count of components by 1.
    end if;
    Move to the next branch in the circuit.
  end loop;
end Count_Components;
```

Delete_Branch

Deletes a circuit branch. The LIST pointer must point to the branch to be deleted.

```
begin
  if (there is at least one branch) then
    if (it is the first branch) then
      Delete the first branch.
    else
      Find the req[illegible] branch.
      Delete that [illegible].
    end if;
  end if;
end Delete_Branch;
```

Empty_Branch

Checks to see if a branch is empty. The LIST pointer must point to the branch that is checked.

```
begin
  if (the branch is out of range) then
    Raise the exception BRANCH_OUT_OF_RANGE.
  end if;
  for (the component in CAPS to CUPLING) loop
    case (of the component) is
      when (the selected component) =>
        Retrieve the value of the selected component.
    end case;
    if (the selected component exists) then
      The branch is not empty.
      Exit from this loop.
    end if;
  end loop;
end Empty_Branch;
```

Equal_Locations

Checks to see if the branch pointed to by LIST is the same branch pointed to by the pointer in the

LOCATION array indexed by LOCATION_INDEX.

```
begin
  if (the present branch is the same as that index in LOCATION) then
    The branches are the same branches.
  else
    The branches are different branches.
  end if;
end Equal_Locations;
```

Find_Branch

Locates the branch between a specified node pair in the netlist. The LIST pointer points to the found branch.

```
begin
  if (the high node = the low node) then
    Raise the exception MEANINGLESS_COMPONENT.
  end if;
  if (the high node < the low node) then
    Swop the values of the two nodes.
  end if;
  Select the first branch in the circuit.
  while (there is still a branch in the circuit and the required branch has not yet been found) loop
    if (the high and low nodes exist in this branch) then
      The branch has been found.
    else
      Move to the next branch in the circuit.
    end if;
  end loop;
  if (the branch has not been found) then
    if (the component is >= CR_ADMIT) then
      Raise the exception NODE_PAIR_DOES_NOT_EXIST.
    else
      Create a new branch and set all the values in it to zero.
    end if;
  end if;
exception
  when (the memory is not enough to create a new branch) =>
    Raise the exception STORAGE_ERROR.
end Find_Branch;
```

Find_Dependent_Source

Locates the branch that contains a dependent source given the nodes related to the dependent source in the circuit. The LIST pointer points to the found branch.

```
begin
  Select the first branch in the circuit.
  while (there is still a branch in the circuit and the dependent source has not been found) loop
    if (the dependent source exists in this branch) then
      The dependent source has been found.
    else
      Move to the next branch in the circuit
    end if;
  end loop;
  if (the dependent source has not been found) then
    Select the first branch in the circuit.
    while (there is still a branch in the circuit and the dependent source has not been found) loop
      if (there is not a dependent source in this branch) then
        The dependent source has been found.
      else
        Move to the next branch in the circuit.
      end if;
    end loop;
  end if;
  if (the dependent source has not been found) then
    if (a new branch must be created) then
      Create a new branch.
    end if;
    Set a parameter to say that the source has not been found.
  else
    Set a parameter to say that the source has been found.
  end if;
exception
  when (there is not enough memory to create a new branch) =>
    Raise the exception STORAGE_ERROR.
end Find_Dependent_Source;
```

Find_Mutuals_and_Coupling

Finds the mutuals and coupling factors for the whole netlist. The LIST pointer is preserved.

```
begin
  Select the first branch in the circuit.
  while (there is still a branch in the circuit and the mutual inductance has not been found) loop
    if (mutual inductance exists in this branch) then
      The mutual inductance has been found.
    else
      Move to the next branch in the circuit.
    end if;
  end loop;
  if (the mutual inductance has not been found) then
    Select the first branch in the circuit.
    while (there is still a branch in the circuit and the mutual inductance has not been found) loop
      if (there is no mutual inductance in this branch) then
        Terminate the search.
      else
        Move to the next branch in the circuit.
      end if;
```

```
    end loop;
  end if;
  if (the mutual inductance is not found and a new branch must be created) then
    Create the new branch.
  end if;
  if (the branch must be created) then
    case (of the type of component) is
      when (mutual inductance) =>
        if (the coupling > the maximum possible value) then
          Set the coupling to the maximum possible value.
        end if;
        Store the value of the component in the netlist branch.
        Calculate the coupling.
        Store the value of the coupling in the netlist branch.
      when (coupling) =>
        if (the input value > the maximum possible value) then
          Set the value to the maximum possible value.
        end if;
        Store the value of the component in the netlist branch.
        Calculate the mutual inductance.
        Store the value of the mutual inductance in the netlist branch.
    end case;
  end if;
exception
  when (there is not enough memory to create a new branch) =>
    Raise the exception STORAGE_ERROR.
end Find_Mutuals_and_Coupling;
```

Find_Node_Maximum

Finds the highest numbered node in the whole netlist. The LIST pointer is not affected.

```
begin
  if (there is no circuit) then
    Set NODE_MAXIMUM = 0.
  else
    Select the first branch in the circuit.
    Set NODE_MAXIMUM = the higher node.
    while (there is still a branch in the circuit) loop
      if (the higher node > NODE_MAXIMUM) then
        Set NODE_MAXIMUM = the higher node.
      end if;
      Move to the next branch in the circuit.
    end loop;
  end if;
end Find_Node_Maximum;
```

First_Branch

Sets the LIST pointer to the first branch in the netlist.

```
begin
  Set the LIST pointer to the head of the netlist.
  if (there is not even one branch in the circuit) then
    Raise the exception NO_CIRCUIT.
  end if;
end First_Branch;
```

Initialize

Initialises the whole netlist.

```
begin
  Clear the netlist and set NODE_MAXIMUM = 0 and FREQUENCY = 1000.0.
end Initialize;
```

New_Branch

Creates a new branch and puts it at the head of the netlist.

```
begin
  Create a new branch.
exception
  when (there is not enough memory to create a new branch) =>
    Raise the exception STORAGE_ERROR.
end New_Branch;
```

Next_Branch

Moves the LIST pointer to the next branch in the netlist.

```
begin
  if (there is still a branch in the circuit) then
    Set the LIST pointer to the next branch in the netlist.
  else
    Raise the exception NO_MORE_BRANCHES.
  end if;
end Next_Branch;
```

Read_Netlist

Reads the value of a component as well as the nodes to which it is connected. The component data is read from the branch pointed to by LIST.

```
begin
  if (the branch is out of range) then
    Raise the exception BRANCH_OUT_OF_RANGE.
  end if;
  case (of the type of component) is
    when (a certain type of component) =>
      Retrieve the component's value from the correct place in this branch.
  end case;
  case (of the type of component) is
    when (a certain type of component) =>
      Retrieve the node data associated with this component from the correct place in this branch.
  end case;
end Read_Netlist;
```

Retrieve_Location

Sets LIST to the pointer of the LOCATION array indexed by LOCATION_INDEX.

```
begin
  Set the present branch pointed to by LIST to that pointed to by the pointer indexed by
    LOCATION_INDEX.
end Retrieve_Location;
```

Save_Location

Sets the pointer of the LOCATION array indexed by LOCATION_INDEX to the present branch pointed to by LIST.

```
begin
  Set the pointer indexed by LOCATION_INDEX to the present branch pointed to by LIST.
end Save_Location;
```

Update_Mutuals_and_Coupling

Updates the mutuals and coupling factors for the whole netlist. The LIST pointer is preserved.

```
begin
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (the mutual inductance is between the first node pair) then
      if (the input value is significant) then
        Calculate the mutual inductance.
        Take the coupling into account.
      else
        Store the coupling value in the netlist.
```

```
      end if;
      Store the mutual inductance value in the netlist.
    end if;
    Move to the next branch in the circuit.
  end loop;
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (the mutual inductance is between the second node pair) then
      if (the input value is significant) then
        Calculate the mutual inductance
        Take the coupling into account.
      else
        Store the coupling value in the netlist.
      end if;
      Store the mutual inductance value in the netlist.
    end if;
    Move to the next branch in the circuit.
  end loop;
end Update_Mutuals_and_Coupling;
```

Write_Frequency

Changes the present frequency for the netlist.

```
begin
  Set the netlist frequency to the input frequency.
end Write_Frequency;
```

Write_Netlist

Writes the value of a component as well as the nodes to which it is connected. The branch into which this data is written is pointed to by LIST.

```
begin
  if (the branch is out of range) then
    Raise the exception BRANCH_OUT_OF_RANGE.
  end if;
  if (the component value is invalid) then
    Set the component value to zero.
  end if;
  case (of the type of component) is
    when (a certain type of component) =>
      Store the component value in the correct place in this branch.
  end case;
  case (of the type of component) is
    when (a certain type of component) =>
      Store the node data associated with this component in the correct place in this branch.
  end case;
  if (the component value = 0.0 and the branch is empty) then
```

```
    Delete this branch from the circuit.
  elsif (the high node 1 > the highest node in the circuit) then
    Set the highest node in the circuit = the high node 1.
  end if;
end Write_Netlist;
```

6.4.4 Outer Methods

Change_Directory

Changes the default directory and shows the new path.

```
begin
  Display process title.
  Display the present path on the main screen.
  Cursor_Next_Line;
  Get the new path from the user and set 0 for the last character (ASCIIZ).
  Change to a new directory.
  Display the present path on the main screen.
end Change_Directory;
```

Change_Drive

Changes the present drive device and shows the new path.

```
begin
  Display process title.
  Display the present path on the main screen.
  Cursor_Next_Line;
  Get the new drive from the user.
  Change to the drive.
  Display the present path on the main screen.
end Change_Drive;
```

Get_Frequency

Displays the present frequency on the Prompts screen and then gets a new frequency if the present one is not suitable.

```
begin
  Display the present frequency and ask if it is acceptable.
  if (the frequency is not acceptable) then
    Get the new frequency from the keyboard.
```

```
    Make sure that the new frequency is within limits.
  end if;
end Get_Frequency;
```

Get_Load_File

Gets the name of a disk file for loading a netlist or Monte-Carlo netlist. The existing files containing netlists or Monte-Carlo netlists are also displayed.

```
begin
  if (the file to load is a monte file) then
    Set the extension to MNT.
  else
    Set the extension to NTL.
  end if;
  Display the present path on the main screen.
  Get the first filename that has a NTL or MNT extension.
  if (there is no file in default directory) then
    Write an error message that there are no recognised files.
    Raise the exception USER_ESCAPE.
  end if;
  while (there is a file) loop
    Add the filename to the dynamic list.
    Find the next filename that has a NTL or MNT extension.
  end loop;
  begin
    Get the name of the monte file to load.
    Try open the file to read it.
    Exit from this loop.
  exception
    when (the disk file does not exist) =>
      Display an error that the file cannot be found.
  end;
end Get_Load_File;
```

Get_Save_File

Gets the name of a disk file for saving a netlist or Monte-Carlo netlist. The existing files containing netlists or Monte-Carlo netlists are also displayed.

```
begin
  if (the file to load is a monte file) then
    Set the extension to MNT.
  else
    Set the extension to NTL.
  end if;
  Display the present path on the main screen.
  Get the first filename that has a NTL or MNT extension.
```

```
  loop
    if (there is such a file) then
      if (there are less than 100 files listed) then
        Display the filename on the main screen.
      else
        Clear the main screen and display the present path.
      end if;
    else
      Exit from this loop.
    end if;
    Find the next filename that has a NTL or MNT extension.
  end loop;
  loop
    begin
      Ask the user for the name of the file that will contain the data for the Netlist object.
      Check if the disk file exists by attempting to open the file.
        An exception will be raised if the file does not exist.
      Close the opened file.
      Tell the user that the file already exists and ask if the file should be overwritten.
      if (the file must be overwritten) then
        Open the file again, but this time in writing mode.
        Exit from this loop.
      end if;
    exception
      when (the disk file does not exist) =>
        Create a new file and exit from this loop.
    end;
  end loop;
end Get_Save_File;
```

Initialize_Directory

Gets the present path.

```
begin
  Gets the present default drive and directory.
end Initialize_Directory;
```

List_Branch

Lists the components in a branch on the main screen.

```
begin
  for (the COMPONENT in CAPS to CUPLING) loop
    case (of the type of component) is
      when (CAPS | INDUCT | CM_ADMIT | POLE_FREQ) =>
        if (the mode of listing = RLC) then
          The component must be listed.
        end if;
```

```
      when (GI_ADMIT | CR_ADMIT | CI_ADMIT) =>
        if (the mode of listing = admittance) then
          The component must be listed.
        end if;
    end case;
    Read the component data from the netlist.
    Select appropriate units for this component.
    if (the component exists in this branch) then
      if (the equivalent inductance form has been chosen) then
        Set the display name of the component.
      end if;
      if (the component = GR_ADMIT) then
        if (the mode of list is RLC) then
          Set the display name and units in impedance form for this component.
          Find the reciprocal value.
        else
          Set the display name and units in admittance form for this component.
        end if;
      end if;
      if (the component is a voltage or current source) then
        if (the component is a voltage source) then
          Set the display name for a voltage source.
        else
          Set the display name for a current source.
        end if;
        case (of the type of component) is
          when (a real voltage source or a real current source) =>
            The value is left unaltered.
          when (an imaginary current source or an imaginary voltage source) =>
            Calculate the magnitude and phase of the source.
            if (the component has a significant magnitude and the display form is equivalent
              inductors) then
              Display the component, the high and low nodes, the magnitude, the phase and
                the units of the component.
            end if;
            The component must be listed.
        end case;
      elsif (the component < CR_ADMIT and the display form is not equivalent inductors and
        the component /= equivalent inductors and the component must be listed or the
        component = equivalent inductor and the display form is equivalent inductors) then
        Display the component, the high and low nodes, it's value and it's units.
      end if;
      if (there is a second node pair associated with this component and the component must
        be listed and the display form is not equivalent inductors or procedure form) then
        Display the component, the high and low nodes, it's value and it's units. The coupled
          or controlling nodes are also shown.
      end if;
    end if;
  end loop;
end List_Branch;
```

Load_Netlist

Recovers a netlist from a disk file by reading the values of the components found in the specified file.

```
begin
  Get the file to load the netlist from.
  Initialise the netlist object.
  if (the file pointer is not at the end of the file) then
    Read the title and frequency of the Netlist object.
  end if;
  while (the file pointer is not at the end of the file) loop
    Read the component values for a branch of the Netlist object.
    Create a new branch and set the values of the components to those read from the file.
    for (the component in CAPS to CUPLING) loop
      Read the chosen component's value from the Netlist object.
      if (the chosen component exists) then
        Check that the chosen component's value is valid.
        Write the chosen component's value into the Netlist object.
      end if;
    end loop;
  end loop;
  Close the disk file.
end Load_Netlist;
```

Load_Monte_Netlist

Recovers a Monte-Carlo netlist from a disk file by reading the values found in the specified file.

```
begin
  Get the file to load the monte netlists from.
  Initialise both monte netlists.
  while (the end of the file has not been reached) loop
    Read an item from the disk file.
    Create a new branch for monte netlist 1.
    Set this new monte branch equal to the item from the disk.
    Read an item from the disk file.
    Create a new branch for monte netlist 2.
    Set this new monte branch equal to the item from the disk.
  end loop;
  Close the disk file.
end Load_Monte_Netlist;
```

Remove_Node

Removes a node from the netlist. The components that are deleted are displayed on the main screen.

```
begin
  if (the highest node in the circuit = 0) then
    Raise the exception NO_CIRCUIT.
  end if;
  Get the node to delete from the user.
  if (the node to be deleted is the reference node (0)) then
    Raise the exception ATTEMPTED_REFERENCE_NODE_REMOVAL.
  elsif (the node to be deleted is not valid) then
    Raise the exception DELETE_NODE_DOES_NOT_EXIST.
  end if;
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (the node to be deleted exists in this branch) then
      List the components in the branch on the screen.
      if (the higher and lower nodes for the dependent source components = 0) then
        Delete this branch from the circuit.
      else
        Set all the general components and node data to zero.
      end if;
    end if;
    Move to the next branch in the circuit.
  end loop;
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (the higher or lower nodes of the dependent source components = to the node to be
      deleted) then
      Set all the dependent source components and the node data to zero.
    end if;
    Move to the next branch in the circuit.
  end loop;
  if (the node cannot be found) then
    Raise the exception DELETE_NODE_DOES_NOT_EXIST.
  end if;
  Find the highest node in the circuit.
exception
  when ATTEMPTED_REFERENCE_NODE_REMOVAL =>
    Display an error message.
  when DELETE_NODE_DOES_NOT_EXIST =>
    Display an error message.
end Remove_Node;
```

Renumber_Nodes

Renumbers an existing node. Certain voltage and current directions are changed. The LIST pointer is not affected.

```
begin
  if (the highest node in the circuit = 0) then
    Raise the exception NO_CIRCUIT.
  end if;
  Get the old and new node numbers.
  if (the old node does not exist) then
```

```
    Raise the exception OLD_NODE_DOES_NOT_EXIST.
  elsif (the new node already exists) then
    Raise the exception NEW_NODE_ALREADY_EXISTS.
  end if;
  while (there is still a branch in the circuit) loop
    if (the old node exists in this branch) then
      Set the old node = the new node.
    end if;
    Move to the next branch in the circuit.
  end loop;
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (a voltage source exists in this branch) then
      if (the higher node < the lower node (now renumbered)) then
        Negate the real voltage.
        Negate the imaginary voltage.
        Swop the higher node number with the lower node number.
      end if;
    end if;
    if (a current source exists in this branch) then
      if (the higher node < the lower node  (now renumbered)) then
        Negate the real current.
        Negate the imaginary current.
        Swop the higher node number with the lower node number.
      end if;
    end if;
    if (the higher node < the lower node  (now renumbered)) then
      Swop the higher node number with the lower node number.
    end if;
    Set SWITCHED to false.
    if (there is a dependent source in this branch) then
      if (the higher node < the lower node) then
        Set SWITCHED to true.
        Swop the higher node number with the lower node number.
      end if;
      if (the higher controlling node < the lower controlling node) then
        Set SWITCHED to true.
        Swop the higher node number with the lower node number.
      end if;
      if (SWITCHED is true) then
        Negate the real admittance.
        Negate the imaginary admittance.
        Negate the magnitude admittance.
      end if;
    end if;
    Set SWITCHED to false.
    if (there is coupling to this branch) then
      if (the higher node 1 < the lower node 1) then
        Set SWITCHED to true.
        Swop the higher node 1 number with the lower node 1 number.
      end if;
      if (the higher node 2 < the lower node 2) then
        Set SWITCHED to true.
        Swop the higher node 2 number with the lower node 2 number.
      end if;
      if (SWITCHED is true) then
```

```
          Negate the inductance.
          Negate the coupling factor.
        end if;
      end if;
      Move to the next branch in the circuit.
    end loop;
    Find the highest node in the circuit.
exception
  when NEW_NODE_ALREADY_EXISTS =>
    Display an error message.
  when OLD_NODE_DOES_NOT_EXIST =>
    Display an error message.
end Renumber_Nodes;
```

Save_Netlist

Saves the netlist by saving the values of the components to a specified disk file.

```
begin
  if (there is no circuit) then
    Raise the exception NO_CIRCUIT.
  end if;
  Get the file to save the netlist in.
  Write the title and frequency of the Netlist object to the file.
  Select the first branch in the circuit.
  loop
    Write the component values in this branch to the file.
    begin
      Move to the next branch in circuit.
    exception
      when (there are no more branches) =>
        Exit from this loop.
    end;
  end loop;
  Close the disk file.
end Save_Netlist;
```

Save_Monte_Netlist

Saves the Monte-Carlo netlist to a specified disk file.

```
begin
  Get the file to save the monte netlists in.
  Select the first branch in both monte netlists.
  loop
    Write the information from monte netlist 1 to disk.
    Write the information from monte netlist 2 to disk.
    begin
      Move to the next branch in both monte netlists.
```

```
    exception
      when (there are no more branches) =>
        Exit from this loop.
    end;
  end loop;
  Close the disk file.
end Save_Monte_Netlist;
```

Set_Circuit_Title

Displays the title of the circuit which can be changed.

```
begin
  Display the title of the circuit.
  Ask if the title needs to be changed.
  if (the title needs to be changed) then
    Get the new title from the keyboard.
  end if;
end Set_Circuit_Title;
```

Show_Node

Lists all the components that are connected between a specified node pair.

```
begin
  if (the highest node in the netlist is zero) then
    Raise the exception NO_CIRCUIT.
  end if;
  Get the node pair to show from the user.
  if (the higher show node < the lower show node) then
    Swop the show nodes.
  end if;
  Display the show nodes and a heading with Component, Nodes, Value and Units.
  Select the first branch in the circuit.
  while (there is still a branch in the circuit) loop
    if (the branch to be displayed is this branch) then
      List the components in this branch.
      for (the component in CR_ADMIT to POLE_FREQ) loop
        Read the component data from the netlist.
        if (there is a second node pair associated with this component and there is such a
          component in this branch) then
          Display the component, the high and low nodes, it's value and it's units. The
            coupled or controlling nodes are also shown.
        end if;
      end loop;
    end if;
    Move to the next branch in the circuit.
  end loop;
end Show_Node;
```

CHAPTER 7

DOS ENVIRONMENT PACKAGE

7.1 Introduction

Ada is meant to be as portable as possible, and so it does not have procedures and functions that deal with specific features of input and output devices. This is a problem for programs like Elector which uses many features of the video display extensively.

Ada has the *Put* and *Get* subprograms in the *Text_IO* package for text input and output. These subprograms are used for mainly Teletype output and are therefore of limited use. Ada does have mechanisms for implementation of computer and operating system dependent features. The mechanisms that are of particular use are that of the pragma interface and that of record representation specifications.

The PC computer has low-level BIOS (Basic Input Output System) routines which provide some of the required facilities. To access these routines, Assembler language functions have to be created and connected to Ada subprograms. These functions can be divided into two groups :

- Functions that directly implement some system feature.
- Functions that implement two memory buffers and related operations for the Window object.

7.2 Interfacing to Other Languages

7.2.1 Pragma Interface

The pragma *interface* allows subroutines written in other languages to be used, as long as all communication is done by parameter and function values. The pragma *interface* specifies the other language name and must be declared after the subprogram specification declaration to which it applies. A body is not allowed for this subprogram since the body is to be written in another language. The form of the pragma in standard Ada is

```
pragma INTERFACE (language_name, subprogram_name);
```

7.2.2 Meridian Ada Interface Implementation

The form of the pragma interface in Meridian Ada [Meridian, 1990] is :

```
pragma INTERFACE (language_name, subprogram_name [,"link_name"]);
```

- *language_name* : Can be Assembly, Builtin, Microsoft_C or Internal. Builtin and Internal are reserved by Meridian.
- *subprogram_name* : The name of a subprogram whose name appears in the same compilation unit.
- *link_name* : A string constant specifying the name of the non-Ada subprogram corresponding to the Ada subprogram named in the second parameter (subprogram_name). The *link_name* can be omitted if the Ada subprogram name is the same as the *link_name*. The *link_name* is always translated to uppercase.

7.2.3 Interface to Assembly Language

When calling assembly language routines, some important points must be considerations :

- Subprograms can only be called if they are declared as FAR PROCs.
- Data segments must be of the class "_DATA".
- Code segments must be of the class "CODE".
- If a case-sensitive assembler is used, it must be remembered that the linker is also case sensitive.
- Segments are always aligned on paragraph boundaries.

7.2.4 Stack Frames

The stack is used for passing parameters to assembly language subprograms. Stack frames must be properly maintained. Otherwise, unpredictable events could occur, including the corruption of variables and even *hanging* the computer.

The stack frame consists of :

- The actual parameters.
- The return address.
- A display. This consists of copies of the frame pointers from previous subprogram activations which allow references to local and non-local objects.
- Local variables.

Points to remember :

- All parameters are passed on the stack.
- Parameters are passed by pushing them onto the stack in reverse order of formal parameter declarations (right to left).

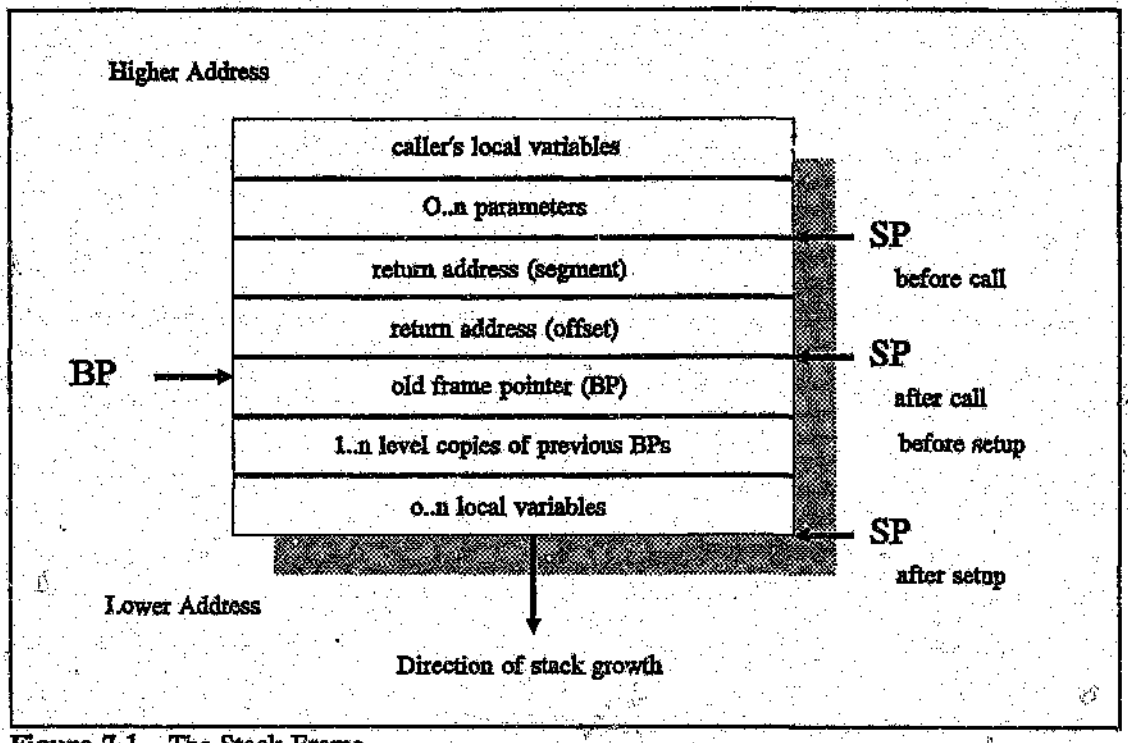

Figure 7-1 The Stack Frame

- The number and types of the parameters must be known in order to access the correct stack locations.
- Non-scalar objects are passed by reference, regardless of mode.
- The assembly language must de-allocate the stack frame of parameters before returning.
- Allocating space for local variables and saving previous frame pointers is the responsibility of the subprogram.
- Registers SS, DS and BP must be preserved. All other registers may be destroyed.
- Values returned by assembly language functions must be arranged as follows:
  (1) 8 bit values are returned in CL
  (2) 16 bit values are returned in CX
  (3) 32 bit values are returned in BX:CX
  (most significant in BX, least significant in CX)
  (4) 64 bit floating point values are returned in the 80x87 stack top

7.3 Representations of Various Types

Data is transferred to and from data structures created by the Meridian compiler. Thus, the compiler's implementation of these data structures must be known.

7.3.1 Discrete Types

An enumeration type occupies 8, 16 or 32 bits depending on the number of values that are declared for the enumeration. In other words, the smallest number of bits, rounded up to the nearest number of bytes.

Data Types	Unpacked	Packed
Boolean	8 bits	1 bit
Character	8 bits	7 bits
Integer	16 bits	16 bits
Long_Integer	32 bits	32 bits

Table 7-1 Discrete Type Representation

Data Types	Bits
Fixed point, short	16
Fixed point, long	32
Floating point	64

Table 7-2 Real Type Representation

7.3.2 Real Types

The 64 bit floating point format of the 80x87 is used for all floating point values.

7.3.3 Array Types

A constraint array object with static bounds occupies just the space required for it's elements. A dynamic array object is represented as a pointer to a memory area containing the actual array elements. The memory area for the array elements is created on the dynamic allocation heap. Separate temporary objects hold the variable bounds of the array. An unconstraint array object (usually a parameter to a subprogram) is represented as an array access object.

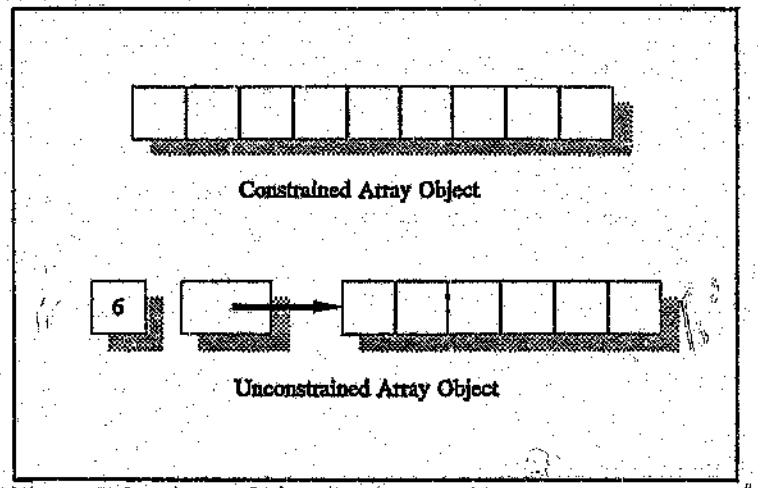

Figure 7-2 Array Object Implementation

7.3.4 Record Types

The components of a record are arranged in the order in which they are declared. In a discriminant record, the discriminant are treated as components of the record. They are the first components of the record. An additional Boolean component appears at the start of a record for records whose discriminants may vary at runtime.

Record representation specifications follow the record type declarations to which they apply. Additional space may be allocated between components in a record to maintain alignments.

7.3.5 Access Types

Access objects for most types are represented simply as pointers to those objects. However, unconstraint array access objects are represented by :

- A pointer to the actual array data.
- The lower and higher bounds for each index.

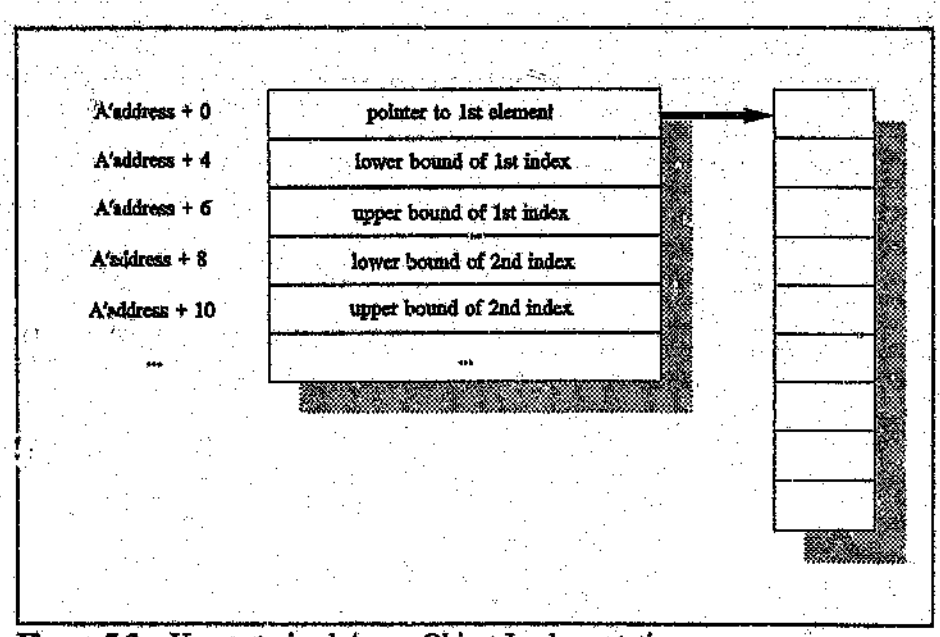

Figure 7-3 Unconstrained Array Object Implementation

7.4 System Features

7.4.1 BIOS, DOS and Mouse Interrupts

Most of the functions use BIOS, DOS or Mouse routines. In some instances they may slow down processing, but the benefits they offer adequately compensate. Some benefits are;

- *Portability* at the Assembly level. All functions are built into the Personal Computer. These

functions are called by interrupts which are the same on all PCs.

- *Understandability*. All these functions are well documented in many books and manuals.
- *Reliability*. The manufacturer will have tested these functions and made sure that they work as expected.

System Feature	Function
Disk	Change_Directory Change_Drive Find_First_File Find_Next_File Get_Drive_Directory
Graphic Screen	Change_Screen Character_To_Graphic Clear_Screen Dot_Horizontal_Line Dot_Vertical_Line Line Plot Plot_Four_Pixels Set_Plot_Colour Set_Plot_Window String_To_Graphic
Hardware Cursor	Change_Cursor_Shape Read_Cursor Set_Cursor Show_Cursor Hide_Cursor
Keyboard	Keyboard_Character_Ready Get_Keyboard_Character
Mouse	Check_For_Mouse Read_Text_Mouse Set_Text_Mouse Show_Text_Mouse Hide_Text_Mouse
Text Screen Character Writing with Attributes	String_To_Screen
Text Screen Display	Clear_Screen Scroll_Up Scroll_Down

Table 7-3 System Features

BIOS Function		Description of the Function
Interrupt	Function	
10h	0	Changes the mode of the screen eg. text to graphic
10h	1	Set the shape of the hardware cursor.
10h	2	Set the position of the hardware cursor.
10h	3	Read the position of the hardware cursor.
10h	6	Scroll up (AL = 0 Clears the screen)
10h	7	Scroll down (AL = 0 Clears the screen)
10h	12	Plot on a graphic screen.
10h	14	Write to screen in teletype mode.
10h	15	Read the screen mode and the active page.
10h	18	Check if EGA is present.
10h	26	Identify display devices for PS/2 and if VGA present.
16h	0	Gets the keycode of a key pressed on the keyboard and removes it from the keyboard buffer. If a key was not pressed, it waits until a key is pressed.
16h	1	Checks if a key has been pressed, and gets it's code if it was pressed. However, the code is not removed from the keyboard buffer.

Table 7-4 BIOS Functions

DOS Function		Description of the Function
Interrupt	Function	
21h	9	Displays a string that is delimited with $.
21h	14	Selects the default disk device.
21h	59	Change the current directory.
21h	71	Gets the default directory for any drive.
21h	76	Terminate program.
21h	78	Finds the first matching file in the path specified.
21h	79	Finds the next matching file in the path specified. Function 78 must be used before this function is used.

Table 7-5 DOS Functions

Mouse Driver		Description of the Function
Interrupt	Function	
33h	0	Check for a mouse, and if present, initialise it.
33h	1	Show mouse pointer.
33h	2	Hide mouse pointer.
33h	3	Read the status of the mouse (position of the mouse pointer and if any buttons were pressed.
33h	4	Set position of mouse pointer.

Table 7-6 Mouse Functions

7.5 Buffering Concepts

There are two different methods for implementing the foundation of the user interface. Both methods need buffers.

7.5.1 Method 1

Two buffers are used for screen operations. The first buffer is used to store the characters and their attributes from the area of screen that will be overwritten by another screen object. Once the screen object has fulfilled it's purpose, it is removed from the screen. This is done by coping the characters and their attributes from the first buffer back to the screen. This gives the impression that the screen object has been removed to reveal the screen underneath.

The second buffer is used to stop flickering caused when the screen object is drawn. This is not a big problem if the screen object is being drawn for the first time. However, when the screen object is moved around the screen, the constant redrawing slows down processing and produces a lot of flicker. By drawing the screen object in a second buffer and then coping it to the screen prevents the flicker. Moving the screen object now requires less processing.

The initial drawing of a screen object is as follows :

(a) Draw the screen in the second buffer. Nothing happens on the screen.
(b) Copy the area of screen that is going to be overwritten by the screen object into the first buffer. Nothing happens on the screen.
(c) Copy the second buffer's contents to the screen. The screen object will appear on the screen.

The moving process is as follows :

(d) Copy the contents of the first buffer back to the screen. The screen object will disappear and the screen underneath will be revealed.

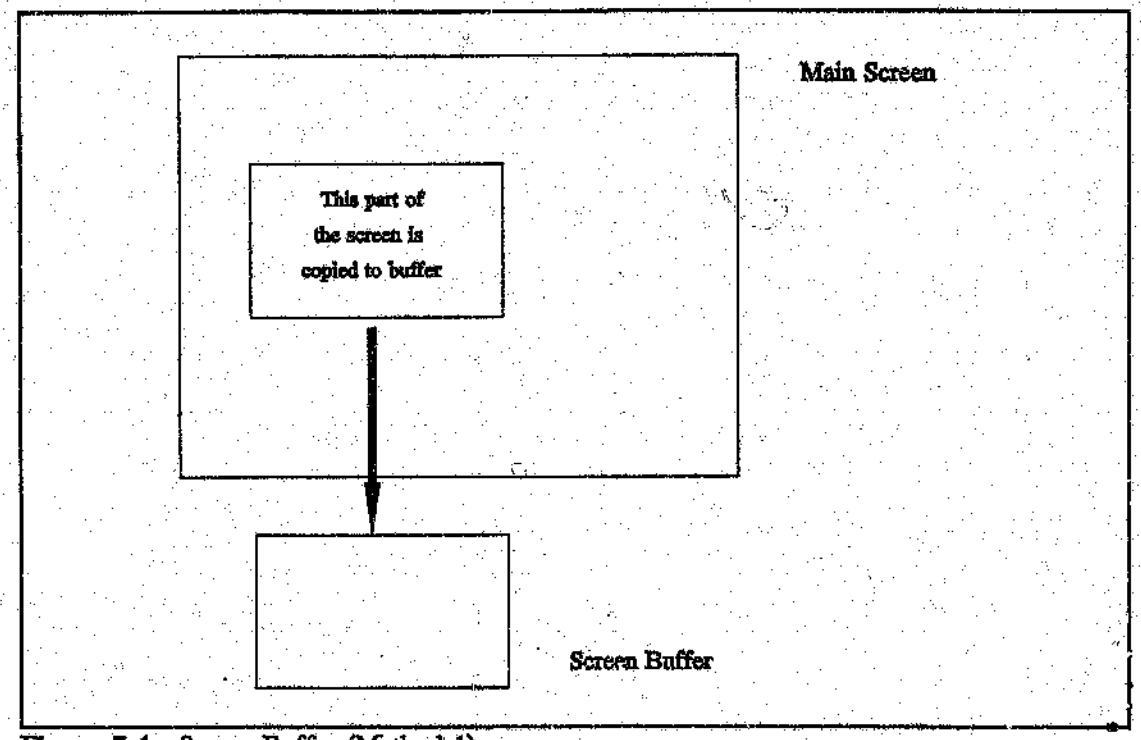

Figure 7-4 Screen Buffer (Method 1)

(e) Copy the area of the screen where the screen object will now be positioned. Nothing happens on the screen.

(f) Copy the second buffer's contents to the screen. The screen object will appear in it's new position on the screen.

7.5.2 Method 2

A minimum of two buffers are used. One buffer (window) is used for a screen object and another (main) for the main screen. Nothing is written directly to the screen, but into one of the buffers. These buffers are then copied to the screen. The order in which the buffers are written to the screen determines which image will appear on top. The main screen is usually copied first and so it appears as the background.

This method works with multiple subscreens and multiple buffers. A more complex user interface system can be developed by having more than one main screen. This requires a separate buffer for each of these main screens. A generalised method to display only portion of a main screen would then copy only part of that buffer to the screen. Thus, a complex windowing system can be developed.

The initial drawing of a screen object is as follows :

(a) Draw the screen object in the window buffer. All the drawings for the main screen are done in the main buffer. Nothing happens on the screen.

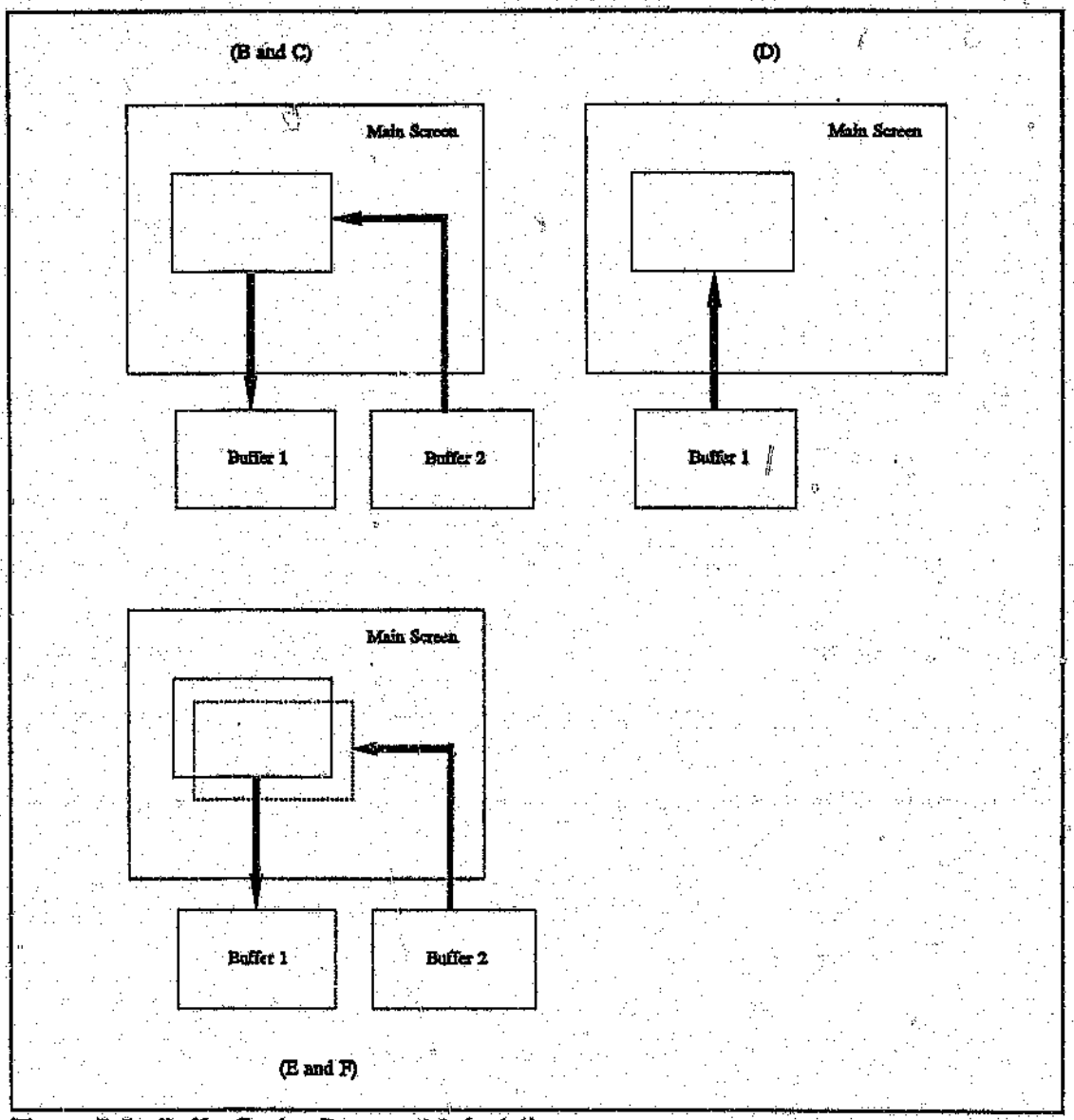

Figure 7-5 Buffer Coping Process (Method 1)

(b) Copy the main buffer to the screen.
(c) Copy the window buffer to the screen. It appears as if the screen object appears in front of the main screen.

The moving process is as follows:

(d) Copy the main buffer to the screen. It appears as if any screen object that was visible is now removed.
(e) Copy the window buffer to a new position on the screen. It appears as if the screen object appears in front of the main screen in a new position.

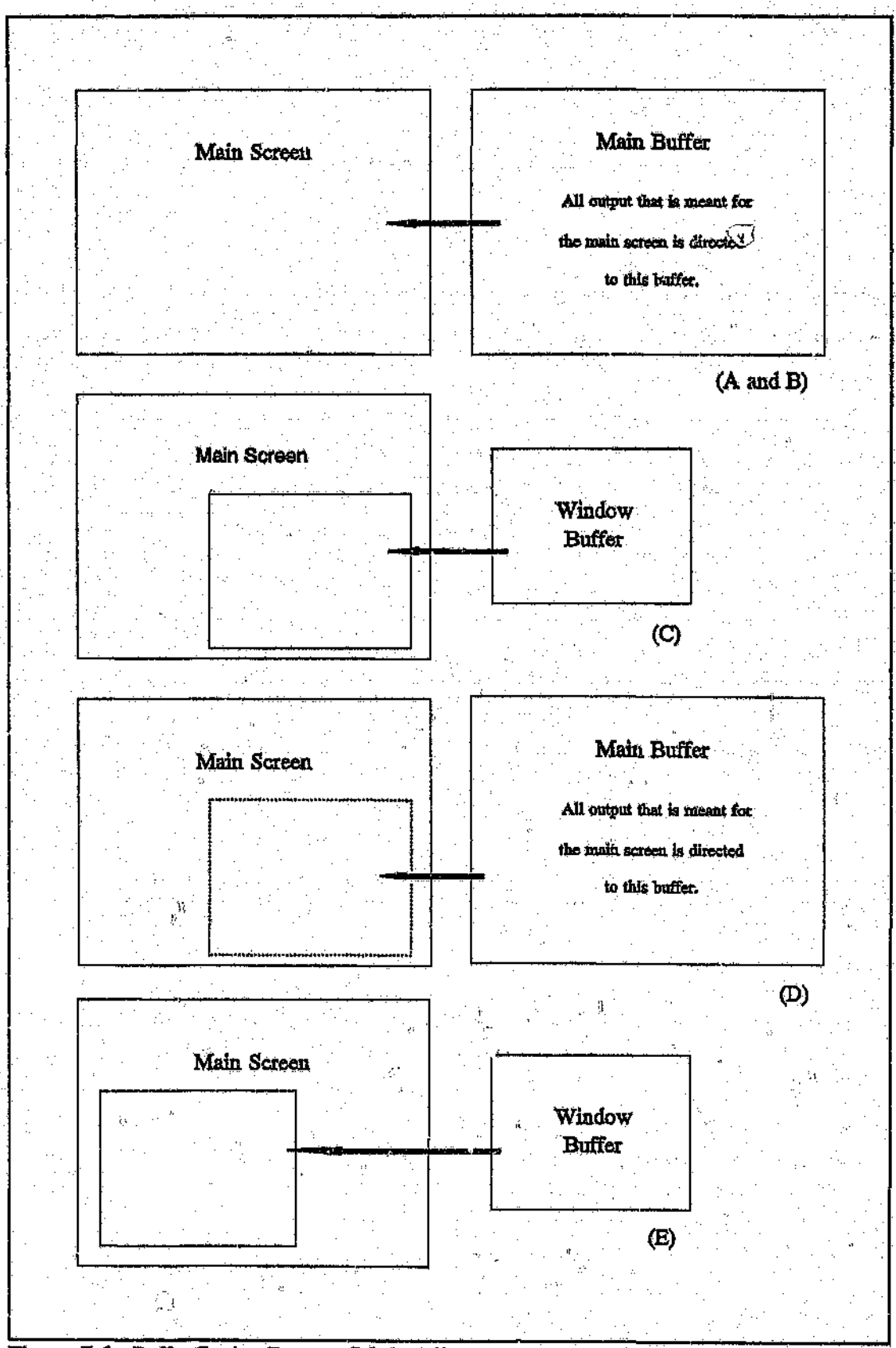

Figure 7-6 Buffer Coping Process (Method 2)

7.5.3 Buffer Implementation

To implement these buffering methods requires transfers of large chunks of memory. The best language for implementation is assembly language because drawing and moving screens objects should be done as fast as possible for it to look both impressive and effective.

The first step in implementing a buffer in assembly language is that the buffer is a linear string of characters while the screen is an array of 80 by 25 characters. However, the addressing of the screen is linear, starting at the top-left corner, going from left to right, to the bottom-right corner. This means that offsets for the actual location within the buffer must be calculated from row and column coordinates.

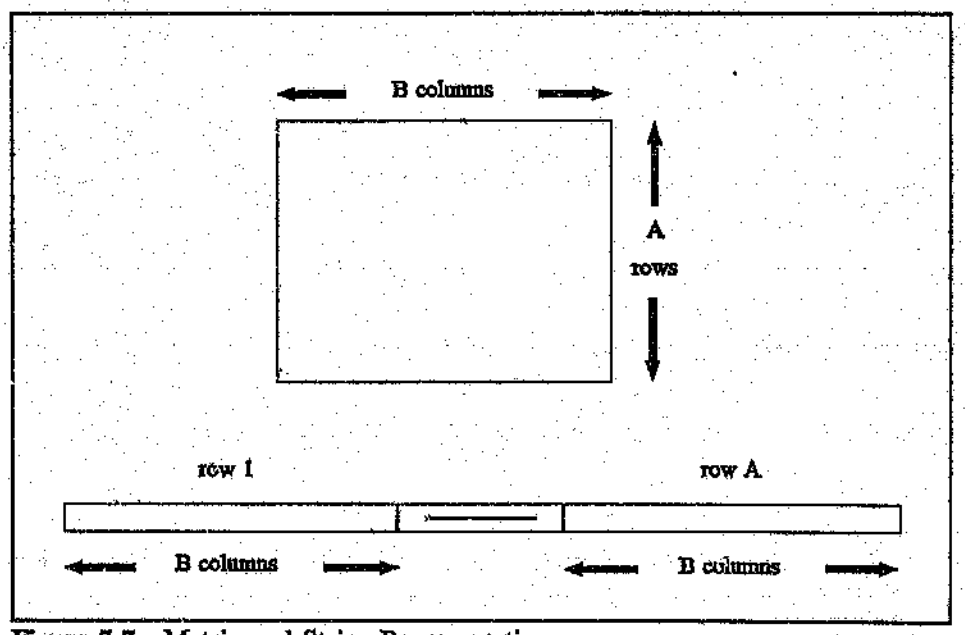

Figure 7-7 Matrix and String Representation

Every odd location starting at the start pointer to the buffer, contains the attributes to a character. Every even location contains the character code. To clear the buffer, all the character location values are changed to spaces (ASCII 32) and the attributes set to a chosen value.

The row and column values for the two buffers are kept in variables which can only be accessed by the assembly language subroutines. These values are used by several assembly language subroutines in calculating what must be copied, and where it must be placed on the screen.

Certain input to some screen objects will cause them to change state which leads to changes in their appearance. They will then redraw themselves in the window buffer (the second buffer). However, their appearance will stay the same on the screen unless the *Window_Buffer_To_Screen* subprogram is called. This copies the second buffer to the screen, thus updating the appearance of that screen object.

Type of Operation	Buffer Method
Initialisation of Window Buffer	Window_Buffer_Setup
Main Buffer Operations	Clear_Main_Buffer Main_Buffer_Line Main_Buffer_To_Screen String_To_Main_Buffer
Window Buffer Operations	Character_To_Window_Buffer Clear_Window_Buffer Frame_Window_Buffer New_Window_Location String_To_Window_Buffer Window_Buffer_Title Window_Buffer_To_Screen
Shadow	Make_Shadow
Line Bar	Make_Line_Bar Remove_Line_Bar

Table 7-7 Buffer Operations

7.5.4 Shadows

To make the window object appear as though it is appearing in front of the screen, a shadow is drawn to the bottom right of it. This is done by setting the attributes of all the characters in the shadow area to light gray on black.

In the first method, the original attribute values for these characters must be saved so that they can be restored when the window object moves or disappears. A small shadow buffer holds the values of these attributes which are copied to it before the shadow is drawn. Once the window object changes position or disappears, the contents of this buffer are copied back to the screen. The shadow disappears from the screen. In the second method, the shadow can be drawn without the need to save the original attributes.

7.5.5 Highlighting Bar

A highlighting bar is needed for the menu objects to highlight menu options. This bar is created by changing the attributes of a line of characters. However, when the bar is moved to the next option, the bar must be removed from the previous option. This is done by restoring the original attributes of this line of characters. The attribute values are copied to a small buffer before the bar is drawn. When the line moves, the contents of this buffer are copied back to the screen. The highlighting bar disappears.

7.5.6 Screen Setup

The screen initialisation method *Screen_Setup* must be called before any buffer or screen writing method is called. This is required because many screen methods write directly to screen memory. If the screen device is a monochrome, the screen memory starts at B000h. If the screen device is a CGA, EGA or VGA, the screen memory starts at B800h.

The available graphic modes are determined by identifying the display device. Mode 13 is used for EGA and VGA adapters. This mode gives 640 x 350 with 16 colours. The best equivalent mode for CGA is mode 6 which gives 640 x 200 with 2 colours. The *Screen_Setup* method makes sure that the correct resolution is used when displaying graphics.

7.6 Mouse Pointer

The pointer of the mouse is displayed on the text screen a block. This block is formed by changing the attributes (foreground and background colour) of the character that is at the same position as the mouse pointer. The attributes are changed by XORing them with some value. By XORing the attributes again with the same value, the original attributes will be restored. This works well if the character's attributes are not changed while the mouse pointer is on it. However, parts of the screen are often written to - usually copying buffers to the screen, making shadows or displaying bars, or even clearing and scrolling. These can all overwrite the previous attributes of the character set by the mouse driver.

The problem is that the mouse driver assumes that the character attribute was changed by it. If the mouse is moved, the mouse driver XORs the attributes of the character the mouse pointer was on. The mouse driver does this thinking that it is removing the mouse pointer and restoring the original attributes. However, the character attributes have changed and XORing them leaves a *hole* in the screen. The *hole* disappears only when the mouse pointer is on top of it, but then so does the mouse pointer. To prevent this, the following procedure should be followed :

(1) Hide the mouse pointer.
(2) Execute the screen modifying process.
(3) Display the mouse pointer.

Also, when copying from the screen to a buffer, the mouse pointer must be hid, otherwise the attributes for one of the characters in the buffer would be wrong. If the mouse pointer is at a different location when the buffer is copied back to the screen, a *hole* will be visible.

7.7 The Structure of Assembler Subroutines

Assembly language subroutines are constructed along certain guidelines [Abel, 1991].

7.7.1 Frame Buffer Setup

The frame buffer needs to be set up before executing the code for the function. On completion, the frame buffer needs to be restored.

```
push    bp              Set up frame buffer
mov     bp, sp
...
Function code fits in here.
...
pop     bp              Restore frame buffer
ret
```

7.7.2 Subprogram Parameters

Assembly language functions that are passed parameters from the Ada subprogram get these off the stack. The way these parameters are retrieved from the stack depends on their type. To illustrate the different methods of retrieval an Ada subprogram heading followed by the assembler code needed to retrieve the variable values is given.

(a) For all scalar objects that are passed to the Ada subprogram with the mode of *in* only, the actual value is passed. This means that the value is retrieved directly from the stack. In the example, the value of VARIABLE1 will be moved into the AX register, and that of VARIABLE2 into the BX register.

```
Ada         :  Procedure_Name (VARIABLE1, VARIABLE2 : in INTEGER);

Assembler   :   mov    ax, 6 [bp]
                mov    bx, 8 [bp]
```

(b) For most other objects, including scalars whose values are passed to the Ada subprogram with the mode *in out* or *out*, the address of the object is passed on the stack. Both the segment and offset addresses are passed - the segment at the higher position on the stack, and the offset at the lower position. In the example, the first assembler instruction will load the ES segment register with the segment address, and the DI register with the offset address. The second assembler instruction does the same as the *les* instruction, except that DS is loaded with the segment address and BX is loaded with the offset address.

```
Ada         :  Procedure_Name (VARIABLE1 : out INTEGER);
Assembler   :  les   di, 6 [bp]     or     lds   bx, 6 [bp]
```

(c) For dynamic unconstraint objects (eg. unconstraint strings usually used as formal parameters in procedures), the address of the pointers to these objects is passed on the stack. This is a two part process :

- The address of the pointer of the unconstrained object is retrieved from the stack.
- The actual value of the pointer to the unconstrained object is then retrieved using the address from the first step. The ES segment override must be used, otherwise the pointer will be set to the value at DS:[DI] instead of ES:[DI].

```
Ada         :  Procedure_Name (VARIABLE1 : in STRING);
Assembler   :  les   di, 6 [bp]
               lds   si, es:[di]
```

The length of the unconstrained object is stored in the next two word locations. To determine the length of the string, use the code below.

```
mov   cx, 8 es:[di]
sub   cx, 6 es:[di]
inc   cx
```

7.8 Assembly Output from the Ada Compiler

Many questions can by answered by examining how the compiler calls assembly language subroutines. A temporary stub subprogram may be written in place of the assembly language subprogram. The stub and the calling subprogram may then be compiled and the assembly language output examined directly.

For the compiler to generate assembly code instead of machine code, use :

ada -fe -S filename.ext

-S : Generates assembly code.

-fe : Annotate assembly language listing. The output is supplemented by comments containing the Ada source statements corresponding to the assembly language code sections written by the code generator.

7.9 Library Management (Version 2.1)

Once the Ada compilation units have been compiled and the assembly language units assembled, they must be linked. BAMP (Build Ada Main Program) will usually link all the necessary program units and produce and executable file. However, with a program like Elector which uses assembler routines, the Ada code must not be linked into an executable file, but a relocatable object file. BAMP is used to get all the separate Ada files and combine them into one large OBJ file. To do this :

bamp -r ada_program_name eg. **bamp -r** elector

After generating an object file, a low-level linker like Microsoft's LINK or Borland TLINK. This will produce an executable file.

link elector dos

7.10 Dos_Environment Subprograms

- *Basic_Plot* : Plots a pixel in the present plot colour on the screen at the position specified. The column must be given in the CX register and the row in the DX register.
- *Change_Cursor_Shape* : Sets the cursor shape. Gets the new cursor shape from parameters on the stack and then calls BIOS. The cursor lines are numbered starting with 0 at the top of the character block. If the starting line is below the ending line, the cursor will wrap around to the top of the character block. Do not use numbers over 15. There are 8 lines for CGA and 14 for monochrome and EGA.
- *Change_Directory* : Sets a new path and then returns the present defaults.
- *Change_Drive* : Sets a new drive and then returns the present defaults.
- *Change_Screen* : Changes the screen from text to graphics mode or back again. It also clears the screen to a blue background and hides the cursor for text screens.
- *Character_To_Graphic* : Copies a character to the graphic screen. The colour of the character can be set. The location to which the character is written is determined from coordinates passed as parameters. The Ada screen coordinates are converted to those used by BIOS.
- *Character_To_Window_Buffer* : Copies a character with it's attribute a number of times to the Window buffer. The actual location for the characters in the buffer is set by passing coordinates. If the coordinates are out of range, the characters are not written to the buffer. If the number of characters is longer than the text line of the window buffer, they are continued on the next line. The characters are kept within the text area. If the number of characters is more than what can be accommodated by the text area, even after wrapping, they is truncated at the end of the text area.
- *Check_For_Mouse* : Checks if a mouse is installed, and if so, gets the number of buttons and

sets the mouse pointer to the centre of the screen, using the default mouse pointer and movement ratios. The mouse text pointer is not displayed. The MOUSE_STATUS flag is set to 1 if a mouse is present, 0 if it is not.

- *Clear_Main_Buffer* : Clears the buffer for the main screen. The attributes of the characters in the buffer can be specified.
- *Clear_Screen* : Clears a specified part of the screen.
- *Clear_Window_Buffer* : Clears the window buffer for pop-up screens. The attributes of the characters in the buffer can be specified. The size of the buffer that was set by *BufferSetup* specifies the area that is cleared.
- *Determine_Left_Corner* : Calculates the top left corner where the pop-up screen will be displayed. The calculated value is returned in BX and is the offset that is added to the base of screen memory to get the required location on the screen.
- *Dot_Horizontal_Line* : Draws a dotted horizontal line starting at (X1, Y1) to (X2, Y1) in the present foreground colour.
- *Dot_Vertical_Line* : Draws a dotted vertical line starting at (X1, Y1) to (X1, Y2) in the present foreground colour.
- *Find_First_File* : Finds the first matching file to PATH1 or PATH2 and then returns the filename. An error value is passed back also.
- *Find_Next_File* : Finds the next matching file after first calling the *Find_First_File* procedure. The filename and an error value are passed back.
- *Frame_Window_Buffer* : Frames the pop-up screen. A string of 7 graphic characters which are used to border the pop-up screen is passed to the procedure. They are in order from 1-7:
    - Top left corner
    - Top side
    - Top right corner
    - Left and right side
    - Bottom left corner
    - Bottom side
    - Bottom right side

  The size of the frame is determined by the size of the buffer. The top and bottom of the frame is next to the top and bottom of the buffer. The left and right sides of the frame are one column spaced from the left and right sides of the buffer. The attributes of the frame characters can also be set.
- *Get_Defaults* : Gets the default drive and directory. It updates search strings used for listing files and returns the default path in the passed string parameter.
- *Get_Filename* : Returns the filename found in the PSP. An error value is passed back which corresponds to the success of the previous search.
- *Get_Drive_Directory* : Gets and returns the present drive and directory.
- *Get_Keyboard_Character* : Gets a keystroke from the keyboard, and waits until a key is pressed if a keystroke is not available when first called. If the key is an extended key, then the high byte contains the keycode while the low byte is 0. If the key is a normal key, then the high byte is 0 and the low byte contains the keycode.

- *Hide_Cursor* : Hides the text cursor. No parameters are passed by the Ada subprogram. The shape of the cursor is preserved.
- *Hide_Text_Mouse* : Hides the mouse text cursor. No parameters are passed by the Ada subprogram. If no mouse is present, (MOUSE_STATUS is not equal to 1), the subprogram exits.
- *KeyBoard_Character_Ready* : Determines if there is a character in the keyboard buffer - has a key been pressed.
- *Line* : Draws a line starting at (X1, Y1) to (X2, Y2) in the present foreground colour.
- *Main_Buffer_Line* : Clears the buffer for the main screen. The attributes of the characters in the buffer can be specified.
- *Main_Buffer_To_Screen* : Copies the characters and their attributes from the main screen buffer to the screen. Anything on top of the main screen disappears.
- *Make_Line_Bar* : Draws a line on the screen by changing the attributes of characters to the specified attribute value. The attributes of the characters (that are changed) are first stored in a line buffer.
- *Make_Shadow* : Draws a shadow for the pop-up screen by changing the attributes of characters to the right (2 columns) and bottom (1 row) to light_grey on black. The size of the shadow is determined from the size of the buffer for the pop-up screen. The attributes of the characters that are changed do not have to be saved.
- *Main_Title* : Displays ELECTOR on the screen. Two lines are written to the screen at a time. The top part is scrolled down, the bottom part is scroiled up. There is a short delay before each two lines of the display are written.
- *MouseVisible* : All subroutines that write to the screen must check to see if the mouse pointer is active. If the mouse pointer is active, it is turned off before writing to the screen.
- *MouseVisibleAgain* : All subroutines that write to the screen must check to see if the mouse pointer is active. If the mouse pointer was active, but *MouseVisible* turned it off, it is turned on again.
- *New_Window_Location* : Gets the new location where the window is going to be drawn.
- *Plot* : Plots a pixel in the present plot colour on the screen at the position specified. It calls *Basic_Plot* to actually do the plotting.
- *Plot_Four_Pixels* : Plots four pixels about a centre point in the present fore-ground colour at the specified position.
- *Read_Text_Mouse* : Reads the mouse text pointer position. Gets the mouse text pointer position from the MOUSE driver. The coordinates are adjusted to the Ada coordinates. The coordinates are put onto the stack as return parameters. The buttons pressed on the mouse are also returned. If no mouse is present, (MOUSE_STATUS not equal to 1), the subprogram exits.
- *Remove_Line_Bar* : Clears the line from the screen by copying the attributes from the line buffer back to those screen characters affected by the line.
- *Screen_Setup* : Determines if a VGA, EGA, CGA or MDA is installed. If MDA is installed, an error message is displayed and the program is terminated. If any of the other 3 adapters are installed, the screen mode and the resolution for X and Y are determined.
- *Set_Cursor* : Sets the cursor position. It gets the new cursor position from parameters on the stack, decrements both the row and column values to convert from the Ada screen coordinates

to those used by BIOS, and then calls BIOS.

- *Set_Plot_Colour* : Sets the plot colour used when drawing lines, plotting 4 points, etc.
- *Set_Plot_Window* : Sets the window where a dot can be plotted.
- *Show_Cursor* : Shows the text cursor. No parameters are passed by the Ada subprogram. The shape of the cursor is preserved.
- *Show_Text_Mouse* ; Shows the mouse text pointer. No parameters are passed by the Ada subprogram.
- *String_To_Graphic* : Copies a string to the graphic screen. The colour of the characters can be set. The location to which the string is written is determined from coordinates passed as parameters. These Ada screen coordinates are converted to those used by BIOS.
- *String_To_Main_Buffer* : Copies a string to the buffer of the main screen. The attributes of the string characters can be set to either of two specified sets. The character ~ is used as a special selector character (this can be changed very easily) which will switch the attributes to the second set if the first are active, or back to the first if the second are active. The location for the string in the buffer is set by passing coordinates. If the coordinates are out of range, the string is not written to the buffer. If the string is longer than the text line of the main screen, it is wrapped around to the next line. If the string is longer than what can be accommodated, even after wrapping, it is truncated at the end of the main buffer.
- *String_To_Window_Buffer* : Copies a string to the Window buffer for the pop-up screen. The attributes of the string characters can be set to either of two specified sets. The character ~ is used as a special selector character (this can be changed very easily) which will switch the attributes to the second set if the first are active, or back to the first if the second are active. The location for the string in the buffer is set by passing coordinates. If the coordinates are out of range, the string is not written to the buffer. If the string is longer than the text line of the pop-up screen, it is wrapped around. String characters are kept within the text area. If the string is longer than what can be accommodated by the text area, even after wrapping, it is truncated at the end of the text area.
- *Tutorial_To_Screen* : This copies the tutorial image read from disk to the screen.
- *WaitSomeTime* : Waits for a certain amount of time.
- *Window_Buffer_Setup* : Sets up the size of the pop-up screen that will be displayed on the screen. The width is in columns and the height is in rows. Note that this is the full size of the pop-up screen. The size of the area for text display within the pop-up screen is smaller.
- *Window_Buffer_Title* : Copies a string to the Window buffer for the pop-up screen. The attributes of the string characters can be set.
- *Window_Buffer_To_Screen* : Copies the characters and their attributes from the Window buffer to the screen. In other words, the pop-up screen pops up. The number of characters copied depends on the size of the Window buffer set by *BufferSetup*.

CHAPTER 8

SCREEN OBJECTS

8.1 Introduction

The user interface is constructed from a number small objects. The smallest objects are called Screen Objects and are found in the Screen_Objects package. These objects form the foundation on which the entire user interface is build. There are six types of objects :

- Buttons
- Boolean Buttons
- Input Lines
- Pick_Lists (Dynamic and Static)
- Forms
- Windows

The windows object forms the base on which the other screen objects appear. Local coordinates are used by all the screen objects except the window object.

There is a seventh object - Inputter object. It does not appear on the screen, but it gets all the input from the user which is directed to the Screen Objects. It is found in a separate Inputter package.

8.2 Coordinates

There are two forms of coordinates that the screen objects use.

- *Global* : This corresponds to the physical screen coordinates and is used by the Windows, the Input_Lines and Forms objects. The Input_Lines and Forms need global coordinates to set the hardware cursor which they use.
- *Local* : This corresponds to the display area of the Window object. All the screen objects that are displayed, except the Window object, use this coordinate system. Local coordinates are calculated as follows :

```
Local Row    = Global Row    - Window.ROW
Local Column = Global Column - Window.COLUMN - 1
```

8.3 Inputter Object

8.3.1 Purpose

It gets input from the keyboard and the mouse.

8.3.2 Appearance and Behaviour

The keyboard value is put into the KEY field if a key was pressed. The OPERATION field is used for communication between the different screen objects. The four remaining fields are set to the current mouse status if a mouse active. Otherwise they are set to the top left corner of the screen with no button pressed.

8.3.3 Components

```
type INPUT_TYPE is
   record
      GOT_MOUSE : BOOLEAN;
      BUTTON    : MOUSE_BUTTONS_TYPE;
      Y         : INTEGER;
      X         : INTEGER;
      KEY       : KEY_VALUES_TYPE;
      OPERATION : OPERATION_TYPE;
   end record;
```

- *GOT_MOUSE* : If this flag is true, then a mouse is active and it's status has been examined. If the flag is false, no mouse was found.
- *BUTTON* : The button that was pressed on the mouse. On a two button mouse, only the left and right buttons are returned here. On a three button mouse, the left, middle and right buttons are returned. If no button was pressed, no button pressed value is returned.
- *X* : The column coordinate of the mouse pointer when it was examined. This ranges from 1 to 80 for a 80 column screen.
- *Y* : The row coordinates of the mouse pointer when it was examined. This ranges from 1 to 25 for a 25 row screen.
- *OPERATION* : Used for communication between the different screen objects. This field is set to OP_DO when the input devices are read. As the input record is passed from one screen object to another, this field may be changed.

8.3.4 Methods

Set_Up

Checks if a mouse is available, and if so, resets the mouse.

```
begin
  Check if a mouse is available, and if so, reset the mouse.
end Set_Up;
```

Get_Input

Checks if a mouse is active, and if so, examines it's status. The keyboard is checked to see if a key was pressed, and if so, gets the key value.

```
begin
  Set the GOT_MOUSE field of the input record to the status of the mouse being active or not.
  if (the mouse is active) then
    Read the location of the mouse cursor and check if any buttons have been pressed. Set the
      X, Y and BUTTON fields of the input record to these values.
  else
    Set the X, Y and BUTTON fields of the input record to the top left corner with no button
      pressed.
  end if;
  if (a key has been pressed) then
    Get the key from the keyboard and set the KEY field of the input record to it.
  else
    Set the KEY field of the input record to no key pressed.
  end if;
  Set the OPERATION field of the input record to 'operation to be done'.
end Get_Input;
```

8.4 Buttons

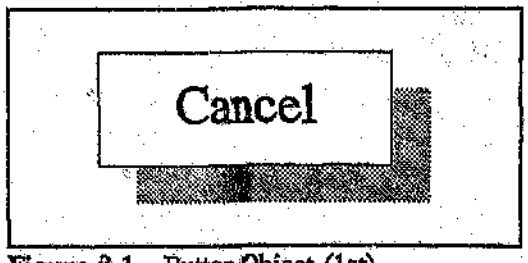

Figure 8-1 Button Object (1st)

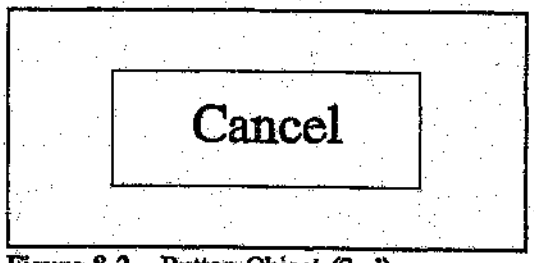

Figure 8-2 Button Object (2nd)

8.4.1 Purpose

It initiates some action. The action can be considered to be connected to the pressing of the button.

8.4.2 Appearance and Behaviour

A button can have two different appearances - depending on the chosen draw method.

- *Appearance 1* : A button is a bar that appears on the screen and has a shadow. When the button is *pressed*, the button appears as though it is being pushed into the screen. This is done by moving the surface of the button one column to the right and removing the shadow. The button is *released* about half a second later and appears like it did before being pressed - the button surface in it's original position with a shadow.
- *Appearance 2* : A button is a bar that appears on the screen and does not have a shadow. When the button is *pressed*, the button changes colour. The button is *released* about half a second later and appears like it did before being pressed - in it's original colour.

The button can be *pressed* in two ways :

- Pressing the Alt key with the character that is highlighted on the button's surface. In the above example, this would be Alt - P.
- Pressing the left button of the mouse when the mouse pointer is on top of the button. This means that the mouse pointer can be anywhere of the button's surface, but not on the shadow, for the button to be pressed.

8.4.3 Components

```
type BUTTON_TYPE is
   record
      NAME_LENGTH      : NATURAL;
      NAME             : STRING (1..15);
      OPERATION        : OPERATION_TYPE;
      COLOURS          : SCREEN_COLOUR_TYPE;
      SELECT_COLOURS   : SCREEN_COLOUR_TYPE;
      UNDER_COLOURS    : SCREEN_COLOUR_TYPE;
      ROW              : ROW_TYPE;
      COLUMN           : COLUMN_TYPE;
      HOT_KEY          : KEY_VALUES_TYPE;
   end record;
```

- *NAME_LENGTH* : The length of the name string. It cannot be larger than the length of the actual string for the name.
- *NAME* : The name of the button that is displayed on its surface. The letter that is highlighted is put between the character "~". The first ~ switches on the highlighting, and the second ~ switches it off. There must be two of these characters for each button name. If the highlighting is not wanted, the select colour must be set to the text colour.
- *OPERATION* : The operation that corresponds to this button being pressed. This operation is passed in the input record.
- *COLOURS* : The colour that the button surface is drawn in.
- *SELECT_COLOURS* : The highlighting colour of the character that is used to select this

button.

- *UNDER_COLOURS* : In the first method, this is the colour of the surface under the button. It is usually the same colour as the window object. In the second method, this is the colour that the button will become when it is pressed.
- *ROW* : The row on which the button is drawn. These are local coordinates.
- *COLUMN* : The left most column on which the button is drawn. These are in local coordinates.
- *HOT_KEY* : The code of the key that will cause this button to be *pressed* if the corresponding key on the keyboard is pressed. The code of the key is the BIOS code of the Alt key pressed in combination with another character.

8.4.4 Methods

Create

Sets initial values for the appearance and behaviour of the button.

```
begin
  Initialise the fields of the button data structure to the values that were passed in.
  Create;
```

Draw

Draws a button in the *normal* or *depressed* state.

```
begin
  Draw the top surface of the button. This is drawn in the buffer.
  For the first method, draw the shadow of the button. This is drawn in the buffer.
end Draw;
```

Press

Draws a button in the *pressed* state.

```
begin
  For the first method, draw the top surface of the button that has moved one column to the
    right. For the second method, draw the top surface of the button in the same position
    but in another colour. This is drawn in the buffer.
  For the first method, remove the shadow of the button by overwriting the area with spaces.
  This is drawn in the buffer.
```

```
end Press;
```

Handle_Input

Checks if the mouse pointer is on the button or the hotkey has been pressed.

```
begin
  if (the mouse is active and the left mouse button was pressed) then
    if (the mouse pointer is on the top surface of the button) then
      Set a flag to indicate that the button has been pressed.
    end if;
  elsif (any key pressed is this button's hot key) then
    Set a flag to indicate that the button has been pressed.
  end if;
  if (the flag for the button being pressed is set) then
    Draw the button being pressed, update the screen, and wait for a short delay. Then restore
      the button's state, update the screen, and wait for a short delay.
    Set the OPERATION field of the input record to the operation that this button represents.
  end if;
end Handle_Input;
```

8.5 Boolean_Buttons

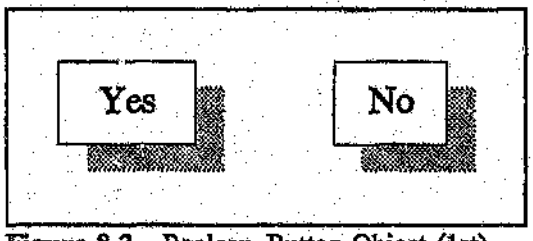

Figure 8-3 Boolean_Button Object (1st)

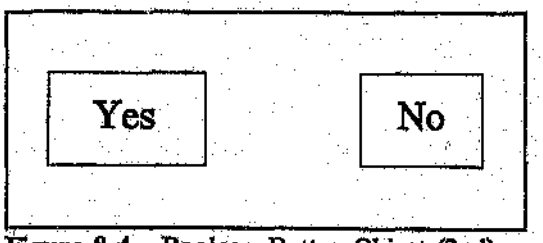

Figure 8-4 Boolean_Button Object (2nd)

8.5.1 Purpose

To get a Boolean input response from the user. In other words, a yes/no, true/false type response.

8.5.2 Appearance and Behaviour

This object is made up from two button objects - a *True* button and a *False* button. The description and behaviour of the two buttons is discussed in the Buttons section.

8.5.3 Components

```
type BOOLEAN_BUTTON_TYPE is
   record
      BT_TRUE  : Buttons.BUTTON_TYPE;
      BT_FALSE : Buttons.BUTTON_TYPE;
   end record;
```

- *BT_TRUE* : Holds the *True* button.
- *BT_FALSE* : Holds the *False* button.

8.5.4 Methods

Create

Sets one button as a *True* button and the other as a *False* button.

begin
  Initialize the fields of the two button data structure to the values that were passed in.
end Create;

Draw

Draws the two buttons.

begin
  Draw the top surface and the shadow of the True button. This is drawn in the buffer.
  Draw the top surface and the shadow of the False button. This is drawn in the buffer.
end Draw;

Handle_Input

Checks if the mouse pointer is on the *True* or *False* button when the left button on the mouse is pressed, or if Alt-T or Alt-F has been pressed on the keyboard.

begin
  Route the input record to the True button's input handler.
  if (the operations field of the input record has not been set) **then**
    Route the input record to the False button's input handler.
  end if;
end Handle_Input;

8.6 Input_Lines

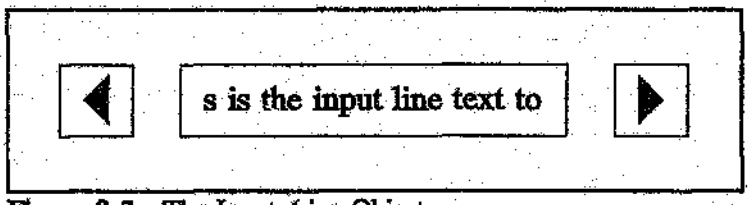

Figure 8-5 The Input_Line Object

8.6.1 Purpose

It gets string input. String input includes integer and real numbers which are entered as strings and then converted.

8.6.2 Appearance and Behaviour

The Input_Line object appears on the screen as three bars in a line. The centre bar is the longest and this is where characters of the string are entered. However, this centre bar can only hold a certain number of characters.

Key	Function of the key
A - Z	Only for string input (E for real input as well).
a - z	Only for string input (e for real input as well).
\/_['^]	Only for string input.
0 - 9	For integer, real and string input.
-	For integer, real and string input.
.eE	For real and string input.
Cursor Left	Move the cursor to the left, or if the cursor is at the left most position, scroll the text to the right if there is a left arrow.
Cursor Right	Move the cursor to the right, or if the cursor is at the right most position, scroll the text to the left if there is a right arrow.
Insert	Toggle between insert and typeover mode. The shape of the cursor will change to a thin line in insert mode, and to block in typeover mode.
Delete	Deletes the character the cursor is on.
Backspace	Deletes the character to the left of the cursor.

Table 8-1 Input_Line Key Definition

If more characters are entered than can be displayed, they will be scrolled off to the left. A left arrow will appear in the first bar to show that there are characters to the left of those displayed, but because of the limited length of the display area, they cannot be displayed. The cursor can be moved to the edges of the display bar to scroll a string that is longer than the display bar to the left or right. If the string is being scrolled to the right and the right most characters disappear, a right arrow will appear to show that there are characters to the right of those displayed. There are two modes of character entry :

- *Insert* : Characters are inserted into the string at the position of the cursor. The characters to the right of the cursor are moved one place to the right before the character is inserted. The cursor appears as a flashing thin line.
- *Typeover* : Characters are inserted into the string at the position of the cursor. The character on which the cursor is flashing is overwritten by the new character. Characters to the right of the cursor remain in their present position. The cursor appears as a flashing block.

8.6.3 Components

```
type LINE_TYPE is
   record
      MAX_LENGTH       : NATURAL;
      LINE             : STRING (1..64);
      ACTUAL_LENGTH    : POSITIVE    := 1;
      DISPLAY_LENGTH   : POSITIVE;
      START_DISPLAY    : POSITIVE    := 1;
      CURSOR           : POSITIVE    := 1;
      INSERTING        : BOOLEAN     := TRUE;
      ROW              : ROW_TYPE;
      COLUMN           : COLUMN_TYPE;
      COLOUR           : SCREEN_COLOUR_TYPE;
   end record;
```

- *MAX_LENGTH* : The maximum length of the input line. It cannot be larger than the length of the string for the line.
- *LINE* : The string into which characters are put. The length of this string is the number of characters that can be entered.
- *ACTUAL_LENGTH* : The number of characters that have been actually entered into the Input_Line object.
- *DISPLAY_LENGTH* : The length of the display bar on which the characters that have been entered are displayed.
- *START_DISPLAY* : The starting place in the string from which characters will be displayed.
- *CURSOR* : The cursor's position in the LINE string. It is here where the next character will be entered. This value is also used to determine if the hardware cursor should be adjusted or the displayed string scrolled.
- *INSERTING* : A flag that indicates if the insert mode is active or inactive (typeover mode). The shape of the cursor is also affected by the insert mode being active or inactive.

- *ROW* : The row on which the Input_Line object is drawn. These are local coordinates.
- *COLUMN* : The left most column on which the first bar of the Input_Line object is drawn. These are in local coordinates.
- *COLOUR* : The colour that the Input_Line's surface is drawn in.

8.6.4 Methods

Create

Sets initial values for the appearance and behaviour of the Input_Line.

```
begin
  Initialise the fields of the input_line data structure to the values that were passed in.
  Offset values for the text cursor are also initialized.
end Create;
```

Draw

Draws the Input_Line object and sets the hardware cursor to the place where the next character will be entered.

```
begin
  Draw part of the line with characters from the LINE field of the input_line data structure.
    The size of the line is specified by the DISPLAY_LENGTH field. The line starts with the
    first character from the LINE field pointed to by START_DISPLAY. All drawing is done
    in a buffer.
  if (there are characters to the left of the line that are not visible) then
    Draw an arrow that points to the left on the left side of the display line.
  end if;
  if (there are characters to the right of the line that are not visible) then
    Draw an arrow that points to the right on the right side of the display line.
  end if;
  Set the hardware cursor to point to the place where the next character will be inserted into
    the line.
end Draw;
```

Get_Line

Returns a string from the Input_Line object.

```
begin
  Return the string of characters from the Input_Line data structure.
```

```
end Get_Line;
```

Handle_Input

Checks if the key pressed on the keyboard is in a predefined range (see table), and if so, enters the value into the string. If it is not, a check is made to see if it is an edit key. If so, the corresponding editing function is undertaken.

Move_Cursor_Left

```
begin
  if (the cursor is not pointing to the first character of the LINE field of the Input_Line) then
    Adjust the cursor by decrementing it so that it points to the previous character.
    if (the cursor is <= the start of the display) then
      Adjust the start of the display to point to the same character as the cursor.
    end if;
  end if;
end Move_Cursor_Left;
```

Move_Cursor_Right

```
begin
  if (the cursor is not pointing to the last character of the LINE field of the Input_Line) then
    Adjust the cursor by incrementing it so that it points to the next character.
    if (the cursor minus the start of the display is >= display length of the line) then
      Adjust the start of the display.
    end if;
  end if;
end Move_Cursor_Right;
```

```
begin
  if (the OPERATION field of the input record is set to OP_MOVED) then
    Update the cursor offset values. The window object has moved and so the Input_Line display
      is no longer at the same place it was when initialized.
    Set the redraw flag to redraw the Input_Line.
  end if;
  case (of the KEY field of the input record) is
    when (the cursor left arrow key) =>
      Move the cursor to the left.
    when (the cursor right arrow key) =>
      Move the cursor to the right.
    when (the backspace key) =>
      if (the cursor is not pointing to the first character of the LINE field of the Input_Line) then
        Move all the characters from the cursor to the end of the line, down one location. The
          pointer for the actual number of characters on the line must be decremented by one.
        Move the cursor to the left.
      end if;
    when (the delete key) =>
      Move all the characters from the character beyond the cursor to the end of the line,
```

```
        down one location.
      if (the cursor < actual length of the line) then
        Adjust the actual length of the line by decrementing it.
      end if;
    when (the insert key) =>
      Toggle the inserting mode flag.
      if (the inserting flag is set) then
        Change the shape of the hardware cursor to a thin line.
      else
        Change the shape of the hardware cursor to a solid block.
      end if;
    when (the key is an uppercase letter, or lowercase letter, or a number, or a valid character) =>
      if (the actual length of the line is not equal to the maximum length of the line) then
        if (the cursor = the actual length of the line) then
          Adjust the actual length of the line by incrementing by 1.
        else
          if (the inserting flag is set) then
            Move all the characters from the cursor to the end of the line, up one location.
            Adjust the actual length of the line by incrementing by 1.
          end if;
        end if;
        Insert the new character into the line.
        Move the cursor to the right.
      elsif (the cursor /= maximum length of the line and typeover mode) then
        Put the new character into the line.
        Move the cursor to the right.
      end if;
    when others =>
      Clear the redraw flag.
  end case;
  if (the redraw flag is set) then
    Draw the Input_Line and update the screen.
  end if;
end Handle_Input;
```

Reset_Line

Resets the Input_Line so that a clear line is ready for input.

```
begin
  Reset the Input_Line so that it is a clear line ready for input.
end Reset_Line;
```

8.7 Pick_Lists (Static)

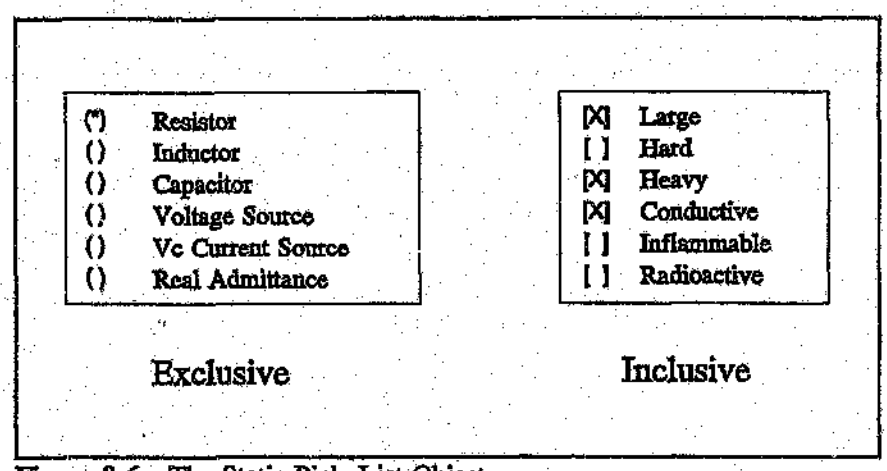

Figure 8-6 The Static Pick_List Object

8.7.1 Purpose

The Pick_List object is used when list-type input is needed. The list can be either exclusive (only one option can be chosen) or inclusive (many options may be chosen simultaneously).

8.7.2 Appearance and Behaviour

The Pick_List appears as a large flat area on the screen. On the left of this area are radio buttons if there is an exclusive list, or switches if there is an inclusive list. Text for each option of the list is written in the remaining area.

Characteristics of Pick_Lists	Exclusive	Inclusive
Option is not chosen	( )	[ ]
Option has been chosen	(*)	[X]

Table 8-2 Pick_List Characteristics

Options can be chosen by the mouse or the keyboard. To use the mouse, just move the mouse pointer to the desired option and press the left button. To use the keyboard, move the selection bar until it highlights the desired option. If the list is inclusive, select or de-select the highlighted option by pressing the space bar. When choosing an option, the behaviour of the Pick_List object will depend on the type of list presented.

- If the list is exclusive, choosing a different option to one already set, will deselect the previously option and select the new option.
- If the list is inclusive, choosing an option will toggle it's state. That means that if the option

is selected, it will be deselected, and vice versa.

Key pressed	Affect on the Pick_List object
Cursor up	Moves the selection bar up to the previous option.
Cursor down	Moves the selection bar down to the next option.
Home	Moves the selection bar to the first option.
End	Moves the selection bar to the last option.
Space	Selects the option that the selection bar is highlighting for inclusive lists.

Table 8-3 Pick_List (Static) Key Definition

8.7.3 Components

```
type LIST_TYPE is
   record
      LENGTHS      : NATURAL;
      EXCLUSIVE    : BOOLEAN;
      ACTIVE       : BOOLEAN;
      WIDTHS       : POSITIVE;
      CHOICE       : POSITIVE   := 1;
      TOP_ROW      : ROW_TYPE;
      LEFT_SIDE    : COLUMN_TYPE;
      COLOUR       : SCREEN_COLOUR_TYPE;
      BAR_COLOUR   : SCREEN_COLOUR_TYPE;
   end record;
```

- *LENGTHS* : The length of the list.
- *EXCLUSIVE* : A flag which indicates if the list is exclusive or not (inclusive).
- *ACTIVE* : A flag which indicates if the Pick_List is active and will accept input.
- *WIDTHS* : The width of the area that displays the text for each option. The widths must be longer than the longest text string for any option.
- *CHOICE* : The option on which the selection bar is currently positioned.
- *TOP_ROW* : The highest row on which the Pick_List object is drawn. These are local coordinates.
- *LEFT_SIDE* : The left most column on which the Pick_List object is drawn. These are in local coordinates.
- *COLOUR* : The colour of the Pick_List's surface.
- *BAR_COLOUR* : The colour of the selection bar.

8.7.4 Methods

Create

Sets initial values for the appearance and behaviour of the Pick_List.

```
begin
  Initialise the fields of the pick_list data structure to the values that were passed in.
  if (the list is an exclusive list) then
    for (the COUNTER from 1 to the length of the list) loop
      if (the option indexed is set) then
        Set the present choice to this option.
      end if;
    end loop;
  end if;
end Create;
```

Draw

Draws the Pick_List object and places the selection bar on the option held by CHOICE.

```
begin
  for (the COUNTER from 1 to the length of the list) loop
    if (the option indexed by COUNTER is true) then
      if (this list is exclusive) then
        Draw a radio button which is set.
      else
        Draw a switch which is set.
      end if;
    else
      if (this list is exclusive) then
        Draw a radio button which is not set.
      else
        Draw a switch which is not set.
      end if;
    end if;
    if (the COUNTER = the present choice and pick_list is active) then
      Adjust the colour to that of the bar.
    else
      Adjust the colour to that of the normal list.
    end if;
    Draw the text line for an option of the list. All drawing is done in the buffer.
  end loop;
end Draw;
```

Handle_Input

Checks to see if the mouse pointer was on an option when the left button of the mouse was pressed. If so, it selects the chosen option. A check is also made to see if one of the predefined selection keys was chosen (see Table 8.3) and if so, the corresponding selection action is undertaken.

```
begin
  if (the pick_list object is not active) then
    if (the OPERATION field of the input record is set to OP_NON_ACCEPTING) then
      Make the pick_list object active.
      Set the OPERATION field of the input record to OP_DONE.
    else
      Return back to the calling routine.
    end if;
  elsif (the mouse is active and the left button was pressed and it has just been pressed) then
    if (the mouse pointer is in the pick_list display area) then
      Calculate the desired choice.
      if (this list is inclusive) then
        Toggle the state of the option indexed by the choice value.
      end if;
    else
      Clear the redraw flag.
    end if;
    Clear the flag which tells if a mouse button has just been pressed.
  else
    case (of the KEY field of the input record) is
      when (the tab key) =>
        Set the pick_list object to be non-active.
        Set the OPERATION field of the input record to OP_NON_ACCEPTING.
      when (the space key) =>
        if (this list is exclusive) then
          Clear all the states of the list.
          Set the state of the option indexed by the choice value.
        else
          Toggle the state of the option indexed by the choice value.
        end if;
      when (the cursor up arrow key) =>
        if (the present choice is the first option) then
          Set the present choice to the last option.
        else
          Set the present choice to the previous option.
        end if;
      when (the cursor down arrow key) =>
        if (the present choice is the last option) then
          Set the present choice to the first option.
        else
          Set the present choice to the next option.
        end if;
      when (the home key) =>
        Set the present choice to the first option.
      when (the end key) =>
        Set the present choice to the last option.
      when others =>
        Clear the redraw flag.
    end case;
    if (the mouse left button is not pressed) then
      Clear the flag which tells if a mouse button has just been pressed.
    end if;
  end if;
  if (the redraw flag is set) then
    if (this list is exclusive) then
      Clear all the states of the list.
```

```
        Set the state of the option indexed by the choice value.
      end if;
      Draw the pick_list and update the screen.
      Set the OPERATION field of the input record = OP_DONE.
    end if;
end Handle_Input;
```

8.8 Pick_Lists (Dynamic)

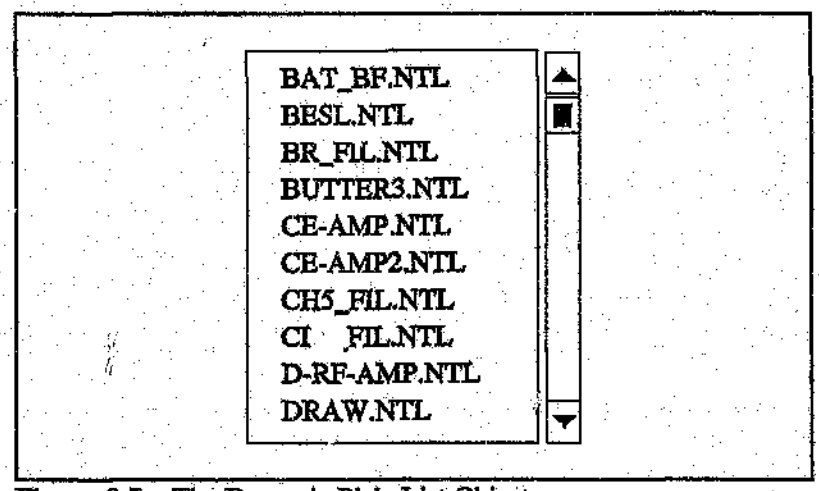

Figure 8-7 The Dynamic Pick_List Object

8.8.1 Purpose

The dynamic Pick_List object is used when lists of different lengths (dynamic) need to be displayed from which one option must be chosen. This object is usually used for choosing a filename from a list.

8.8.2 Appearance and Behaviour

The dynamic Pick_List appears as a large flat area with a possible scroll bar to the right. On the flat area is a text list of available options. If there are more options than can be displayed, a scroll bar appears. There is an up and a down arrow. An indicator shows what part of the total list is visible. The highlighting bar can be moved up or down to the desired choice by pressing the cursor up or down keys respectively.

The mouse can also be used to choose an option. By placing the mouse pointer on the list and pressing the left mouse button, the option that the mouse pointer is on will be highlighted and selected. Placing the mouse pointer on the up arrow of the scroll bar and pressing the left mouse

button moves the highlighting bar to the previous option if there is one. Placing the mouse pointer on the down arrow of the scroll bar and pressing the left mouse button moves the highlighting bar to the next option if there is one.

8.8.3 Components

The dynamic list is a doubly-linked list with both forward and backward pointers.

```
type LIST_DYNAMIC_TYPE;

type NAME_LINK is access LIST_DYNAMIC_TYPE;

type LIST_DYNAMIC_TYPE is
   record
      NAME   : STRING (1..12);
      NEXT   : NAME_LINK;
      BACK   : NAME_LINK;
   end record;

type LIST_D_TYPE is
   record
      START              : NAME_LINK;
      PRESENT_TOP        : NAME_LINK;
      CHOICE_POINTER     : NAME_LINK;
      MAX_LENGTH         : NATURAL;
      PRESENT_LENGTH     : NATURAL;
      LENGTHS            : NATURAL;
      CHOICE             : POSITIVE;
      TOP_ROW            : ROW_TYPE;
      LEFT_SIDE          : COLUMN_TYPE;
      COLOUR             : SCREEN_COLOUR_TYPE;
      BAR_COLOUR         : SCREEN_COLOUR_TYPE;
      SCROLLER_COLOUR    : SCREEN_COLOUR_TYPE;
   end record;
```

- *START* : Points to the first option on the list. It is the start of the list.
- *PRESENT_TOP* : Points to the first option that is displayed on the screen.
- *CHOICE_POINTER* : Points to the current option in the list that is highlighted.
- *MAX_LENGTH* : The maximum length of the list and is used by the scroll bar to determine when the scroll bar indicator must move.
- *PRESENT_LENGTH* : The index value of the highlighted choice within the list. It is used by the scroll bar to determine when the scroll bar indicator must move.
- *LENGTHS* : Length of the list that will be displayed.
- *WIDTHS* : The width of the area that displays the text for each option. The widths must be longer than the longest text string for any option.
- *CHOICE* : The option on which the selection bar is currently positioned.
- *TOP_ROW* : The first row on which the Pick_List object is drawn. These are local coordinates.
- *LEFT_SIDE* : The left most column on which the Pick_List object is drawn. These are in

local coordinates.

- *COLOUR* : The colour of the Pick_List's surface.
- *BAR_COLOUR* : The colour of the selection bar.
- *SCROLLER_COLOUR* : The colour of the scroll bar.

8.8.4 Methods

Create

Sets initial values for the appearance and behaviour of the Pick_List.

```
begin
  Initialise the fields of the pick_list data structure to the values that were passed in.
end Create;
```

Create_List

Sets all the data for the list of the dynamic Pick_List object.

```
begin
  Create a new option for the list and set it's text field to the input text.
  if (this is not the first option) then
    Link this option to the first option.
  end if;
  Set the relevant pointers to this new option.
end Create_List;
```

Destroy_List

Destroys all the options of the dynamic list that is used by the dynamic Pick_List.

```
begin
  while (there is still an option) loop
    Destroy the that option
  end loop;
end Destroy_List;
```

Draw

Draws the dynamic Pick_List object into the window.

```
begin
  for (the COUNTER from 1 to the number of options that can be displayed) loop
    if (the COUNTER = the present choice) then
      Set the colour to that of the bar.
    else
      Set the colour to the normal list.
    end if;
    if (there is an option) then
      Get the options text.
    else
      Set the options text to a blank line.
    end if;
    Draw the text line for the option of the list.
  end loop;
  if (there are more options than can be displayed) then
    Draw the up arrow of the scroll bar.
    for (the COUNTER from 1 to display length of the scroll bar) loop
      if (the COUNTER = indicator position) then
        Set drawing character to indicator.
      else
        Set drawing character to background.
      end if;
      Draw side bar of the scroll bar using drawing character.
    end loop;
    Draw the down arrow of the scroll bar.
  end if;
end Draw;
```

Handle_Input

Checks to see if the mouse pointer was on an option when the left mouse button was pressed. If so, it selects the option. If the mouse pointer is on the up arrow or down arrow of the scroll bar, the list is scrolled up or down.

Move_Up

```
begin
  if (the present choice is not the first displayed option) then
    Set the present choice to the previous displayed option.
    Update the scroll bar index by decrementing by 1.
  elsif (the first option has not been reached) then
    Scroll all the options down by one.
    Update the scroll bar index by decrementing by 1.
  end if;
end Move_Up;
```

Move_Down

```
begin
  if (there is another option) then
```

```
      if (the present choice is not the last displayed option) then
        Set the present choice to the next displayed option.
      else
        Scroll all the options up by one.
      end if;
      Update the scroll bar index by incrementing by 1.
    end if;
  end Move_Down;

begin
  if (the mouse is active and the left button has was pressed) then
    if (the mouse pointer is in the pick_lists display area) then
      Calculate the present choice index.
      for (the COUNTER from 1 to the choice index - 1) loop
        if (there is another option) then
          Set the CHOICE_POINTER field to this option.
        else
          Set the choice index to COUNTER.
          Exit from this loop.
        end if;
      end loop;
      Calculate the new position for the scroll bar.
    elsif (the mouse pointer is on the scroll bar up arrow) then
      Move to the previous option.
    elsif (the mouse pointer is on the scroll bar down arrow) then
      Move to the next option.
    else
      Clear the redraw flag.
    end if;
    Delay for about 0.1 seconds.
  else
    case (of the KEY field of the input record) is
      when (the cursor down key) =>
        Move to the next option.
      when (the cursor up key) =>
        Move to the previous option.
      when (the enter key) =>
        Set the OPERATIONS field of the input record to OP_OK.
      when others =>
        Clear the redraw flag.
    end case;
  end if;
  if (the redraw flag is set) then
    Draw the pick_list object and update the screen.
    Set the OPERATIONS field of the input record to OP_DONE.
  end if;
end Handle_Input;
```

Get_Name

Returns the chosen name and destroys the list.

```
begin
  Return the chosen filename.
  Destroy the file list.
end Get_Name;
```

8.9 Forms

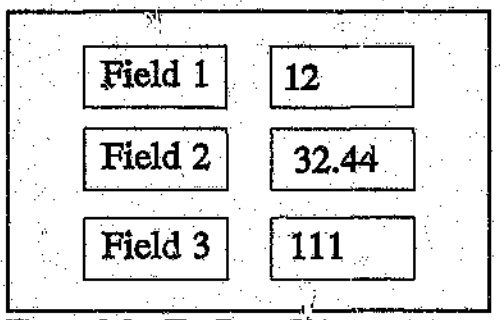

Figure 8-8 The Form Object

8.9.1 Purpose

The form object is used when many related numerical values need to be entered.

8.9.2 Appearance and Behaviour

A text string is displayed to the left of each input field. The size of the input field shows the number of characters that can be entered. Real values and integer values can be entered, but only one type for each input field. Real and integer values that are entered are checked to see that they are not greater than a maximum value, and not less than a minimum value. An integer value can also be checked against another integer value from another input field to see if they are equal.

8.9.3 Exceptions

A number of exceptions can be raised if the entered value fails a test.

- *INTEGER_ERROR* : The value entered is not a recognised integer value.
- *INTEGER_MIN_ERROR* : The integer value entered is less than a defined minimum value.
- *INTEGER_MAX_ERROR* : The integer value entered is greater than a defined maximum value.
- *INTEGER_EQUAL_ERROR* : The integer value entered is equal to an integer value in another

defined field.

- *REAL_ERROR* : The value entered is not a recognised real value.
- *REAL_MIN_ERROR* : The real value entered is less than a defined minimum value.
- *REAL_MAX_ERROR* : The real value entered is greater than a defined maximum value.

Key Pressed	Affect on the Form object
Cursor up	Moves the cursor to the previous field. If the cursor is already in the first field, it moves to the last field.
Cursor down	Moves the cursor to the next field. If the cursor is already in the last field, it moves to the first field.
Cursor left	Moves the cursor to the left by one column. If the cursor is already at the start of the field, the cursor moves to the previous field.
Cursor right	Moves the cursor to the right by one column. If the cursor is already at the of the text in the field, it moves to the next field.
Tab	Makes the form inactive. This prevents the form from accepting any more input. This is used when switching between the form and static Pick_List objects.

Table 8-4 Form Key Definition

8.9.4 Components

```
type FORM_TYPE is
   record
      ACTIVE_FIELD    : POSITIVE;
      CURSOR_ROW      : ROW_TYPE;
      CURSOR_COLUMN   : COLUMN_TYPE;
      TEXT_COLUMN     : COLUMN_TYPE;
      VALUE_COLUMN    : COLUMN_TYPE;
      ACTIVE          : BOOLEAN;
      INSERTING       : BOOLEAN;
      ACTIVE_COLOUR   : SCREEN_COLOUR_TYPE;
      FIELD_COLOUR    : SCREEN_COLOUR_TYPE;
      TEXT_COLOUR     : SCREEN_COLOUR_TYPE;
   end record;
```

- *ACTIVE_FIELD* : The index value of the field that is currently active.
- *CURSOR_ROW* : The hardware cursor offset row coordinate.
- *CURSOR_COLUMN* : The hardware cursor offset column coordinate.
- *TEXT_COLUMN* : The left most column from which the text for the field is drawn. These are in local coordinates.
- *VALUE_COLUMN* : The left most column from which the input field is drawn. These are in local coordinates.
- *ACTIVE* : A flag which indicates if the form is active and accepting input.
- *INSERTING* : A flag which indicates if the form is in inserting mode or not.

- *ACTIVE_COLOUR* : The colour of the active field.
- *FIELD_COLOUR* : The colour of the other non-active fields.
- *TEXT_COLOUR* : The colour of the text for each field.

8.9.5 Methods

Create

Sets initial values for the appearance and behaviour of the Form.

```
begin
  Initialise the fields of the forms object data structure to the values that were passed in.
end Create;
```

Draw

Draws the Forms object into the window.

```
begin
  for (COUNTER from 1 to the number of fields) loop
    if (the field indexed by COUNTER is accepting input) then
      if (this field is the active field and the form object is active) then
        Set the colour to that of the active field colour.
      else
        Set the colour to that of the non-active field colour.
      end if;
      Write the text for the field.
      Write the field.
    end if;
  end loop;
  Set the hardware cursor.
end Draw;
```

Handle_Input

Handles input that is directed to the Form object, otherwise it ignores it.

Process_Field

```
  begin
    Draw the form object.
    if (the type of field is an integer) then
```

```
    begin
      Convert the string to an integer value.
    exception
      when (the string cannot be properly converted) =>
        Raise the exception INTEGER_ERROR.
    end;
    if (the integer value is less than the smallest allowed integer) then
      Raise the exception INTEGER_MIN_ERROR.
    elsif (the integer value is greater than the largest allowed integer) then
      Raise the exception INTEGER_MAX_ERROR.
    elsif (this field must be tested against another field) then
      if (the integer value is equal to the other integer value) then
        Raise the exception INTEGER_EQUAL_ERROR.
      end if;
    end if;
  else
    begin
      Convert the string to a real value.
    exception
      when (the string cannot be properly converted) =>
        Raise the exception REAL_ERROR.
    end;
    if (the real value is less than the smallest allowed real) then
      Raise the exception REAL_MIN_ERROR.
    elsif (the real value is greater than the largest allowed real) then
      Raise the exception REAL_MAX_ERROR.
    end if;
  end if;
end Process_Field;
```

Next_Field

```
begin
  Process the field.
  if (the next field must be moved to) then
    loop
      if (the last field has not been reached) then
        Move to the next field.
      else
        Move to the first field.
      end if;
      Exit from this loop when the field chosen is accepting input.
    end loop;
    Set the cursor column to the first position in the chosen field.
  else
    loop
      if (the first field has not been reached) then
        Move to the previous field.
      else
        Move to the last field.
      end if;
      Exit from this loop when the field chosen is accepting input.
    end loop;
    Set the cursor column to the end of the text in the chosen field.
    if (the length of the text fills the field) then
```

```
        Set the cursor column to the last column of the chosen field.
      end if;
    end if;
    Set the cursor row to the row of the chosen field.
  end Next_Field;
```

Move_Cursor_Right

```
  begin
    if (cursor is on the field text and the end of the field has not been reached) then
      Move the cursor to the next column.
    else
      Move to the next field.
    end if;
  end Move_Cursor_Right;


begin
  if (the OPERATION field of the input record is set to OP_MOVED) then
    Update the cursor offset values. The window object has moved and so the forms display
      is no longer at the same place it was when initialised.
    Set the cursor to it's new position.
  end if;
  if (the forms object is not active) then
    if (the OPERATION field of the input record is set to OP_NON_ACCEPTING) then
      Make the forms object active and show the hardware cursor.
      Set the OPERATION field of the input record to OP_DONE.
    else
      Return back to the calling routine.
    end if;
  else
    case (of the KEY field of the input record) is
      when (the left cursor key) =>
        if (the cursor is at the first column of the field) then
          Move to the previous field.
        else
          Move to the previous column.
        end if;
      when (the right cursor key) =>
        Move the cursor the next column or field.
      when (the cursor up key) =>
        Move to the previous field.
      when (the cursor down key) =>
        Move to the next field.
      when (the tab key) =>
        Process the field and hide the hardware curso.
        Set the forms object to be non-active.
        Set the OPERATION field of the input record to OP_NON_ACCEPTING.
      when (the backspace key) =>
        if (the cursor is not pointing to the first character in the field) then
          Move all the characters from the cursor to the end of the line, down one location.
          Move the hardware cursor to the left one column.
          Clear the last character of the line.
          Adjust the actual length of the line by decrementing it.
        end if;
```

```
      when (the delete key) =>
        Move all the characters from the character beyond the cursor to the end of the line,
          down one line.
        if (the cursor <= actual length of the line) then
          Clear the last character of the line.
          Adjust the actual length of the line by decrementing it.
        end if;
      when (the insert key) =>
        Toggle the inserting mode flag.
        if (the inserting flag is set) then
          Change the shape of the hardware cursor to a thin line.
        else
          Change the shape of the hardware cursor to a solid block.
        end if;
      when (the key is used to construct an integer or real value) =>
        if (the actual length of the line < field length) then
          if (the cursor = field length) then
            Adjust the actual length of the line by incrementing by 1.
          elsif (the inserting flag is set) then
            Move all the characters from the cursor to the end of the line, up one location.
            Adjust the actual length of the line by incrementing by 1.
          end if;
          Insert the new character into the line.
          Move the cursor to the next column or field.
        elsif (the inserting flag is cleared) then
          Insert the new character into the line.
          Move the cursor to the next column or field.
        end if;
      when others =>
        Clear the redraw flag.
    end case;
  end if;
  if (the redraw flag is set) then
    Draw the forms object and update the screen.
    if (the OPERATION field of the input record /= OP_NON_ACCEPTING) then
      Set the OPERATION field of the input record to OP_DONE.
    end if;
  end if;
end Handle_Input;
```

8.10 Windows

8.10.1 Purpose

The Window display object can be moved around the physical screen and contain other objects.

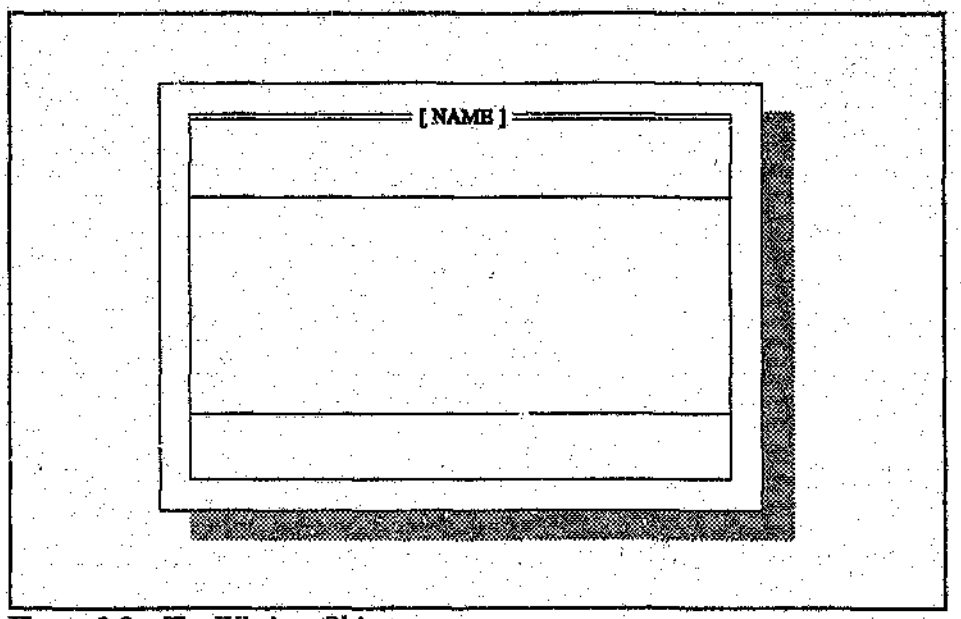

Figure 8-9 The Window Object

8.10.2 Appearance and Behaviour

The window appears on the physical screen as a window display with a shadow to the bottom right of it. It is framed with a single border. The window usually has a name and can have text in it. The window object also converts global coordinates to local coordinates for those objects that appear inside it.

The window can be moved around the screen using the mouse or the keyboard cursor keys. When the window is in a moving mode, it's frame will change to a different colour. To move the window with the mouse, press the left mouse button when the mouse pointer is on the top frame of the window. Keep the left button down while dragging the window to the desired position. The keyboard mode cannot be invoked directly, but by another object, usually a button.

8.10.3 Components

- *NAME_LENGTH* : The length of the window name. It cannot be larger than the length of the string for the name.
- *KEYS_LENGTH* : The length of the text that appears in the bottom section of the window object.
- *NAME* : The name of the window that will be displayed in the centre of the top frame of the window.
- *KEYS* : The text that appears in the bottom section of the window object. This text usually informs the user about the mapping of buttons to keyboard keys.
- *TOP_ROW* : The top row of the window object on the physical screen. These are in global

```
type WINDOW_TYPE is
   record
      NAME_LENGTH    : NATURAL;
      KEYS_LENGTH    : NATURAL;
      NAME           : STRING (1..15);
      KEYS_TEXT      : STRING (1..74);
      TOP_ROW        : ROW_TYPE;
      LEFT_SIDE      : COLUMN_TYPE;
      LENGTHS        : ROW_TYPE;
      WIDTHS         : COLUMN_TYPE;
      KEY_MOVING     : BOOLEAN  := FALSE;
      MOUSE_MOVING   : BOOLEAN  := FALSE;
      TEXT_COLOUR    : SCREEN_COLOUR_TYPE;
      FRAME_COLOUR   : SCREEN_COLOUR_TYPE;
      TITLE_COLOUR   : SCREEN_COLOUR_TYPE;
   end record;
```

coordinates.

- *LEFT_SIDE* : The left side of the window object on the physical screen. These are in global coordinates.
- *LENGTHS* : The length of the window object in rows. This includes the row on either side required for the border.
- *WIDTHS* : The width of the window object in columns. This includes the two columns on either side required for the border.
- *KEY_MOVING* : When this flag is set true, the window can be moved around the screen by the cursor keys. When the Enter key is pressed, the flag is cleared.
- *MOUSE_MOVING* : When this flag is set, the window is being moved by the mouse. The left mouse button must be kept down for this action. When the left mouse button is released, the flag is cleared.
- *TEXT_COLOUR* : The colour of text in the window. The background of the window is determined from the background colour.
- *FRAME_COLOUR* : The colour of the frame of the window. To make the frame invisible, make the frame foreground and background colour the same as that of the window background colour.
- *TITLE_COLOUR* : The colour of the title on the top frame of the window.

8.10.4 Methods

Close

Removes the Window from the physical screen.

```
begin
  Remove the window object from the physical screen.
end Close;
```

Create

Sets initial values for the appearance and behaviour of the Window.

```
begin
  if (the length * the width of the window >= maximum allowed window size) then
    Raise the exception that the window is too large.
  end if;
  Initialise the fields of the window data structure to the values that were passed in.
end Create;
```

Paragrapher

Writes a string to the Window. If a word carries on beyond the edge, it is wrapped around. If there are less than 11 characters on the line before the Window border. If there are more characters, the word is truncated.

```
begin
  loop
    Set ENDING = the length of a line on the window plus START - 1.
    if (the ENDING < length of the string) then
      for (the COUNTER2 in 0 to the wrap around length) loop
        if (the character of the string at (ENDING - COUNTER2) is a space) then
          Subtract COUNTER2 from ENDING.
          Exit from this loop.
        end if;
      end loop;
      Write the string from START to ENDING to the buffer.
      Set the START = ENDING + 1.
      Increment the first counter.
    else
      Write the string from START to last character of the string.
      Exit from this loop.
    end if;
  end loop;
end Paragrapher;
```

Draw

Draws the Window in the buffer and frames it. It writes the name and any text to the buffer.

```
begin
  Set the size of the window in the buffer.
  Update the appearance of the window object.
  Set the new location of the window object.
  Make a shadow.
end Draw;
```

Update

Draws the Window object. This is used when the Window object has not moved from it's initial position.

```
begin
  Clear and frame the window buffer.
  Write the name of window to the buffer.
  Write any text to the buffer, making sure that words shorter than 11 characters on a line are
  not truncated.
  Write the keys text to the buffer.
end Update;
```

Key_Move

Moves the Window around on the physical screen using the keyboard cursor keys.

```
begin
  case (of the KEY field of the input record) is
    when (the cursor up arrow key) =>
      if (the top row of the window is below the physical screen top line) then
        Decrement the top row of the window.
      end if;
    when (the cursor down arrow key) =>
      if (the top row of the window plus it's length is < the maximum length of the physical
        screen) then
        Increment the top row of the window.
      end if;
    when (the cursor left arrow key) =>
      if (the left side of the window is > the first column of the physical screen) then
        Decrement the left side of the window.
      end if;
    when (the cursor right key) =>
      if (the left side of the window plus it's width < the last column of the physical screen) then
        Increment the left side of the window.
      end if;
    when (the enter key) =>
      Clear the flag that shows that the window is being moved around the physical screen by
        the cursor_keys.
      The frame of the window is set back to normal.
      The X and Y fields of the input record are set to the top row and left side of the window
        respectively. The OPERATION field is set to OP_MOVED to tell any other object on
        top of the window that the window has moved.
      The flag to redraw the window is set.
    when others =>
      Do nothing.
  end case;
  if (the top row of the window is /= to the old top row value or the left side of the window is
    /= to the old left side value) then
    The flag to redraw the window is set.
    The old top row value is set to the present top row value of the window and the old left
```

```
      side value is set to the present left side value of the window.
  end if;
  if (the redraw flag is set) then
    Copy the main buffer to the screen.
    Copy the window buffer to the screen.
    Make a shadow for the window.
  end if;
  if (the OPERATION field of the input record is /= OP_MOVED) then
    Set the OPERATION field of the input record = OP_DONE.
  end if;
end Key_Move;
```

?" use_Move

Moves the Window around on the physical screen using the mouse.

```
begin
  if (the left mouse button has been pressed) then
    if (the mouse pointer's row coordinate is in the range in which the window's top row is
      allowed to be in) then
      Set the top row of the window to the row coordinate of the mouse pointer.
    else
      Set the top row of the window to the maximum value the top row can have.
    end if;
    if (the difference between the mouse pointer's column coordinate and the mouse pointer
      offset is > the difference between the maximum screen width and the window width) then
      Set the mouse pointer offset to the sum of the mouse pointer's column coordinate and the
        window width, but minus the maximum screen width.
    elsif (the difference between the mouse pointer's column coordinate and the mouse pointer
      offset is < physical screen's left side) then
      Set the window's left side to the physical screen's left side, and the mouse pointer offset
        to the mouse pointer's column coordinates.
    else
      Set the window's left side to the difference between the mouse pointer's column
        coordinate and the mouse pointer offset.
    end if;
    if (the top row of the window is /= to the old top row value or the left side of the window
      is /= to the old left side value) then
      The flag to redraw the window is set.
      The old top row value is set to the present top row value of the window and the old left
        side value is set to the present left side value of the window.
    end if;
  else
    Clear the flag that shows that the window is being moved around the physical screen by
      the mouse.
    The frame of the window is set back to normal.
    The X and Y fields of the input record are set to the top row and left side of the window
      respectively. The OPERATION field is set to OP_MOVED to tell any other object on
      top of the window that the window has moved.
    The flag to redraw the window is set.
  end if;
  if (the redraw flag is set) then
    Copy the main buffer to the screen.
```

```
    Copy the window buffer to the screen.
    Make a shadow for the window.
  end if;
  if (the OPERATION field of the input record is /= OP_MOVED) then
    Set the OPERATION field of the input record = OP_DONE.
  end if;
end Mouse_Move;
```

Handle_Input

Checks if the Window is in keyboard or mouse moving mode and acts accordingly. If not, it checks the mouse pointer and button to see if it must change the Window's state to mouse moving mode. If none of the previous cases are valid, it adjusts the coordinates from global to local.

```
begin
  if (the flag showing that the window is being moved around the physical screen by the mouse
    is set) then
    Move the window using the mouse.
  elsif (the flag showing that the window is being moved around the physical screen by the
    cursor keys is set) then
    Move the window using the cursor keys.
  elsif (the mouse is active and the mouse pointer is on the top frame of the window and the
    left mouse button is pressed down) then
    Set the flag that shows that the window is being moved around the physical screen by the
      mouse.
    The frame of the window is set to the moving frame and the window display on the physical
      screen is updated.
    The old top row value is set to the present top row value of the window and the old left
      side value is set to the present left side value of the window. The mouse pointer offset
      is set to the difference between the mouse pointer's column coordinate and the left side
      of the window.
    Set the OPERATION field of the input record = OP_DONE.
  elsif (the OPERATION field of the input record = OP_MOVE) then
    Set the flag that shows that the window is being moved around the physical screen by the
      cursor keys.
    The frame of the window is set to the moving frame and the window display on the physical
      screen is updated.
    The old top row value is set to the present top row value of the window and the old left
      side value is set to the present left side value of the window.
    Set the OPERATION field of the input record = OP_DONE.
  else
    Adjust the mouse pointer coordinates to local coordinates that are relative to the window's
      top left display corner. All screen objects on top of the window have coordinates that are
      relative to this top left display corner.
  end if;
end Handle_Input;
```

CHAPTER 9

THE HELP OBJECT

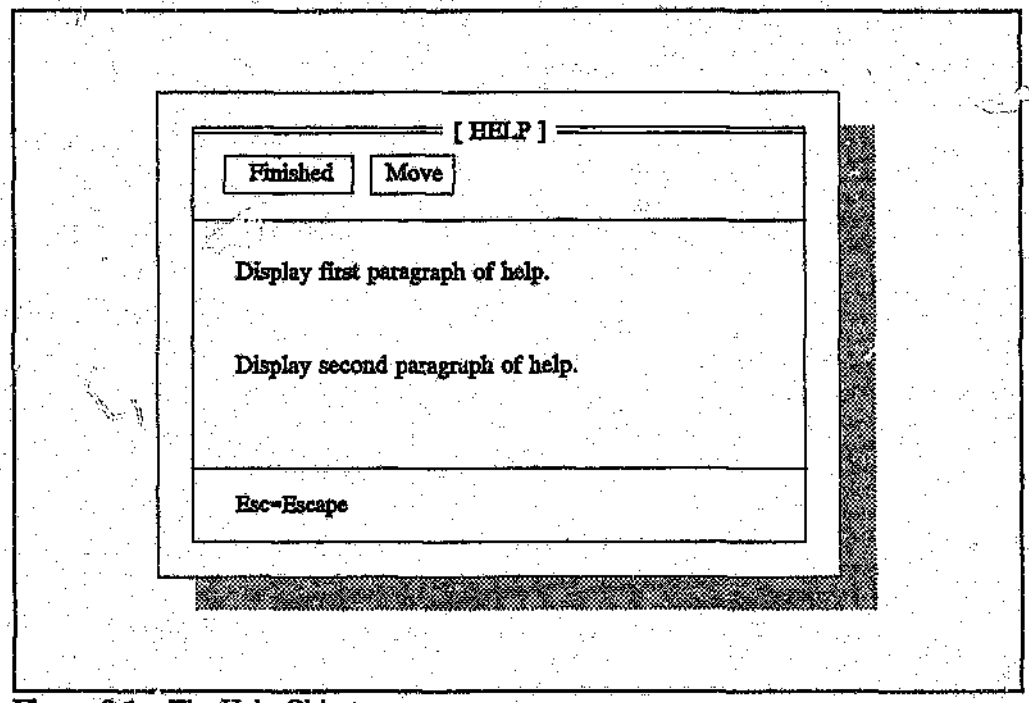

Figure 9-1 The Help Object

9.0.1 Purpose

Displays help text for the highlighted item on a menu, a particular subscreen, or during some process of the application program.

9.0.2 Appearance and Behaviour

The help text is a paragraph of information that can be placed anywhere on the physical screen. To prevent the help text from destroying any information that is already on the screen, the help object uses the window object buffer. The following procedure for displaying help is followed :

- Set up the window object buffer. This includes clearing and framing it
- Draw the help object in the window object buffer.
- Copy the window buffer to the physical screen.
- Make a shadow for the help object.

Once the help text has been read, copy the main buffer to the screen. It will appear as though the help object has been removed.

The help text can be quite large (depending on how many items of help it consists of) and it is not used that often. This would waste memory if it were hard-coded into the program - Memory that could be used in many of the application's simulation processes. The text is therefore written to a file (ELECTOR.HLP) that is stored on disk in the same directory as the program. If the help file is not found when help is requested, an error message will be displayed instead of the help text. If the help file is found, but there is no help text for the option requested, an appropriate error message will be displayed instead of the help text.

Two paragraphs of help text can be displayed. Words with less than 11 characters on a line are wrapped around to the next line, otherwise the words are truncated at the edge. The help window can be moved around the physical screen using the mouse, or by pressing the *Move* button and then using the cursor keys.

9.0.3 Components

```
type HELP_ITEM_TYPE is
   record
      INDEX          : HELP_SCREEN_TYPE;
      LINE1_LENGTH,
      LINE2_LENGTH   : NATURAL := 0;
      LINE1,
      LINE2          : STRING (1..250);
   end record;


type HELP_SCREEN_TYPE is
   record
      SCR       : Windows.WINDOW_TYPE;
      BUTTON1   : Buttons.BUTTON_TYPE;
      BUTTON2   : Buttons.BUTTON_TYPE;
   end record;
```

The *HELP_ITEM_TYPE* is used to retrieve the help text from the help disk file. The *HELP_SCREEN_TYPE* contains the screen objects that displays help text on the screen.

- *INDEX* : The help index associated with a help item. It is used to search the help file to find the help item needed.
- *LINE_LENGTH1* : Length of the first string that is used for the first paragraph of help text.
- *LINE_LENGTH2* : Length of the second string that is used for the second paragraph of help text.
- *LINE1* : First string that holds the text for the first paragraph of help text.
- *LINE2* : Second string that holds the text for the second paragraph of help text.
- *SCR* : The help window object.
- *BUTTON1* : The *Ok* button which is situated to the bottom left of the window.
- *BUTTON2* : The *Move* button which is situated to the bottom right of the window.

9.0.4 Methods

Initialize_Help_Path

Gets the default drive and directory of the help file at start of program execution. This will be used whenever the help file needs to be read.

```
begin
  Get default drive and directory.
end Initialize_Help_Path;
```

Show

This method is divided into three parts. The main body handles input that is directed to the help object. The other two parts deal with finding the help item in the disk file ELECTOR.HLP, and displaying the help on the screen.

Find_Help

```
begin
  begin
    Open the disk file ELECTOR.HLP.
  exception
    when (the file cannot be found) =>
      Clear the flag of found file.
      Assign a missing file error message to the help_item record.
  end;
  if (the help file was found) then
    Clear the flag of found help item.
    while (not the end of the help file) loop
      Read a help item from the disk file.
      if (the help required = the item read from the file) then
        Set the flag of found help item.
        Exit from this loop.
      end if;
    end loop;
    if (the help item has not being found) then
      Assign an error message to the help_item record saying that there is no help available
        for this item.
    end if;
    Close the help disk file.
  end if;
end Find_Help;
```

Display_Help

```
  begin
    Create the Help window object.
    Create the Finished button.
    Create the Move button.
    Instruct the help window object to draw itself.
    if (there is a second paragraph of help) then
      Display the second paragraph making sure that words near the edge are not truncated,
        but moved to the next l' ie.
    end if;
    Instruct the Finished button object to draw itself.
    Instruct the Move button object to draw itself.
    Write the window buffer to the physical screen.
  end Display_Help;


begin
  Find the he!, item form the disk file.
  Display the help on the screen.
  loop
    Get input from the Inputter object.
    Send the input to the window object. Any input that concerns it, will be handled by it.
    Send the input to the Finished button object. Any input that concerns it, will be handled by it.
    Send the input to the Move button object. Any input that concerns it, will be handled by it.
    if (the OPERATION field of the input record = OP_MOVE) then
      Send the inp[t to the window object so that it can put itself in the keyboard move mode
        (Moved around on the screen with the cursor keys).
    elsif (the OPERATION field of the input record = OP_FINISHED) then
      Exit form this loop.
    end if;
  end loop;
  Copy the n. in buffer to the screen.
end Menu_Hi
```

CHAPTER 10

THE SCREENS OBJECT

10.1 Introduction

This package uses the *Screen Objects* to build larger objects which are more useful, but also more complex. A generic subscreen object builds the foundation for the ERROR, PROMPT and STATUS subscreen objects.

10.2 Subscreen

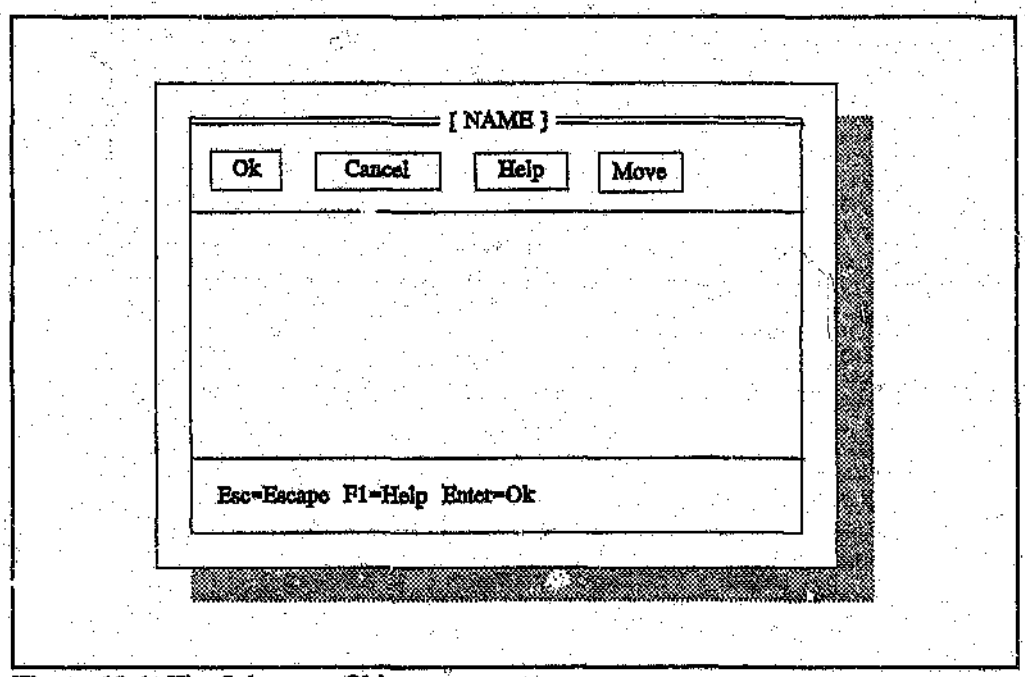

Figure 10-1 The Subscreen Object

10.2.1 Purpose

Subscreens get input from the user and give output to the user. Input is a ;:cifying parameters for processes or choosing from a list of options. Output is status or error messages.

10.2.2 Appearance and Behaviour

The subscreen object can be placed anywhere on the physical screen. To prevent the subscreen from destroying any information that is already on the screen, the subscreen uses the Window

object buffer. There are two different representations for the Window object buffer (See Chapter 7). The following process is followed :

- Set up the Window object buffer. This includes clearing and framing it.
- Copy the part of the physical screen that is going to be overwritten to a separate buffer. (Required for method 1).
- Draw the subscreen object in the window buffer. The order of execution between the previous point and this one can be interchanged.
- Copy the window buffer to the physical screen.
- Make a shadow for the subscreen object.

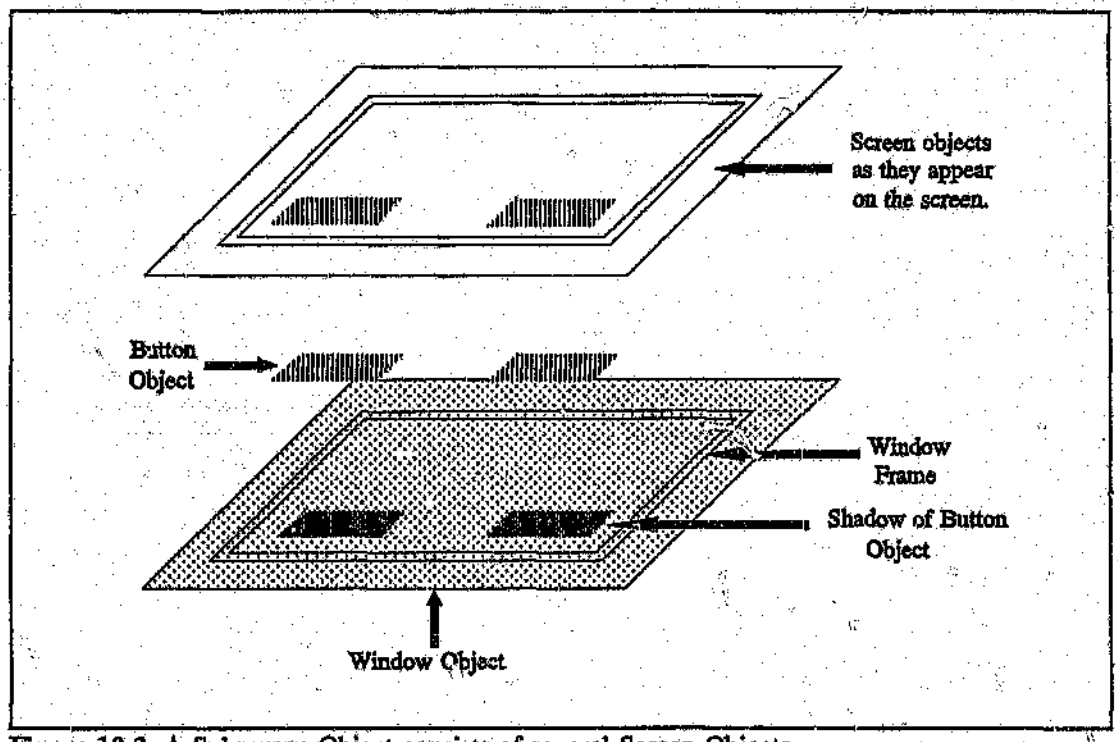

Figure 10-2 A Subscreen Object consists of several Screen Objects.

Once the subscreen is no longer needed, using method 1 :

- Remove the shadow.
- Copy the contents of the second buffer back to the physical screen. It will appear as though the subscreen has been removed.

Once the subscreen is no longer needed, using method 2 :

- Copy the main buffer to the screen. It will appear as though the subscreen has been removed.

The subscreen object consists of a window object and four buttons - *Ok*, *Cancel*, *Help* and *Move*. The subscreen can be moved around the physical screen using the mouse, or by pressing the *Move* button and then using the cursor keys.

Button	Purpose of the Button
Ok	This tells the subscreen object that the input entered is correct (if input was requested) or that the message has been read.
Cancel	This cancels the present input as well as the present process. Control is given back to the scheduler. This appears to the user as though the process was cancelled, another process can then be chosen from the menus.
Help	This calls the help object to display on-line help about the present subscreen or a part of the application program.
Move	This allows the subscreen to be moved around the physical screen by the cursor keys. The frame will change colour when this mode is active. Once a suitable position is found for the subscreen, press the Enter key. The frame will change back to it's normal colour.

Table 10-1 Purpose of the Subscreen Buttons

10.2.3 Components

```
type STATUS_TYPE is
   record
      OBJECT_ACTIVE    : INTEGER_TYPE;
      CURSOR_ACTIVE    : BOOLEAN;
      NON_ACCEPTANCE   : BOOLEAN;
      HELP_INFO        : INTEGER_TYPE;
      TEXT_LENGTH      : NATURAL;
      TEXT             : STRING (1..100);
   end record;

type BUTTON_HOLDER_ARRAY is array (1..4) of Buttons.BUTTON_TYPE;

type SCREEN_TYPE is
   record
      SCR          : Windows.WINDOW_TYPE;
      BUTTONS      : BUTTON_HOLDER_ARRAY;
      BL_BUTTONS   : Boolean_Buttons.BOOLEAN_BUTTON_TYPE;
      LINER        : Input_Lines.LINE_TYPE;
      PICKER       : Pick_Lists.LIST_TYPE;
      PICKER2      : Pick_Lists.LIST_D_TYPE;
      FORMER       : Forms.FORM_TYPE;
      STATUS       : STATUS_TYPE;
   end record;
```

- *OBJECT_ACTIVE* : Holds the code determining the active screen objects for the subscreen.
- *CURSOR_ACTIVE* : A flag which determines if the cursor should be active. It is needed when getting help for a subscreen that has an active Input_Line or Forms object.
- *NON_ACCEPTANCE* ; A flag used to pass control between the Pick_List and Forms objects.
- *HELP_INFO* : The help index for the subscreen.
- *TEXT_LENGTH* : Length of the text that is displayed on the subscreen.
- *TEXT* : Text that is displayed on the subscreen.

- *SCR* : The Window object on which the subscreen is build.
- *BUTTONS* : An array of four buttons which initiate certain processes.
- *BL_BUTTON* : A Boolean_Buttons object if needed by the PROMPT subscreen object.
- *LINER* : A Input_Line object if needed by the PROMPT subscreen object.
- *PICKER* : A static Pick_List object if needed by the PROMPT subscreen object.
- *PICKER2* : A dynamic Pick_List object if needed by the PROMPT subscreen object.
- *FORMER* : A Forms object if needed by the PROMPT subscreen object.
- *STATUS* : The present status of certain properties of the subscreen.

10.2.4 Methods

Halt_Processing

Checks to see if a key was waits until another key is pressed.

```
begin
   Get the status of the keyboard buffer.
   if (there is a keyboard character in the keyboard buffer) then
      Get the key that was pressed.
   end if;
   Return the status of the keyboard buffer.
end Halt_Processing;
```

Continue_Processing

Waits until a key is pressed on the keyboard.

```
begin
   Wait until a key is pressed on the keyboard.
end Continue_Processing;
```

Create

Sets initial values for the appearance and behaviour of the subscreen. A subscreen consists of various screen objects which each need to be initialized.

```
begin
   Create the Ok button with the required properties.
   Create the Cancel button with the required properties.
   Create the Move button with the required properties.
   Create the Help button with the required properties.
```

```
case (the additional active screen object) is
  when (Boolean_Buttons) =>
    Create the Boolean_Button with the required properties.
  when (Input_Line) =>
    Create the Input_Line with the required properties.
  when (dynamic Pick_List) =>
    Create the dynamic Pick_List with the required properties.
  when (Form or static Pick_List) =>
    begin
      if (there is a form and a list) then
        Add "Tab=List<->Form" to the keys text.
      end if;
      if (there is a list) then
        for (the COUNTER from 1 to last option) loop
          if (indexed option text length > maximum text length) then
            Set the maximum text length to the indexed option text length.
          end if;
        end loop;
        Create the static Pick_List with the required properties.
        Calculate left margin for form.
      else
        Calculate left margin for form.
      end if;
      if (there is a form) then
        for (the COUNTER from 1 to last field) loop
          if (the indexed field is accepting input) then
            if (the indexed field value length > maximum value length) then
              Set the maximum value length to the indexed field value length.
            end if;
            if (the indexed field text length > maximum text length) then
              Set the maximum text length to the indexed field text length.
            end if;
            if (the indexed field row > maximum row) then
              Set the maximum row to the indexed field row.
            end if;
          end if;
        end loop;
      end if;
      if (the width of the window < least possible width) then
        Set the width of the window to the least possible width.
      end if;
      if (the length of the window < list length plus padding) then
        Set the length of the window to the list length plus padding.
      end if;
    end;
  when others => Do nothing.
end case;
Create a Window with the required properties.
end Create;
```

Draw

Draws or updates the appearance of the subscreen object in the window buffer which is then

copied to the physical screen.

```
begin
  if (updating the screen) then
    Update the window buffer.
  else
    if (the cursor active status is set) then
      Show the hardware cursor.
    end if;
    Draw the subscreen in the window buffer.
  end if;
  for (the COUNTER from 1 to the number of screen buttons) loop
    Draw the screen button indexed by COUNTER.
  end loop;
  case (the additional active screen object) is
    when (Boolean_Buttons) =>
      Draw the Boolean_Button in the window buffer.
    when (Input_Line) =>
      Draw the Input_Line in the window buffer.
    when (dynamic Pick_List) =>
      Draw the dynamic Pick_List in the window buffer.
    when (Form or static Pick_List) =>
      if (there is a list) then
        Draw the static Pick_List in the window buffer.
      end if;
      if (there is a form) then
        Draw the Form in the window buffer.
      end if;
    when others => Do nothing.
  end case;
  Copy the window buffer to the screen and show the mouse pointer.
end Draw;
```

Process_Form

Converts the data in a field of a Form to an integer or real number. If there is an error, an error message will be displayed.

Field_Error

```
  begin
    Write error message to buffer and then display on screen.
    Delay for about 1.5 seconds.
    Redraw the subscreen.
  end Field_Error;

begin
  Handle the input for the forms.
  if (the OPERATION field of the input record = OP_NON_ACCEPTING) then
    Set the subscreen flag for non acceptance to true.
```

```
  end if;
  Set the OPERATION field of the input record to OP_FORM_PROCESSED.
exception
  when (unrecognised integer value) =>
    Write error message that integer value is not recognised.
  when (integer value < minimum) =>
    Write error message that integer value must be >= minimum.
  when (integer value > maximum) =>
    Write error message that integer value must be <= maximum.
  when (two specified values equal) =>
    Write error message that integer value cannot equal other group values.
  when (unrecognised real value)=>
    Write error message that real value is not recognised.
  when (real value < minimum) =>
    Write error message that real value must be >= minimum.
  when (integer value > maximum) =>
    Write error message that real value must be <= maximum.
end Process_Form;
```

Handle_Input

This is a generalised routine that is used by most of the objects in the Screens package. It sends the input to the Window object and Button objects. Any input that concerns them will be handled by them. If the *Cancel* button is pressed, a USER_ESCAPE exception will be generated and the subscreen object will be removed from the screen. If the *Help* button is pressed, a help object with relevant information about this subscreen object, or part of the application program, will be displayed.

```
begin
  Send the input to the window object. Any input that concerns it, will be handled by it.
  case (of the KEY field of the input record) is
    when (the enter key) =>
      Set the KEY field of the input record to that for the Ok button.
    when (the F1 key) =>
      Set the KEY field of the input record to that for the Help button.
    when (the escape key) =>
      Set the KEY field of the input record to that for the Cancel button.
    when others => Do nothing.
  end case;
  Set a counter to index the first button.
  loop
    Send the input to the button object indexed by the counter. Any input that concerns it
      will be handled by it.
    Exit from this loop when the input has been handled by a button or there are no more
      buttons.
    Increment the counter for the next button.
  end loop;
  case (of the OPERATION field of the input record) is
    when (the Cancel button was pressed) =>
      if (the cursor active flag is set) then
        Hide the hardware cursor.
```

```
    end if;
    Clear the present subscreen from the screen.
    Raise the exception USER_ESCAPE.
  when (the Help button was pressed) =>
    Clear the present subscreen from the screen.
    if (the cursor active flag is set) then
      Hide the hardware cursor.
    end if;
    Get the help object to display the relevant help.
    if (the cursor active flag is set) then
      Show the hardware cursor.
    end if;
    Show the mouse pointer.
    Draw the subscreen in the window buffer.
  when (the Move button was pressed) =>
    Send input to the Window's Handle_Input to put it in moving mode.
  when (the Window has moved) =>
    case (of the active screen objects) is
      when (Input_Line) =>
        Send input to the Input_Line's Handle_Input to get the new offset coordinates.
      when (Forms) =>
        Send input to the Forms's Handle_Input to get the new offset coordinates.
      when others => Do nothing.
    end case;
  when (OP_DO) =>
    case (the additional active screen object) is
      when (Boolean_Buttons) =>
        case (of the KEY field of the input record) is
          when (Y or y key) =>
            Set the KEY field of the input record to that for the Yes button.
          when (N or n key) =>
            Set the KEY field of the input record to that for the No button.
          when others => Do nothing.
        end case;
        Send the input to the Boolean_Buttons object. Any input that concerns it
          will be handled by it.
      when (Integer_Input_Line) =>
        if (the KEY field of the input record is in 0..9 | MINUS | BACKSPACE |
          a key > 255) then
          Send the input to the Input_Line object. Any input that concerns it
            will be handled by it.
        end if;
      when (Real_Input_Line) =>
        if (the KEY field of the input record is in 0..9 | E | e | PERIOD | MINUS |
          BACKSPACE | a key > 255) then
          Send the input to the Input_Line object. Any input that concerns it
            will be handled by it.
        end if;
      when (String_Input_Line) =>
        Send the input to the Input_Line object. Any input that concerns it
          will be handled by it.
      when (dynamic Pick_List) =>
        Send the input to the dynamic Pick_List object. Any input that concerns it
          will be handled by it.
      when (Form or static Pick_List) =>
        if (non acceptance flag is set) then
```

```
            Clear the non acceptance flag.
            Set the cursor active flag to false.
            Set the OPERATION field of the input record to = OP_NON_ACCEPTING.
          end if;
          if (there is a list) then
            Send the input to the static Pick_List object. Any input that concerns it
              will be handled by it.
          end if;
          if (the OPERATION field of the input record = OP_NON_ACCEPTING) then
            Set the cursor active flag to true.
          end if;
          if (there is a form and the OPERATION field of the input record /= OP_DONE) then
            Send the input to the Form object. Any input that concerns it
              will be handled by it.
          end if;
        when others => Do nothing.
      end case;
    when others => Do nothing.
  end case;
end Handle_Input;
```

10.3 Error Subscreen

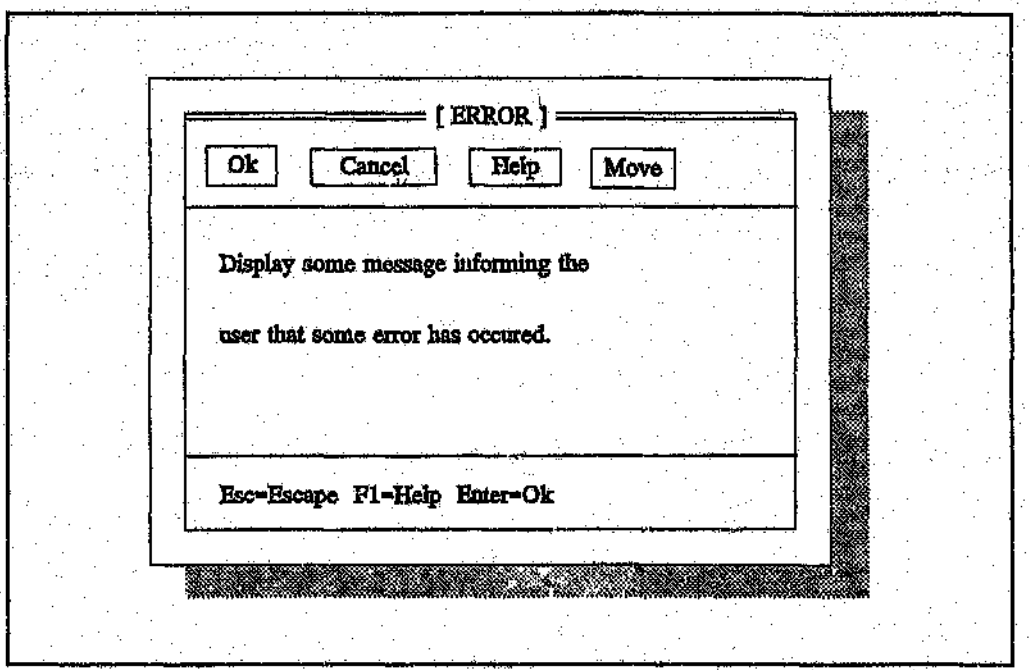

Figure 10-3 The Error Subscreen Object

10.3.1 Purpose

The ERROR subscreen displays error messages that the program needs to inform the user about.

The frame will appear in red as an additional reminder that an error has occurred.

10.3.2 Methods

Write

Creates an ERROR subscreen object. Displays the error message and once it has been read, it is removed from the screen.

```
begin
  Set the additional active screen objects to none.
  Set the cursor active flag to false.
  Set the help info to the present help info.
  Set the text for the subscreen to the message passed in.
  Create the Error subscreen.
  Draw the Error subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Error subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      Exit from this loop.
    end if;
  end loop;
  Remove the Error subscreen from the physical screen.
end Write;
```

10.4 Prompt Subscreen

10.4.1 Purpose

The PROMPT subscreen displays a message and then gets the requested input. Different *Screen Objects* will be used to get different types of input.

Screen Object	Type of Input
Boolean_Buttons	Boolean response (Yes/No)
Input_Line	Integer value
Input_Line	Real value
Input_Line	String
Form	Related integer and real values
Pick_List (static)	List
Pick_List (dynamic)	List (usually filenames)

Table 10-2 Different types of Prompt Subscreen Inputs

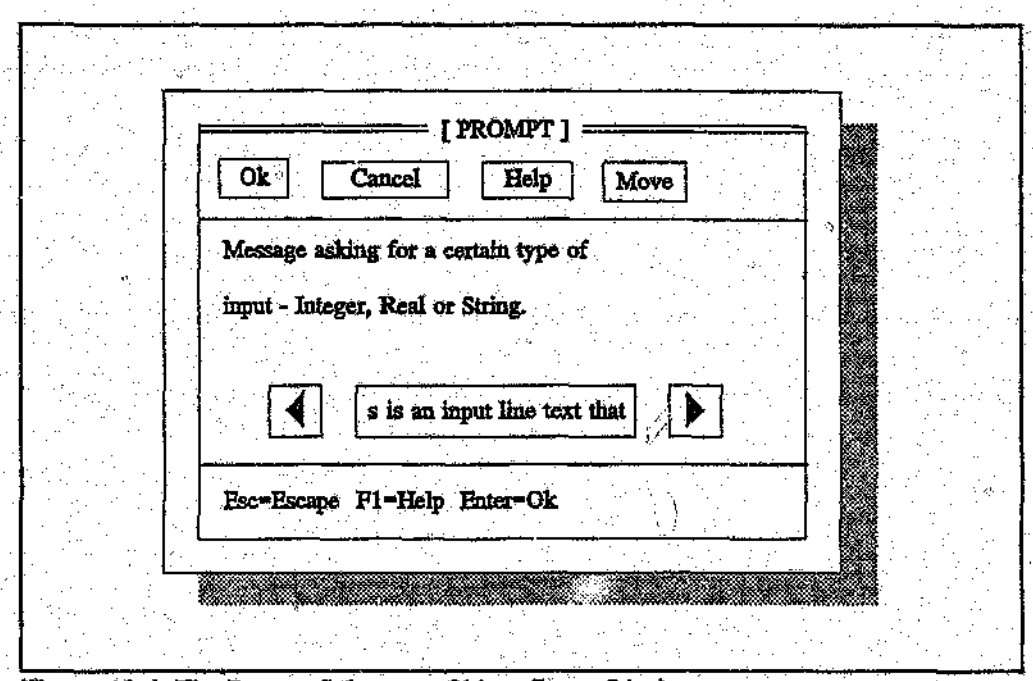

Figure 10-4 The Prompt Subscreen Object (Input_Line)

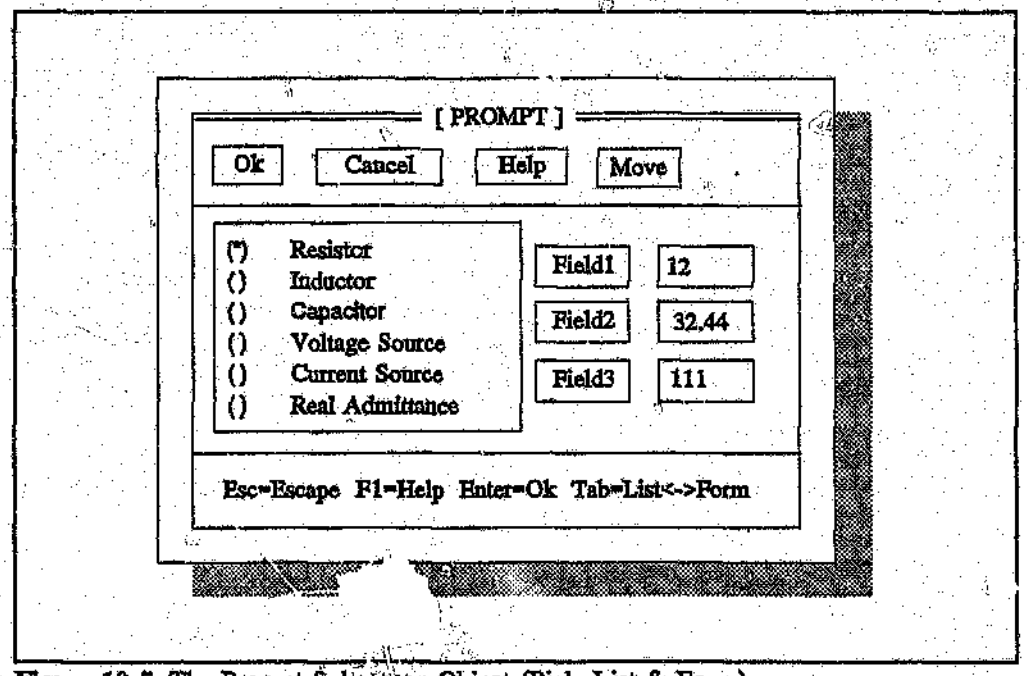

Figure 10-5 The Prompt Subscreen Object (Pick_List & Form)

10.4.2 Methods

Add_To_List

Adds another option to the dynamic list from which the user must choose.

```
begin
  Add name to dynamic list.
end Add_To_List;
```

Read (List and Form)

Creates and draws a PROMPT subscreen with a Pick_List and Forms object. Options on the list can be set and the form is filled in by the user. After the *Ok* button is pressed, the subscreen is removed from the screen.

```
begin
  Set the additional active screen objects to the static Pick_List and Form.
  Set the non accepting flag to true.
  Set the cursor active flag to false.
  Set the help info to the present help info.
  Create the Prompt subscreen.
  Draw the Prompt subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Prompt subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      if (there is a form) then
        Process the input of the present field.
        if (the OPERATION field of the input record = OP_FORM_PROCESSED) then
          Exit from this loop.
        end if;
      else
        Exit from this loop.
      end if;
    end if;
  end loop;
  Remove the Prompt subscreen from the physical screen.
end Read;
```

Read (Form)

Creates and draws a PROMPT subscreen with a Form object. The form is filled in by the user. After the *Ok* button is pressed, the subscreen is removed from the screen.

```
begin
  Call generalised Read method with an actual form but an empty list.
end Read;
```

Read (List)

Creates and draws a PROMPT subscreen with a Pick_List object. Options on the list can be set by the user. After the *Ok* button is pressed, the subscreen is removed from the screen.

```
begin
  Call generalised Read method with an actual list but an empty form.
end Read;
```

Read (Dynamic list)

Creates and draws a PROMPT subscreen with a dynamic Pick_List object. An option on the list can be chosen by the user. After the *Ok* button is pressed, the subscreen is removed from the screen.

```
begin
  Set the additional active screen objects to the dynamic Pick_List.
  Set the cursor active flag to false.
  Set the help info to the present help info.
  Create the Prompt subscreen.
  Draw the Prompt subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Prompt subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      Get the option name from the dynamic Pick_List.
      Exit from this loop.
    end if;
  end loop;
  Remove the Prompt subscreen from the physical screen.
end Read;
```

Read (Boolean)

Creates and draws a PROMPT subscreen object with a Boolean_Button object. The subscreen is removed from the screen after a boolean value has been entered.

```
begin
  Set the additional active screen objects to the Boolean_Buttons.
  Set the cursor active flag to false.
  Set the help info to the present help info.
  Set the text for the subscreen to the message passed in.
  Create the Prompt subscreen.
  Draw the Prompt subscreen.
  loop
    Get input from the Inputter object.
```

```
    Send the input to the Prompt subscreen object. Any input that concerns it will be handled by it.
    case (the OPERATION field of the input record) is
      when (OP_OK or OP_TRUE) =>
        Set the output item to true.
        Exit from this loop.
      when OP_FALSE =>
        Set the output item to false.
        Exit from this loop.
      when others => Do nothing.
    end case;
  end loop;
  Remove the Prompt subscreen from the physical screen.
end Read;
```

Read (Integer)

Creates and draws a PROMPT subscreen object with an Input_Line object. An integer value is entered by the user. If it is not correct, an error message is displayed before getting more input. Once correct input has been entered, the subscreen is removed from the screen.

```
begin
  Set the additional active screen objects to the Integer_Input_Line.
  Set the cursor active flag to true.
  Set the help info to the present help info.
  Set the text for the subscreen to the message passed in.
  Create the Prompt subscreen.
  Draw the Prompt subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Prompt subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      begin
        Get the string from the Input_Line object.
        Convert the string to an Integer value.
        Exit from this loop.
      exception
        when (an error in the conversion) =>
          Display an error message in the subscreen.
      end;
    end if;
  end loop;
  Remove the Prompt subscreen from the physical screen.
end Read;
```

Read (Real)

Creates and draws a PROMPT subscreen object with an Input_Line object. A real value is entered by the user. If it is not correct, an error message is displayed before getting more input. Once

correct input has been entered, the subscreen is removed from the screen.

```
begin
  Set the additional active screen objects to the Real_Input_Line.
  Set the cursor active flag to true.
  Set the help info to the present help info.
  Set the text for the subscreen to the message passed in.
  Create the Prompt subscreen.
  Draw the Prompt subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Prompt subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      begin
        Get the string from the Input_Line object.
        Convert the string to a Real value.
        Exit from this loop.
      exception
        when (an error in the conversion) =>
          Display an error message in the subscreen.
      end;
    end if;
  end loop;
  Remove the Prompt subscreen from the physical screen.
end Read;
```

Read (String)

Creates and draws a PROMPT subscreen object with an Input_Line object. A string is entered by the user. If no characters were entered, an error message is displayed before getting more input. Once correct input has been entered, the subscreen is removed from the screen.

```
begin
  Set the additional active screen objects to the String_Input_Line.
  Set the cursor active flag to true.
  Set the help info to the present help info.
  Set the text for the subscreen to the message passed in.
  Create the Prompt subscreen.
  Draw the Prompt subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Prompt subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      Get the string from the Input_Line object.
      if (there is at least one character) then
        Set the output string to this string.
        Exit from this loop.
      else
        Display an error message in the subscreen.
      end if;
    end if;
```

end loop;
Remove the Prompt subscreen from the physical screen.
end Read;

10.5 Status

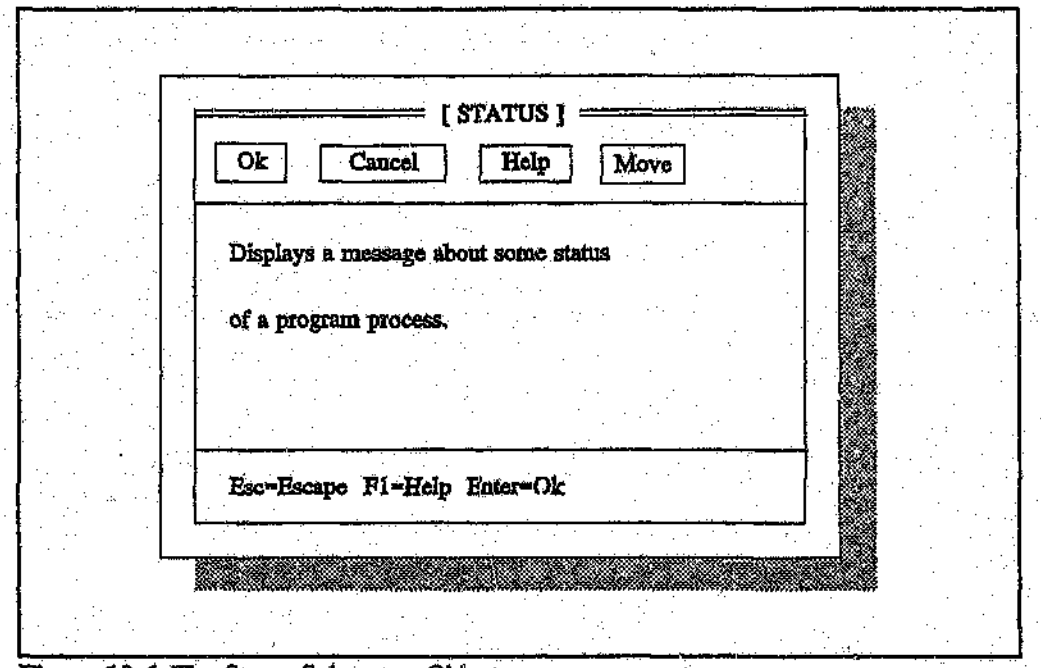

Figure 10-6 The Status Subscreen Object

10.5.1 Purpose

The STATUS subscreen displays status messages that the program needs to inform the user about.

10.5.2 Methods

Write

Creates a STATUS subscreen object. Displays a status message and once it has been read, it is removed from the screen.

begin
Set the additional active screen objects to none.
Set the cursor active flag to false.

```
  Set the help info to the present help info.
  Set the text for the subscreen to the message passed in.
  Create the Status subscreen.
  Draw the Status subscreen.
  loop
    Get input from the Inputter object.
    Send the input to the Status subscreen object. Any input that concerns it will be handled by it.
    if (the OPERATION field of the input record = OP_OK) then
      Exit from this loop.
    end if;
  end loop;
  Remove the Status subscreen from the physical screen.
end Write;
```

Wait

Displays the message : "Press any key to continue." This method is used when a screen is full of output and any more would cause some of the previous output to be lost. This gives the user a chance to first read the output before it is overwritten.

```
begin
  Calls the Status write method with the message that the Ok button must be pressed to continue
    with processing.
end Wait;
```

10.6 Main Screen

10.6.1 Purpose

The *Main* screen is an object incorporating the physical screen. Objects in the application part of the program can request services from the *Main* screen object - they cannot do screen output themselves. This permits the decoupling of the physical screen from the rest of the program.

10.6.2 Appearance and Behaviour

The *Main* screen consists of 24 rows by 80 columns - though this can be changed. There is an internal cursor which keeps track of where the next character will be written. A disk device and printer device can be engaged by opening a disk file or printer file. These files can be opened or closed at any time while the program is executing. However, it was found to be more economical to have only the disk file facility. Printer output can be generated by sending the disk file to the printer.

10.6.3 Methods

New_Cursor_Position

Calculates the new cursor position after a string has been written to the Main screen.

```
begin
  Calculate the new position by adding COLUMNS and the length of the string to be printed.
  if (this position > last column of the Main screen) then
    Calculate the number of rows that will be needed and prepare the Main screen for the next
      writing.
  else
    Set COLUMNS to the new position after writing.
  end if;
exception
  when (the ROWS or COLUMNS variables do not point to a location on the Main screen) =>
    Set ROWS to the last row on the Main screen.
    Set COLUMNS to the last column on the Main screen.
end New_Cursor_Position;
```

Clear

Clears the Main screen from the second row to the last. The first row is where the spread menu appears.

```
begin
  Clear the Main screen from the second row downward. This is done because it must not
    interfere with the menu line on the first row.
  Home the cursor.
end Clear;
```

Cursor_Read

Returns the present cursor position on the Main screen.

```
begin
  Return the values of the internal ROWS and COLUMNS variables which store the next
    location for a write to the Main screen operation.
end Cursor_Read;
```

Cursor_Set

Sets the cursor position on the Main screen.

```
begin
  Set the values of the internal ROWS and COLUMNS variables which store the next location
    for a write to the Main screen operation.
end Cursor_Set;
```

Cursor_Down

Moves the cursor down one or more rows.

```
begin
  if (ROWS plus the number of rows to move down >= the last row on the Main screen) then
    Set ROWS to the last row on the Main screen.
  else
    Add the number of rows to move down to ROWS.
  end if;
end Cursor_Down;
```

Cursor_Up

Moves the cursor up one or more rows.

```
begin
  if (ROWS minus the number of rows to move up <= the second row on the Main screen) then
    Set ROWS to the second row on the Main screen.
  else
    Subtract the number of rows to move up from ROWS.
  end if;
end Cursor_Up;
```

Cursor_Right

Moves the cursor right one or more columns.

```
begin
  if (COLUMNS plus the number of columns to move right >= the last column on the Main
    screen) then
    Set COLUMNS to the last column on the Main screen.
  else
    Add the number of columns to move right to COLUMNS.
  end if;
```

```
end Cursor_Right;
```

Cursor_Left

Moves the cursor left one or more columns.

```
begin
  if (COLUMNS minus the number of columns to move left <= the first column on the Main
    screen) then
    Set COLUMNS to the first column on the Main screen.
  else
    Subtract the number of columns to move left from COLUMNS.
  end if;
end Cursor_Left;
```

Cursor_Next_Line

Moves the cursor to the start of the next line. This includes sending line feeds to disk and printer files if they are active.

```
begin
  if (the next row for a write >= last row on the Main screen) then
    Set ROWS to the last row of the Main screen.
    Scroll the whole Main screen up one row.
  else
    Increment ROWS by 1.
  end if;
  Set COLUMNS to first column of the Main screen.
  if (output to the disk is active) then
    Write a new_line to the disk file.
  end if;
  if (output to the printer is active) then
    Write a new_line to the printer.
  end if;
end Cursor_Next_Line;
```

Cursor_Home

Moves the cursor to the start of the second row. The first row is occupied by the spread menu.

```
begin
  Set the next position to write to the top left corner of the Main screen. ie. The second row.
end Cursor_Home;
```

Cursor_Left_Margin

Moves the cursor to the left column.

```
begin
  Set the next position to write to the first column.
end Cursor_Left_Margin;
```

Open_Screen_Devices

Opens and closes a disk and printer file depending on the user's response.

```
begin
  Get options from the keyboard to see if any devices must change their status.
  if (a disk file must be active and there is no active file) then
    begin
      Get the name and path of the disk file.
      Open the disk file to read.
      Close the disk file.
      Display that the file already exists and ask if it should be overwritten. Get the response
        from the keyboard.
      if (the file must be overwritten) then
        Open the disk file to write.
      end if;
    exception
      when (the name of the file cannot be found) =>
        Create the file.
    end;
  elsif (a disk file must be inactive and there is one active) then
    Close the disk file.
  end if;
  if (the printer must be active and it is presently not) then
    Open the printer file.
  elsif (the printer must be inactive and it is presently not) then
    Close the printer file.
  end if;
end Open_Screen_Devices;
```

Process_Title

Writes the title of a processes to the Main screen. The title is centred on the first row.

```
begin
  Clear the Main buffer and make a single band along the top.
  Write the process title in the centre of this band.
  Copy the Main buffer to the screen.
end Process_Title;
```

Write (Integer)

Writes an integer value to the Main screen.

```
begin
  Convert the integer value to a string.
  Write the string to the Main buffer.
  Copy the Main buffer to the screen.
  Calculate the new cursor position.
  if (the output to disk is active) then
    Write the string to the disk file.
  end if;
  if (the output to printer is active) then
    Write the string to the printer.
  end if;
end Write;
```

Write (Real)

Writes a real value to the Main screen.

```
begin
  Convert the real value to a string.
  Write the string to the Main buffer.
  Copy the Main buffer to the screen.
  Calculate the new cursor position.
  if (the output to disk is active) then
    Write the string to the disk file.
  end if;
  if (the output to printer is active) then
    Write the string to the printer.
  end if;
end Write;
```

Write (String)

Writes a string in a certain format to the Main screen

```
begin
  case (of the form the string is to take) is
    when (bold) =>
      Set certain colour combinations for this attribute.
    when (heading) =>
      Set certain colour combinations for this attribute.
    when (normal) =>
      Set certain colour combinations for this attribute.
    when (reversed) =>
      Set certain colour combinations for this attribute.
```

```
    when (underlined) =>
      Set certain colour combinations for this attribute.
  end case;
  Write the string to the Main buffer.
  Copy the Main buffer to the screen.
  Calculate the new cursor position.
  if (the output to disk is active) then
    Write the string to the disk file.
  end if;
  if (the output to printer is active) then
    Write the string to the printer.
  end if;
end Write;
```

CHAPTER 11

THE MENUS OBJECT

11.1 Introduction

There are two types of menus - Spread and Stacked. They have the same internal structure and use the same methods. The difference between the two menus are their appearance and the extra hotkey selection option for Stacked menus.

11.1.1 Purpose

The menus allow selection of certain options which correspond to the operations and process that the program can undertake.

11.1.2 Appearance and Behaviour

The menus can appear in two different forms, either as spread-out menus or stacked menus. Menu options can be selected in three different ways. For certain options in stacked menus, a fourth option - hotkeys, can be used.

11.1.3 Components

The MENU_LINK type definition is a recursive definition needed for the dynamic tree structure of the menus.

- *FRAME* : The foreground and background colours for the frame of the menu. Usually the background colour is the same as that of the rest of the window (text colour). The frame can be set to blink.
- *TEXT* : The foreground and background colours for the text of the selectable options. This colour is also used as the colour of the window object, excluding the frame colour. The text can be set to blink.
- *NON_SELECT* : The foreground and background colours for options that are not selectable. The non selectable options can be made to blink, but this will be confusing. Usually the colour is duller than that of the text colour.
- *SHORTCUT* : The foreground and background colours for the shortcut character that is highlighted for each menu option. The background colour is usually the same as that of the text background colour . The character can be made to blink.
- *SELECT_BAR* : The foreground and background colours for the selection bar. The background colour is usually different from the text colour so that the bar will be distinct.

```
type TYPES_OF_MENU_TYPE is (STACKED_MENU, SPREAD_MENU);

type COLOUR_TYPE is
   record
      FRAME        : SCREEN_COLOUR_TYPE;
      TEXT         : SCREEN_COLOUR_TYPE;
      NON_SELECT   : SCREEN_COLOUR_TYPE;
      SHORTCUT     : SCREEN_COLOUR_TYPE;
      SELECT_BAR   : SCREEN_COLOUR_TYPE;
   end record;

type MENU_TYPE (LENGTHS : ROW_TYPE);
type MENU_LINK is access MENU_TYPE;

type MENU_TYPE (LENGTHS : ROW_TYPE) is
   record
      TYPE_OF_MENU  : TYPES_OF_MENU_TYPE;
      PARENT        : MENU_LINK;
      DATA          : DATA_TYPE (1..LENGTHS);
      CHOICE        : ROW_TYPE;
      TOP_ROW       : ROW_TYPE;
      LEFT_SIDE     : COLUMN_TYPE;
      WIDTHS        : COLUMN_TYPE;
   end record;

type MENU_ITEM_TYPE (TEXT_WIDTH : COLUMN_TYPE := 10) is
   record
      TEXT          : STRING (1..TEXT_WIDTH);
      OPERATION     : OPERATION_TYPE;
      SELECTABLE    : BOOLEAN;
      HOT_KEY       : KEY_VALUES_TYPE;
      SHORTCUT_KEY  : CHARACTER;
      CHILD         : MENU_LINK;
   end record;

type DATA_TYPE is array (POSITIVE range <>) of MENU_ITEM_TYPE;

type MENU_BASE_TYPE is
   record
      FIRST     : MENU_LINK;
      CURRENT   : MENU_LINK;
      PREVIOUS  : MENU_LINK;
      COLOUR    : COLOUR_TYPE;
   end record;
```

The foreground colours determine how the text of the highlighted menu option will be displayed.

- *TYPE_OF_MENU* : The tag which identifies the type of menu. It can be either a spread menu or a stack menu.
- *PARENT* : Points to the parent menu. For the root menu, this points to null.
- *DATA* : The menu options and all the associated data. The value in LENGTHS determines how many menu options there are and how long the menu window must be.
- *CHOICE* : The present menu choice. The menu selection bar will highlight the corresponding menu option. When an option is chosen, the value of CHOICE is used to determine which menu option was actually chosen.
- *TOP_ROW* : The screen row on which the menu window's top row will be drawn. These values are in global coordinates.

- *LEFT_SIDE* : The screen column on which the menu window's left side will be drawn. These values are in global coordinates.
- *WIDTHS* : The actual width of the window object in which the menu is placed.
- *TEXT* : The text of the menu item which is displayed on the screen. The letter that is highlighted is put between the character "~". The first ~ switches on the highlighting, and the second ~ switches it off. There must be two of these characters for each menu item text. The length of the text is stored in TEXT_WIDTH.
- *OPERATION* : The operation that will be performed by the program if the corresponding menu option is chosen. If this operation code is SUN_MENU or PARENT_MENU, the current menu is removed and the desired menu displayed. If the operation code is not an internal menu code, it is passed out to other objects who it might concern.
- *SELECTABLE* : A flag to show if an option can be selected or not. If an option can be selected, it will be displayed in text colours, otherwise it will be displayed in non-selectable colours.
- *HOT_KEY* : The key code of a hotkey. This code is a function key, or a combination of the alternate, control or shift key pressed with a function key.
- *SHORTCUT_KEY* : The key code of a shortcut key. A shortcut key is the key that corresponds to the highlighted character of the desired menu option. The key code must be that of the uppercase character corresponding to the highlighted character, not the lowercase character. (eg. 71 'G' not 103 'g')
- *CHILD* : Points to a submenu if there is one, or to null otherwise. If there is a submenu, the operation field must be set to SUB_MENU.
- *FIRST* : Points to the first menu in the menu tree data structure. It can also be considered the root menu and is usually a spread menu type.
- *CURRENT* : Points to the current menu that is active on the screen. All input, other than hotkeys, are directed to this menu.
- *PREVIOUS* : Points to the previous menu that was active.
- *COLOUR* : A record data structure containing the colour values needed for the display of the menus.

11.1.4 Methods

Draw_Menu

Determines the type of the current menu, and then instructs the corresponding method to draw the current menu.

```
begin
  if (the type of menu is a spread menu) then
    Draw a spread menu.
```

```
  else
    Draw a stacked menu.
  end if;
end Draw_Menu;
```

Find_Hotkey

Searches the menu structure to find the required hotkey. If the hotkey is found, the OPERATION field of the input record is set to the OPERATION field of the corresponding menu option.

```
begin
  for (the COUNTER in 1 to the last menu option) loop
    if (the operation corresponding to the present menu option = SUB_MENU) then
      Recursively call itself to look in the submenu for the required hotkey.
      if (the OPERATIONS /= OP_DO after returning from the recursively called procedure -
        if needed) then
        Exit from this loop because the hotkey and the corresponding operation to the hotkey
          has been found.
      end if;
    elsif (the hotkey corresponding to the present menu option = search hotkey) then
      if (the present menu option is selectable) then
        Set the OPERATIONS field of the input record = the operation corresponding to the
          present menu option.
      else
        Set the OPERATIONS field of the input record = NON_SELECTABLE. This is done
          to end the search process.
      end if;
      Exit from this loop.
    end if;
  end loop;
end Find_Hotkey;
```

Handle_Input

Determines wether the current menu is a spread menu or a stacked menu. The input is then routed to the appropriate menu method.

```
begin
  if (the type of menu is a spread menu) then
    Send the input to spread object methods.
  else
    Send the input to stack object methods.
  end if;
end Handle_Input;
```

Find_Menu_Option

The menu structure is searched for the required menu option. This will require recursive calling of this procedure if there are submenus in the menu structure.

```
begin
  for (the COUNTER in 1 to last menu option) loop
    if (the operation corresponding to the present menu option = SUB_MENU) then
      Recursively call itself to look in the submenu for the required option.
      if (the OPERATIONS = OP_DONE after returning from the recursively called procedure -
        if needed) then
        Exit from this loop because the option and the corresponding selectable state has been
          found and adjusted.
      end if;
    elsif (the operation corresponding to the present menu option = search operation) then
      Set the selectable state of the present menu option.
      Set the OPERATION field of the input record OP_DONE.
      Exit from this loop.
    end if;
  end loop;
end Find_Menu_Option;
```

Change_Menu_Option

Searches through the menu structure to find the desired menu option. When it is found, it's state is changed to the required state (selectable or not). If it is not found, an exception is raised.

```
begin
  Search the menu structure for the required operation and set the selectable state to the
    required state.
  if (the required operation cannot be found in the menu structure) then
    Raise the exception NO_SUCH_OPERATION_IN_MENUS.
  end if;
end Change_Menu_Option;
```

Redraw

This draws all the menus up to that point to by BASE.CURRENT on the screen.

```
begin
  Copy the main buffer to the screen.
  Set the pointer for the menu to be drawn to the first menu.
  Draw the menu.
  while (the pointer for the menu to be drawn is not the current menu) loop
    Set the pointer for the menu to be drawn to the submenu indexed by this menu's choice field.
    Draw the menu.
  end loop;
```

```
end Redraw;
```

Get_Menu_Operation

Gets the next operation that must be performed by the program from the menus. This procedure returns with the desired menu option. All movement within the menu structure is done within this procedure.

```
begin
  loop
    Get input from the Inputter object.
    case (of the KEY field of the input record) is
      when (all function keys up to F10, including with shift, control and alternate, except F1) =>
        Search the menu structure for the hotkey and return the corresponding operation.
      when others =>
        Send the input to the Handle_Input method.
    end case;
    case (of the OPERATION field of the input record) is
      when (the help was displayed) =>
        Redraw the menus.
      when LEFT_MENU =>
        loop
          if (the choice of the current menu's parent =
            first option of the current menu's parent) then
            Set the choice of the current menu's parent to the last option.
          else
            Set the choice of the current menu's parent to the previous option.
          end if;
          Exit from the loop when the choice of the current menu's parent has a submenu.
        end loop;
        Set the current menu to the submenu indexed by the current menu's parent's choice field.
        Redraw the menus.
      when RIGHT_MENU =>
        loop
          if (the choice of the current menu's parent =
            last option of the current menu's parent) then
            Set the choice of the current menu's parent to the first option.
          else
            Set the choice of the current menu's parent to the next option.
          end if;
          Exit from the loop when the choice of the current menu's parent has a submenu.
        end loop;
        Set the current menu to the submenu indexed by the current menu's parent's choice field.
        Redraw the menus.
      when SUB_MENU | PARENT_MENU =>
        Set the previous menu pointer to the current menu pointer.
        if (the submenu) then
          if (there is actually a submenu) then
            Set the current menu to this submenu.
          end if;
        else
          if (there is actually a parent menu) then
```

```
            Set the current menu to this parent menu.
          end if;
        end if;
        if (the current menu pointer /= the previous menu pointer) then
          if (a submenu must be drawn) then
            Draw the submenu.
          else
            Redraw all the menus.
          end if;
        end if;
      when others =>
        if (the OPERATION field of the input record /= OP_DO) then
          Exit from this loop.
        end if;
    end case;
  end loop;
end Get_Menu_Operation;
```

11.2 Spread Menu

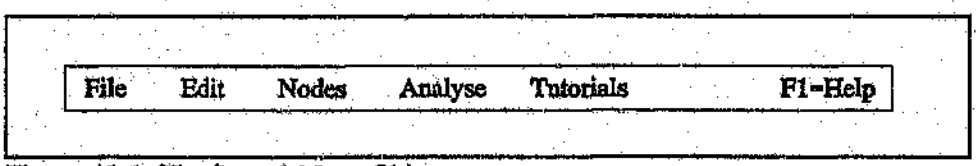

Figure 11-1 The Spread Menu Object

11.2.1 Purpose

The spread menu present the user with a list of options from which one is selected.

11.2.2 Appearance and Behaviour

The menu options are spread out one next to another. The menu is written to the main buffer which is then to the screen. It is assumed that the area in which the menu appears will not be overwritten. All other methods that directly affect the physical screen have been adjusted so that they leave the first line of the physical screen alone. This can be altered.

An option on the menu can be selected in three different ways :

- *Menu Selection Bar* : The menu selection bar is a highlighted bar that appears on the menu. This bar highlights one of the options on the menu. This option will be selected if the *Enter* key is pressed. Certain cursor keys can move the selection bar.
- *Menu Option Shortcut Key* : On the menu, one of the characters for each option will be highlighted. The character shows which key must be pressed on the keyboard to select that

option. Pressing the highlighted character key is equivalent to moving the selection bar to that option and pressing *Enter*.

- *Mouse Pointer* : The mouse can be used to select a menu option. Move the mouse pointer to the desired option and press the left mouse button. The mouse pointer can be anywhere on the menu option to select it.

Key pressed	Affect on the Spread Menu
Cursor left	Moves the selection bar left to the previous option.
Cursor right	Moves the selection bar right to the next option.
Home	Moves the selection bar to the first option.
End	Moves the selection bar to the last option.
Enter	Selects the option that the selection bar is highlighting.

Table 11-1 Spread Menu Key Definition

11.2 Methods

Draw_Menu

Draws a spread menu. The menu is written to the main buffer which is then written to the screen.

```
begin
  Draw the menu line across the first row of the physical screen.
  for (the COUNTER in 1 to length of the menu) loop
    Write the menu option corresponding to the value of the COUNTER.
  end loop;
  Draw the menu line bar
end Draw_Menu;
```

Handle_Input

Handles the input for a spread menu. If the left mouse button was pressed when the mouse pointer was on a selectable menu option, the OPERATION field of the input record is set to the corresponding menu option. If a key was pressed, a check is made to see if it is a valid key. If it is, the corresponding menu operation is carried out.

```
begin
  if (there is a mouse and the left button is depressed) then
    if (the mouse pointer is on the menu row) then
      Set the menu choice to the option the mouse pointer is on.
```

```
        if (the chosen menu option is selectable) then
          Set the OPERATION field of the input record equal to the operation corresponding
            to the chosen menu option.
        end if;
      end if;
    else
      case (of the KEY field of the input record) is
        when (Enter) =>
          if (the chosen menu option is selectable) then
            Set the OPERATION field of the input record equal to the operation corresponding
              to the chosen menu option.
          end if;
        when (cursor left) =>
          if (the present menu choice = first option) then
            Set the menu choice to the last menu option.
          else
            Set the menu choice to the previous menu option.
          end if;
          Display the menu bar at the present menu choice.
        when (cursor right) =>
          if (the present menu choice = last option) then
            Set the menu choice to the first menu option.
          else
            Set the menu choice to the next menu option.
          end if;
          Display the menu bar at the present menu choice.
        when (Home) =>
          Move the menu bar to the first menu option.
        when (End) =>
          Move the menu bar to the last menu option.
        when (upper or lower case letters) =>
          for (the COUNTER in 1 to the length of the menu) loop
            if (the KEY field of the input record = the menu's shortcut key (indexed by
              counter)) then
              if (the chosen menu option is selectable) then
                Set the OPERATION field of the input record equal to the operation
                  corresponding to the chosen menu option.
                Set the menu choice to the option chosen.
              end if;
            end if;
          end loop;
        when KB_F1 =>
          Instruct the help object to display the required help.
        when others =>
          Do nothing.
      end case;
    end if;
end Handle_Input;
```

11.3 Stacked Menu

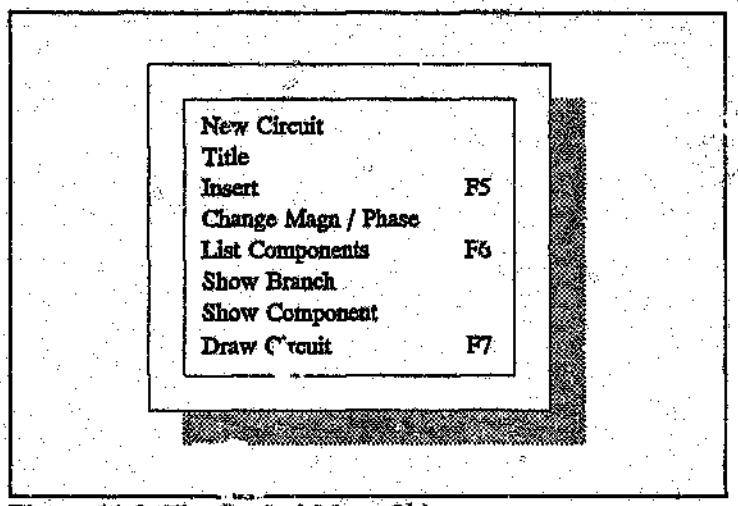

Figure 11-2 The Stacked Menu Object

11.3.1 Purpose

The stack menu presents the user with a list of options from which one is selected.

11.3.2 Appearance and Behavior

The menu options are displayed one under another in the form of a stack. The menu can be placed anywhere on the physical screen. To prevent the menu from destroying any information that is already on the screen, the menu uses the window buffer. The following procedure is followed :

- Set up the window buffer. This includes clearing and framing it.
- Copy the part of the physical screen that is going to be overwritten to a separate buffer. (Required by method 1)
- Draw the menu in the window buffer. The order of execution between the previous point and this one can be interchanged.
- Copy the window buffer to the physical screen.
- Make a shadow for the menu.

Once the menu option has been chosen, or another menu is required, using method 1 :

- Remove the shadow.
- Copy the contents of the separate buffer back to the physical screen. It will appear as though the menu as been removed.

Once the menu option has been chosen, or another menu is required, using method 2 :

- If the required menu is a submenu, just draw it in the window buffer and copy the window

buffer to the screen.

- If the required menu is the parent menu (ie. exiting this menu), then copy the main buffer to the screen to remove all the menus. Redraw all the menus up to the parent menu.

An option on the menu can be selected in four different ways :

- *Menu Selection Bar* : The menu selection bar is a highlighted bar that appears on the menu. This bar highlights one of the options on the menu. This option will be selected if the *Enter* key is pressed. Certain cursor keys can move the selection bar.
- *Menu Option Shortcut Key* : On the menu, one of the characters for each option will be highlighted. The character shows which key must be pressed on the keyboard to select that option. Pressing the highlighted character key is equivalent to moving the selection bar to that option and pressing *Enter*.
- *Menu Option Hotkey* : Next to some of the menu items, there is a key description eg. *F2*. This method differs from the shortcut key in that the option will be selected even if a different menu is on the screen when this key combination is pressed.
- *Mouse Pointer* : The mouse can be used to select a menu option. Move the mouse pointer to the desired option and press the left mouse button. The mouse pointer can be anywhere on the menu row of the desired option to select it.

Key pressed	Affect on the Stack Menu
Cursor up	Moves the selection bar up to the previous option.
Cursor down	Moves the selection bar down to the next option.
Cursor left	Moves to the previous menu on the same level.
Cursor right	Moves to the next menu on the same level.
Home	Moves the selection bar to the first option.
End	Moves the selection bar to the last option.
Enter	Selects the option that the selection bar is highlighting.

Table 11-2 Stack Menu Key Definition

11.3.3 Methods

Draw_Menu

Draws a stack menu. This requires the use of the window buffer. The options are displayed one below another.

begin
Set up the buffer to the size needed for the menu.

```
  Clear and frame the buffer.
  for (the COUNTER in 1 to the length of the menu) loop
    if (the present menu option is selectable) then
      Write this option to the buffer in menu text colours.
    else
      Write this option to the buffer in non-selectable colours.
    end if;
  end loop;
  Write the menu to the window buffer.
  Copy the window buffer to the physical screen.
  Display the menu bar on the screen.
  Make a shadow for the menu.
end Draw_Menu;
```

Handle_Input

Handles the input for a stack menu. If the left mouse button was pressed when the mouse pointer was on a selectable menu option, the OPERATION field of the input record is set to the corresponding menu option. If a key was pressed, a check is made to see if it is a valid key. If it is, the corresponding menu operation is carried out.

```
begin
  if (there is a mouse and the left button is depressed and this is a new depression) then
    if (the mouse pointer is on the menu) then
      Set the menu choice to the option the mouse pointer is on.
      if (the chosen menu option is selectable) then
        Set the OPERATION field of the input record equal to the operation corresponding to
          the chosen menu option.
      end if;
    else
      Exit from this menu and go to the parent menu by setting the OPERATION field of the
        input record to PARENT_MENU.
    end if;
    Clear the depression flag because input has been handled and only when the left button is
      released, can a new mouse input be accepted.
  else
    case (of the KEY field of the input record) is
      when (Enter) =>
        if (the chosen menu option is selectable) then
          Set the OPERATION field of the input record equal to the operation corresponding
            to the chosen menu option.
        end if;
      when (cursor up) =>
        if (the present menu choice = first option) then
          Set the menu choice to the last menu option.
        else
          Set the menu choice to the previous menu option.
        end if;
        Display the menu bar at the present menu choice.
      when (cursor down) =>
        if (the present menu choice = last option) then
          Set the menu choice to the first menu option.
```

```
          else
            Set the menu choice to the next menu option.
          end if;
          Display the menu bar at the present menu choice.
        when (cursor right) =>
          Set the OPERATION field of the input record to RIGHT_MENU.
        when (cursor left) =>
          Set the OPERATION field of the input record to LEFT_MENU.
        when (Home) =>
          Move the menu bar to the first menu option.
        when (End) =>
          Move the menu bar to the last menu option.
        when (upper or lower case letters) =>
          for (the COUNTER in 1 to the length of the menu) loop
            if (the KEY field of the input record = the menu's shortcut key (indexed by
              counter)) then
              if (the chosen menu item is selectable) then
                Set the OPERATION field of the input record equal to the operation
                  corresponding to the chosen menu option.
                Set the menu choice to the option chosen.
              end if;
            end if;
          end loop;
        when KB_F1 =>
          Copy the main buffer to the screen.
          Instruct the Help object to display the required help.
          redraw the all the menus.
        when (Esc) =>
          Exit this menu and go to the parent menu by setting the OPERATION field of the
            input record to PARENT_MENU.
        when others => Do nothing.
      end case;
      if (the left button of the mouse has been released) then
        Set the depressed flag becau ; the next time the left button is pressed on the mouse, it
          will be considered a new mouse input.
      end if;
    end if;
  end Handle_Input;
```

CHAPTER 12

TUTORIAL SUPPORT PROGRAM

12.1 Purpose

The tutorial support program is used to make and modify all the tutorial screens for the Elector program. The tutorial screens are created or modified on the screen and then written to a disk file ELECTOR.TUT. This can be used for any program that uses the same method as Elector to access the tutorial screens.

12.2 Behaviour

The program is similar to a graphic paint program, except that ASCII characters are used. The foreground and background colours can be set. The surface that can be *painted* is initially blue. The program is a WYSIWYG program (what you see is what you get). At the bottom left corner there is an indicator that shows the present foreground colour, background colour and blinking settings. The next character entered will have these attributes.

Key	Purpose
Down Arrow	Move cursor down.
Up Arrow	Move cursor up.
Left Arrow	Move cursor left.
Right Arrow	Move cursor right.
F1	Increment the foreground colour. There are 16 colours to choose from.
F2	Increment the background colour. There are 8 colours to choose from.
F3	Toggle blinking attribute
F5	Load a tutorial screen.
F6	Save a tutorial screen.
F10	Exit the program without saving the present tutorial screen.

Table 12-1 Tutorial Support Program Key Definition

Any of the *ascii* characters may be entered. Characters that aren't represented on the keyboard can be entered by holding down the *Alt* key and typing in the ASCII code on the number keypad.

12.3 Tutorial Index

When the program is started, the tutorial index is prompted for. This index is used to search for a particular tutorial screen. The index you enter should correspond to one that exists in Elector.

Tutorial Group	Tutorial Option	Index
Elector Information	What is it for?	1001
	User Interface	1002
	Subscreens	1003
	Windows	1004
	Buttons	1005
	Input Lines	1006
	Pick Lists	1007
	Help	1008
Circuit Modelling	Rules 1	1021
	Rules 2	1022
	Rules 3	1023
	Design 1	1024
	Design 2	1025
Elec Engineering	Definitions	1041
	Passive Components	1042
	Ideal Sources	1043
	Real Sources	1044
	Electrical Power	1045
	Kirchhoff's Laws	1046
	Thevenin	1047
	Frequency Domain	1048
	Impedance	1049
	AC Power	1050
	Power Factor	1051
	Three Phase	1052
	Resonance	1053
Equivalent Circuits	Transistor	1061
	Op Amp	1062
	Transformer	1063
	Induction Motor	1064
	Synchronous Motor	1065
	Fluid	1066
	Mechanical 1	1067
	Mechanical 2	1068
	Thermal	1069

Table 12-2 Tutorial Options used in Elector

12.4 Creating Tutorial Screens

- Press *F5* to load.
- The tutorial index will be prompted for. Enter an index that does not already exist in the ELECTOR.TUT file, but corresponds to an index used by Elector. If the file does not exist, then any index that corresponds to an index used in Elector can be chosen.
- An error message will be displayed informing you that the chosen tutorial screen does not exist. This is what you want, because you are creating a new tutorial screen.
- If you don't get an error message, load another tutorial screen. If you don't, you will be modifying an existing tutorial screen.
- Press the *Ok* button on the ERROR subscreen.
- Create the tutorial screen.
- Press *F6* to save the screen. The screen is saved to the disk file ELECTOR.TUT using the index that was specified last.

12.5 Modifying Tutorial Screens

- Press *F5* to load.
- The tutorial index will be prompted for. Enter an index that already exists in the ELECTOR.TUT file. If you are not sure if such an index exists, enter it and see if an error message is displayed. If one is, then that tutorial screen does not exist. Simply load another tutorial subscreen by entering another index.
- The tutorial screen will appear on the main screen.
- Modify the tutorial screen.
- Press *F6* to save the screen. The screen is saved to the disk file ELECTOR.TUT using the index that was specified last. The previous corresponding tutorial screen will be overwritten by this one.

CHAPTER 13

TESTING AND VALIDATING ELECTOR

Elector was tested in two phases. In the first phase, each module was tested using separate test programs. Testing strategies including using common values, boundary values, error values, etc.

Test	Packages Tested
1	Text and Maths
2	Dos_Environment
3	Inputter and Screen_Objects
4	Screens and Help
5	Menus
6	Netlists and Netlist_Supporter
7	Analysers and Central_Solver
8	Optimise
9	Graphs, Drawer and Plotter
10	Monte

Table 13-1 Test Subprograms

In the second phase, integration testing was undertaken with a number of test circuits.

Elecsim was also used for checking the validity of Elector. There are some differences in the generated results because Elecsim uses 32 bits (4 bytes) for real numbers. This gives about 7 significant digits in the mantissa. Elector uses 64 bits (8 bytes) for real numbers. This gives about 15 significant digits. Any numbers that are different from the eighth significant digit will not be visible in Elecsim.

Circuit File	Type of Circuit
3PH-SSDL.NTL	3 Phase star source delta load.
BAT_BF.NTL	Battocleti band pass filter.
BESL.NTL	Bessel all pass - phase shift (delay) filter.
BR_FIL.NTL	Band reject filter.
BUTTER3.NTL	Butterworth 3rd order low pass filter.
CE-AMP.NTL	Single stage common emitter amplifier.
CE-AMP2.NTL	Single stage common emitter amplifier.
CH5_FIL.NTL	Chebyshev 5th order high pass filter.
CLP_FIL.NTL	Cauer low pass filter.
D-RF-AMP.NTL	Differential 3rd order low pass filter.
DRAW.NTL	Draw test for different combinations of components in each branch.
DRAW2.NTL	Draw test for connection between each and every node.
INST.NTL	Instrument amplifier.
PLL-RC.NTL	Parallel RC impedance.
RES-HA.NTL	Resonant harmonic analyser.
RLC.NTL	RLC series.

Table 13-2 Test Circuits

REFERENCES

Abel, P. (1991) *IBM PC Assembly Language and Programming, 2nd Ed*, Prentice-Hall, United States of America, 1991

Borland (1990) *Turbo Pascal Version 6 : Turbo Vision Gulde*, Borland International, Inc, Scotts Valley, United States of America, 1990

Meridian (1990) *Meridian Ada 4.1*, Meridian Software Systems, United States of America, 1990

Schach, S.R. (1990) *Software Engineering*, Alssen Associates, United States of America, 1990

Sommerville, I. and Morrison, R. (1987) *Software Development with Ada*, Addison-Wesley, Great Britain, 1987,