COSC 460
Honours Project Report
Department of Computer Science
University of Canterbury

A Digital Logic Simulator for the Apple Macintosh

Supervisor: Dr. M. A. Maclean

Michael Wild

October 1985

Contents

1. Introduction 3

2. The Macintosh User Interface and Macapp 5

2.1 The User Interface 5
2.2 MacApp 6

3. The Circuit View 8

3.1 Outline 8
3.2 The Palette 9
3.3 Components 9
3.4 Wires 11
3.5 Symbol Selection 13
3.6 Dragging of Symbols 14
3.7 Naming of Components 15
3.8 Deleting Components 15

4. Simulation and the Trace View 17

4.1 Outline 17
4.2 The Simulation Algorithm 17

4.3 Running the Simulator 18
4.4 Signal Sources 19
4.5 Probes 20
4.6 The Trace View 20

5. Summary and Conclusions 22

6. References 24

Appendix A. Program Listing 26

1. Introduction

The aim of this project was to design a digital logic simulator to run on the Apple Macintosh. It was envisaged that users would be able to design a network of gates, flip-flops and signal sources interactively, then simulate the operation of the circuit. If the circuit did not behave as expected, it could be changed and a simulation run on the updated circuit. The cost of developing circuits using hardware is high; the use of simulation makes development a much cheaper process.

The interactive approach used in the system presented in this report offers several advantages. The user builds a circuit by selecting components from a "palette" and placing them on the screen. The components are connected by drawing wires between them. The layout of a circuit is clearly visible. If mistakes are made in the design, it is a simple matter to alter the circuit. The operation of a circuit can be simulated at any time, and the results of doing so are available immediately, displayed in an easily understandable form.

There are two parts to this project. The first of these is the interactive development of the circuit. The main problems here are the placing of components on the screen and the interconnection of components by wires. Simulation of the circuit is the other major part of the project. An efficient algorithm for simulation of logic networks is important here. It is also necessary to provide a trace of the outputs from the circuit so that the user can see what is happening. By placing a probe on a wire, the user can monitor the value of the signals passing through that wire.

One of the reasons that the Apple Macintosh was chosen for the simulator was its user interface. The Macintosh is highly suited for use in interactive applications. Section 2 of this report introduces the more important aspects of the interface. It also describes MacApp, an application development tool used extensively in the implementation of the system.

The interactive development of circuits is the subject of section 3 of the report. The methods used for placing and drawing components are described. Probably the most difficult aspect of this part of the system is the connection of individual components with wires.

Section 4 is about the simulation of networks and the display of the

resulting output. The algorithm used for simulation is presented, and the display of output tracing is discussed in this section.

Not all the functions discussed in the report were implemented fully. Section 5 gives a summary of the work still to be done, and some conclusions are presented. Appendix A gives a listing of the declaration part of the program, showing all the procedures used. Comments indicate the purpose of each procedure.

2. The Macintosh User Interface and MacApp

2.1 The User Interface

The Macintosh has a highly developed user interface that makes it ideal for an application such as the one described here. Information is organised into "documents", a concept similar to that of a file. A document contains a representation of the data used in an application. The user is presented with a view of part of a document on the screen, and can manipulate the data in the view. The main tool for the manipulation of objects on the screen is the mouse, a pointing device used to move a cursor, or pointer, around the screen. The mouse is used to select and move objects, which may then be operated on by some set of commands. Probably the most visible aspects of the user interface, used in nearly every Macintosh application, are windows and menus.

Windows are objects used to display information. A window will typically show the data in a part of a document. The user can move the window over the document by "scrolling" the display within the window. There may be any number of windows open on the screen at once. They can be moved around the screen and changed in size, and some may overlap others. The front most is known as the "active" window, and all user actions and commands are handled by the part of the application that owns this window.

The top part of the screen is used to display the "menu bar". This contains the titles of several menus. To see what is in a menu, the user positions the pointer over the menu and presses the mouse button. The menu will appear. By moving the pointer down the menu the user can select an item from the menu. The action associated with that item will be executed immediately.

Since not all actions are relevant all the time, some of the items in a menu may be disabled. While they still appear in the menu, disabled items are displayed in a different style and cannot be selected. For example, a command to delete a set of selected objects would be disabled when there is nothing selected in the active window. Some groups of items represent different states of the same thing. For instance, a window may be able to display data in either binary or hexadecimal format. Both would appear in the menu; to indicate which is being used, the relevant item is "checked". A small tick will appear beside that item in the menu.

In a Macintosh application, the user will usually select some part of

the document and manipulate it in some way. The objects selected within the active window are known as the "active selection" and are usually highlighted in some way. The next command given by the user will generally operate on this selection. For instance, to delete objects from the document the required objects are first selected using the mouse, then the delete command is chosen from a menu. The command operates on the active selection, and the objects disappear from the screen. When placing new objects on the screen, the active selection is often just an insertion point. In placing components, the insertion point follows the pointer, and a rectangle showing the area affected is displayed around this point.

2.2 MacApp

Since most applications use common objects in a standard way, Apple has developed MacApp. This is a development tool that implements most of the basic features required when writing an application. Code for reading and writing data from files is supplied, as is code for printing the contents of a document. The creation of windows and menus is handled generically by the system. By writing code on top of MacApp it is much easier to build an application than it would be to build from scratch.

Macapp is written in Object Pascal, an extension of Pascal designed for object oriented programming. A new structured type, the "object", is added to the language. This is similar to a record type in that it has fields for data. However, a member of an object type also has several "methods", procedures used for manipulating the fields of the object. Many standard objects are provided, but it is possible to override the way in which these are implemented as well as adding new types of objects.

There are several standard kinds of object in MacApp. Document objects contain the underlying data structures of the document. In the application described in this report, the document object has a list of all components, and there are pointers that represent the way in which the components are connected to each other. Window objects are provided to implement the windows described above. The most important objects from the users point of view are views and commands.

A view is a way of looking at the data in a document. Views appear within a window associated with the document. Some documents may have several different views. One of the methods of a view object is "Draw", a procedure to draw the view into the window. The user can look at the data in different ways by using different view objects to draw it. The logic simulator presented here has two views. The first is the Circuit View, a

simple circuit diagram of the network. This shows the layout of components and the wires connecting them. The other kind of view is the Trace View. This shows the values output when the circuit is simulated. The circuit view is used while building the circuit, the trace view when the circuit is simulated. The two views appear in different windows, and the trace view need not be on the screen when it is not being used.

Command objects are created in response to user actions. A command may be a key command (from typing a key), a mouse command (pressing the mouse button), or a menu command (selecting a menu item). When a command is issued by the user, a command object is created. The methods of this object carry out the appropriate action. The command object is also responsible for undoing the action if the user chooses to do so. This usually involves holding the state of the document before the command was issued until it is no longer possible to undo it. Examples of commands are deleting components in the circuit view, and starting the simulation of a circuit. The menus are set up to reflect the appropriate commands at any time.

In writing an application using MacApp, it is necessary to write code for the views and commands peculiar to the application. The document object must be customised for the data structures used. Little else needs to be done. Some of the standard objects may need to be changed slightly. For instance, it was necessary to write a method for creating a window for the circuit view, since this window contains a palette of symbols in its left hand side. MacApp incorporates any new objects into the application, and handles user input itself. The programmer can work on a higher level, concentrating on the code specific to the application, without having to worry about low level events.

3. The Circuit View

3.1 Outline

The circuit view is the part of the system that allows the user to build a circuit interactively. A diagram of the logic network is presented in the window, and components and wires may be added and removed. An example of a circuit view window is given below in figure 3.1. Inside the left-hand edge of the window is a palette of symbols, one of which is selected. The selected symbol is placed in the diagram by using the mouse, and symbols are connected by drawing wires between them. It is also possible to cut components from the circuit. This is done by selecting the components to be deleted and then selecting the menu command "cut". The active selection can be moved by dragging it across the screen. The mouse button is pressed when the pointer is on the selection, and the selection is moved to wherever the button is released. When a component is first placed in the circuit, it becomes the current selection by default so that the user can see where it is. The symbol can be removed just after it has been placed by selecting the "undo" command from the menu. Typing a backspace on the keyboard also invokes an undo command.

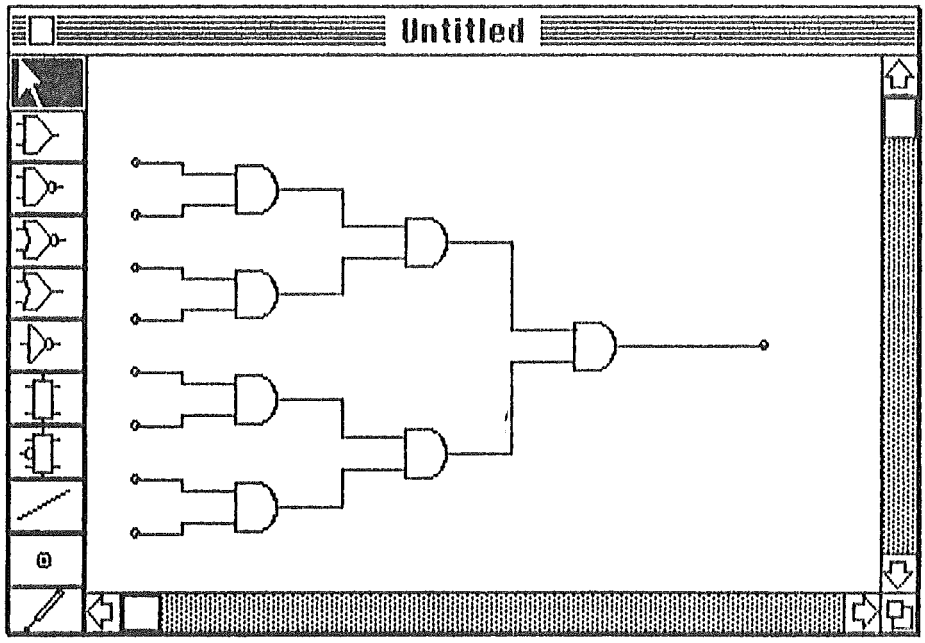

fig. 3.1 Sample circuit view window

Several sample programs were supplied with MacApp. One of these was "Draw", a simple graphics application that allowed the user to place

shapes on the screen and to move them around. This program was used as a starting point for the application. New types of symbols were defined, and the data structures were changed to allow for wires between the symbols. The cut command for deleting symbols was also added. Using this program as a base saved a lot of time, since the basic methods for positioning and moving symbols were already written.

For drawing objects on the screen, calls are made to a collection of graphics routines on the Macintosh known as QuickDraw. Procedures are available for drawing lines, circles and rectangles, as well as for writing text strings and highlighting areas of the screen. With the bit-mapped display used on the Macintosh these routines are fast and drawing of symbols can be done easily.

In the rest of this section the circuit view is described in more detail. The parts of the view are presented, and the placing of symbols and drawing of wires are discussed. The commands associated with the view are also described.

3.2 The Palette

In the left hand edge of the window is a collection of symbols known as the palette. To select one of these symbols the mouse button is clicked while the pointer is over the box containing the symbol. This symbol will be highlighted and is known as the current palette selection. When the user places a component in the circuit view this is the symbol that will appear. Wires also appear in the palette. When the wire symbol is selected the user can draw wires between components. The drawing of wires is discussed later in the report. At the top of the palette is an arrow symbol. This is different from the other symbols in the palette in that it does not represent a circuit symbol. When this is selected, the mouse can be used to select and move components. Selection and dragging of objects in the circuit view are also discussed later.

3.3 Components

When a component symbol is selected in the palette, that symbol can be placed anywhere in the circuit diagram by using the mouse. When the mouse button is pressed a box appears around the pointer. This is the extent rectangle of the symbol and shows the area in which the component

will be drawn. If the mouse is moved this outline follows the pointer so that the user can see where the symbol will be placed and avoid overlapping existing components. When the button is released the component will be placed at that position. A ShapeSketcher command object is created to track the mouse and set up the component object when the button is released. This is responsible for making the extent rectangle follow the pointer, and for creating a new component object and placing it in the document data structures.

So that components can be lined up easily in the diagram there are invisible grid points at regular intervals across the screen. When a component is placed, its centre (the point where the mouse was released) is aligned with the grid. The symbol is then drawn by calls to graphics routines. The top left corner of the extent rectangle is used as a reference point and drawing is done relative to this point. All graphics operations are done in what are known as "master coordinates", which take this point as the origin.

Currently the components available are and, nand, or, nor and not gates, D flip-flops and JK flip-flops. There are also pins, which can be either input points to the circuit acting as signal sources, or output points; wires, used for connecting components; and probes, used in tracing the operation of the circuit. Some of these components are similar in many of their characteristics. And, or, nand and nor gates all have similar shapes, and wires connect to them in the same places. JK flip-flops are similar to D flip-flops. For this reason the object types TGate and TFlipFlop are used. These are templates for the gates and flip flops. Part of the symbol is drawn by the drawing methods of the template object, the rest by the methods of the symbol itself. The main purpose of these templates is to reduce the amount of repetition of code.

When a new symbol is placed in the view it must also be inserted properly into the data structures for the view. The document that owns the view uses a doubly linked list of all components in the view, as in figure 3.2. When a symbol is created it is inserted at the end of the list. There is no ordering of symbols in the list at present, but it is possible that some ordering of components may be desirable when simulating large circuits. The most efficient order will probably have to be found by extensive testing.

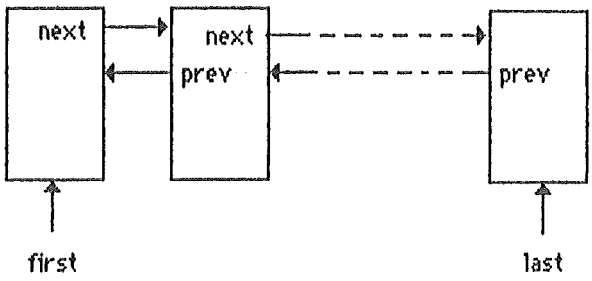

figure 3.2 Document data structure

So that wires can be joined to a component each symbol has several "connection points". These are points to which wires may be connected. The connection points for a two input and gate are shown in figure 3.3. The connection points for all types of two input gates are the same, so these are defined in TGate. At the moment these gates can have only two outputs, a severe limitation. This problem is discussed briefly in section 5.

figure 3.3 Connection points

When the user tries to join a wire to a component as described in the next section, there must be some way of knowing which connection point the wire should use. All symbols have a method called FindConnectPoint that returns the closest connection point to where the mouse button was released. This is done by finding the distances to all the points, and choosing the one with the smallest distance. The method ConnectWire is then used to join the wire to this point.

3.4 Wires

Wires are used to connect the components in the circuit to form a logic network. When the operation of the circuit is simulated, they form the path for signals propagating through the network. Wires start and finish on the connection points of components, and a wire can also connect to another wire. Here all points on other wires are possible connection points, and we just connect to the closest one.

A single wire is made up of several vertical and horizontal line

segments. Each of these has a start point and a finish point; the finish of one will be the start of the next. The end of the last segment will be at a connection point. A wire object has a pointer to a list of these line segment objects, as shown in figure 3.4. Note that the segments are not part of the main list of objects in the system, but rather a sub-list within the wire object. A list of this type is known as an object list as each entry in the list has a pointer to some object, in this case a line segment. To draw the wire, the list is traversed and each line segment drawn.

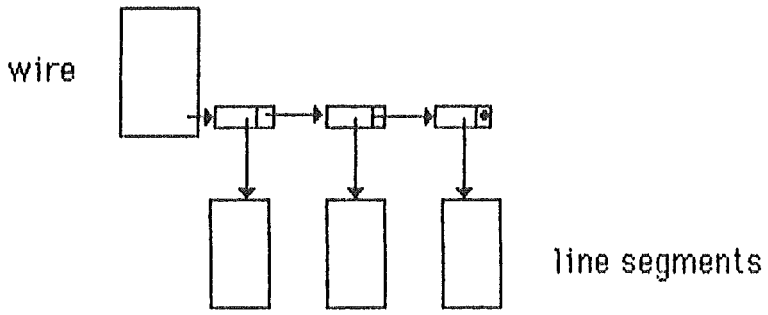

figure 3.4 Wire data structure

The endpoints of the wire are where it joins to components. The user clicks the mouse button at the first endpoint to start drawing the wire. When the button is pressed at the second endpoint the end of the wire has been reached and it is linked into the data structures. A WireSketcher command object is created when the user starts to draw the wire, and the methods of this object control the drawing of the wire.

There are two possible approaches to the drawing of the wires. The first of these is automatic routing, where the user selects the two endpoints and the system finds a path for the wire, avoiding obstacles such as other symbols. This frees users from decisions about the routing of wires, but at the same time restricts them to the routing made by the system. The second approach is manual routing. Here the user builds up the wire by clicking the mouse button at the end of each line segment. This allows more flexible routing at the cost of having to specify all the turning points in the wire. As each line segment is created it is put in the lines list of the wire.

Probably the best solution is to use a combination of these two approaches. If users wish to route their own wires, they can turn automatic routing off and draw the wires by hand. However, at this stage the manual routing has not yet been implemented, and the procedure used

for automatic routing is very primitive. All it does is to draw a line segment across half way to the other endpoint, come down to the same level, and then go the rest of the way across. This is satisfactory for testing purposes but needs to be improved.

To maintain the logical structure of the network each component object has pointers to all the wires connected to it. The wire also has pointers to the components that it joins. Since several other wires may be joined onto a wire, it also keeps a list of all the wires that are connected to it. By following the series of pointers it is possible to determine how the components are connected to each other. This is important when the operation of the network is to be simulated. Figure 3.5 shows the pointers for a simple example. The top wire (1) has two other wires joined to it, and maintains a list with pointers to both of these. The other two wires have pointers to the first one, since it is also joined to them.

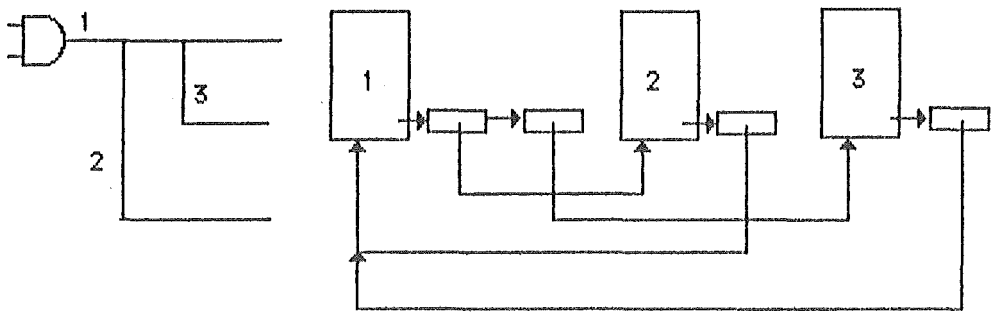

figure 3.5 Wire interconnection data structure

When connecting one wire to another, the user is not allowed to join the outputs of two components together. If outputs are connected, problems will occur when the two components try to drive the wire to different logic states. If the user does attempt to join outputs, the system will not allow it and a warning will appear.

3.5 Symbol Selection

Most commands operate on the active selection, the part of the view that has been selected by the user. When the arrow symbol in the palette is selected the user can select symbols and groups of symbols in the circuit. A component is selected by clicking the mouse button when the pointer is over it. Groups of symbols are selected by clicking on them individually while the shift key is held down, or by defining a selection area - the

mouse button is pressed, the pointer moved to the opposite corner of the area, and the button released. All symbols whose extent rectangles intersect this area are selected. A special type of command object, the ShapeSelector, is created for an area selection. This is responsible for following the movement of the pointer and looking for all intersecting symbols once the button has been released. If the mouse button is clicked on a symbol the selection is handled without having to create a new command object. To see if the button has been pressed on a symbol, the document object has a method called HitDetect. This goes through the list of symbols, and returns a pointer to the first one that it finds that is close to the clicked point. The mouse does not have to be clicked exactly on top of the symbol, since HitDetect defines a small rectangle around this point and looks for an intersection with each extent rectangle.

The symbols in the active selection are highlighted by inverting their extent rectangles. All white parts within the rectangle become black, and all black parts white. This allows the user to see what is selected. Once the selection has been defined the selected symbols can be manipulated by various commands.

3.6 Dragging of Symbols

The symbols in the active selection can be moved within the view by pressing the mouse button on one of them and releasing it at the desired new position. A large rectangle follows the pointer; this shows the area in which the symbols will be redrawn. A ShapeDragger command object is created to carry out the required actions. When the mouse is released the distance moved is calculated, adjusted to line up with the grid, and the symbols are drawn in their new positions. The command can be undone by selecting "undo" from the menus; the symbols will be moved back to their original positions.

The main problem arising in the dragging of symbols concerns wires leaving the selection. They cannot simply be moved, because one of the endpoints would no longer be at a component connection point. There are two solutions to this problem. The first is to stop users from dragging symbols in this situation. If they try to do it a warning will be given. The second solution is to delete all the wires that are affected. The first is the easier to implement, and is the one used at the moment; a future version of the system could include the second option.

3.7 Naming of Components

When simulating the operation of the circuit it is essential to know which components are producing the outputs shown in the trace. For this reason there is provision for giving names to components and wires. To avoid confusion it is only possible to name one component at a time. A menu item is used to do this, and it is only enabled when there is exactly one component in the selection.

When the name command is selected from the menu a NameCmd object is created. This asks the user to type in the name. When the return key is typed the end of the name has been reached, and it becomes the new name of the symbol. Not all components have to have names, and there is nothing to make sure that a name is unique. It was felt that the user should be given as much freedom as possible in this regard.

3.8 Deleting Components

As well as placing new symbols in the view, users may also want to remove existing ones. To do this, the "cut" menu item is chosen, creating a CutCmd command object to carry out the operation. The components and wires in the active selection are deleted from the view, and the data structures are updated. To avoid any "dangling" wires, all the wires that leave the selection are also removed. As is the case with most commands, choosing the "undo" menu item will reverse the operation.

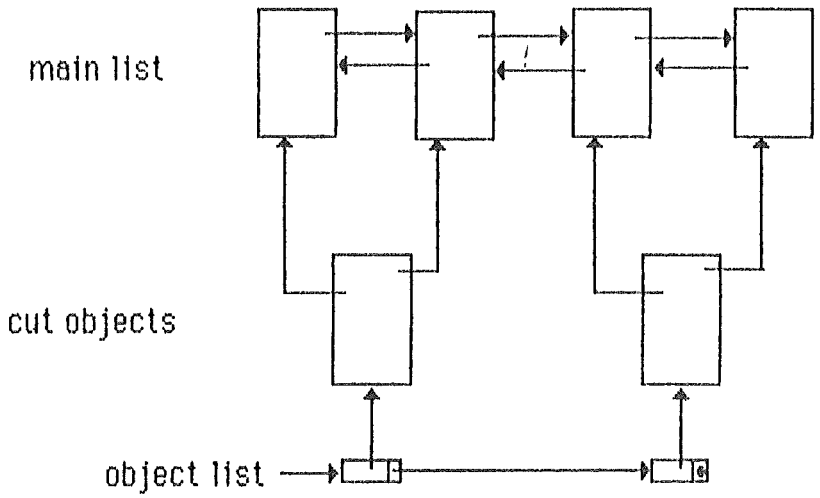

figure 3.6 Cut object list

The first thing the CutCmd object does is to build a list of all symbol objects to be deleted. This is an object list similar to the ones used for the line segments of a wire. An example is shown in figure 3.6. The objects in the list are removed from the main document list, but still keep their references into the main list.

After the symbols have been removed from the list the affected parts of the view will be redrawn. If the command is undone, all that is necessary is to change the pointers back to their original values. When the display is redrawn this time the symbols will reappear.

4. Simulation and the Trace View

4.1 Outline

Once a circuit has been built using the circuit view, the operation of that circuit can be simulated. Inputs are taken from signal sources specified by the user, and the new state of the system is determined. This is then repeated for the next set of input values. The simulation can be halted at any time and later restarted; it can also be be done one step at a time.

The results of the simulation appear in a window showing a trace view. Probes are placed on wires in the circuit view, and the values of signals passing through the wires are monitored. These values are shown in the trace view, usually in a waveform format. The trace is drawn as the simulation proceeds and the user can scroll through the view at any time. This allows the user to see exactly how the circuit behaves.

This part of the project has not been completed. This section of the report presents it as it would have been implemented had there been time. Much of the code has been written in a skeleton form, but has not been tested. This section first presents the algorithm used for the simulation. The operation of this algorithm is discussed in detail. The use of signal sources to give inputs to the circuit is described next. Finally the use of probes and the trace view are discussed.

4.2 The Simulation Algorithm

As mentioned above, the simulation of a circuit proceeds in a series of steps. At the start of each step signals are introduced to the inputs of the circuit, and these are allowed to propagate through the network before the start of the next step. This approach is adapted from those taken by McCusker[2] and McDermott [3].

In a hardware circuit each logic device produces a delay , so that a signal may take some time to reach the outputs of the circuit. To allow signals to propagate correctly each simulation step is broken down into a number of "gate cycles". In each of these cycles all the gates check to see if their outputs should be changed. This allows for the proper operation of feedback loops, where the outputs of a component are later fed back to its inputs. Since there is a delay between the time the inputs to a component

change and the time that the new output values appear, there is no problem caused by a device altering its own inputs. If no outputs in the circuit change in a cycle, all signals must have propagated as far as they can and the end of this step has been reached.

All the component objects in the system have a method called "Transform" used to perform the logic function of the component. This method takes the values of the inputs to the circuit and calculates the new values of the outputs. For example, the transform method for an "and" gate will take the logical and of the inputs of the gate. A three level logic system is used, with values being 'high', 'low' or 'undefined'. A component with undefined inputs may have undefined outputs. For instance "and-ing" any signal with undefined gives an undefined value, while "or-ing" high with undefined gives a high value.

Once the new output values of a component have been calculated by a transform method, those values must be sent through the wires connected to the outputs. This is done by calling the method SetSignal State for each of the wires. This sets the state of the wire to the new value. All other wires connected to the wire will also have their states set. The "state" of a wire is the value of the signal in the wire at the end of a step. This value will be used in the next step of the simulation as described below. SetSignalState is called only if the value of the output is changing. This is done to reduce the number of procedure calls made and speed up the operation of the algorithm. There is no delay involved in setting the state of a wire.

In a single gate cycle transform should be called for all components simultaneously. This assumes that all logic devices produce the same delay. Since it is not possible in practice to transform all devices simultaneously each wire object holds two values. The "current" state is the state that the wire was in at the end of the last gate cycle; the "next" state is the value that the wire will have at the end of the current cycle. The transform methods take the current values of the inputs and operate on these values to give the next state of each output wire. At the start of each cycle the current values are updated by replacing them with the next values from the previous cycle.

4.3 Running the Simulator

To start simulating the user chooses the "Run" option from the "Simulate" menu. To allow MacApp to handle all events normally, a new

step of the simulation is started only if there are no other events waiting to be serviced. When there are no events the system is said to be idle. A method known as "DoIdle" is called continually when the system is in this state. It performs various tasks such as setting up menus. This has been changed so that if the system is simulating then a procedure is called to perform one step. Since DoIdle is called repeatedly as long as the user does nothing, this should not slow the operation of the simulator too much.

The simulator can be stopped at the end of the current step by selecting the "Halt" command from the menu. It can then be restarted at the same point by selecting "Run" again. To start the simulation from the beginning, the state of the system must first be initialised by choosing the "Reset" item from the menu. This sets the signal sources (described below) back to their starting states; any other initialisations required should also be done here. If the simulation has finished normally then choosing "Run" will do a reset before starting the run.

By selecting "Step" from the menu it is possible to perform a single step of the simulation. This stepping can be done after a halt, or from the start instead of running. After any step the simulation can be reset or run normally. This allows users to follow the operation of the circuit at their own pace.

4.4 Signal Sources

One of the symbols that can be placed in the circuit view is the pin, drawn as a small circle and representing the end of a wire. A pin can be either an input or an output of the logic network depending on what the other end of the wire is connected to. While output pins have no special significance, input pins act as a source of signals for the wire. The user can specify some input function for the signal source, such as a constant value, a clock with alternating high and low values, or a collection of values typed in by the user. This would be done by carrying on a dialog with the user, giving a range of options to choose from. When the simulation is run the input function is used to get the value of the input at the start of each step. The wire joined to the pin is set to the value of this function, then the algorithm described above is used to allow these signals to propagate.

4.5 Probes

To monitor the signals passing through a wire a probe is placed on that wire. The probe symbol is selected in the palette, and is placed by clicking the mouse on the wire. When the operation of the circuit is simulated the probe picks up the value of the signal passing through the wire at the end of each gate cycle. This value is then stored in the data structure used by the probe. All the signals in a run are recorded in a block of memory and the probe maintains a pointer to this. The trace view uses this pointer to get the values to plot the trace for each probe as discussed below.

4.6 The Trace View

The trace view displays the values recorded by the probes, and appears in a special kind of window. In the left hand side of this window is the name panel, which contains the names of all the probes. This can be scrolled up or down with the rest of the window if necessary, but cannot be scrolled sideways. When the contents of the window are scrolled sideways they pass under the name panel, which is not affected. The main part of the window shows the traces of the signals for each probe next to its name. An example is shown in figure 4.1. Since the trace view has not been fully implemented, this is only an impression of the way it should look.

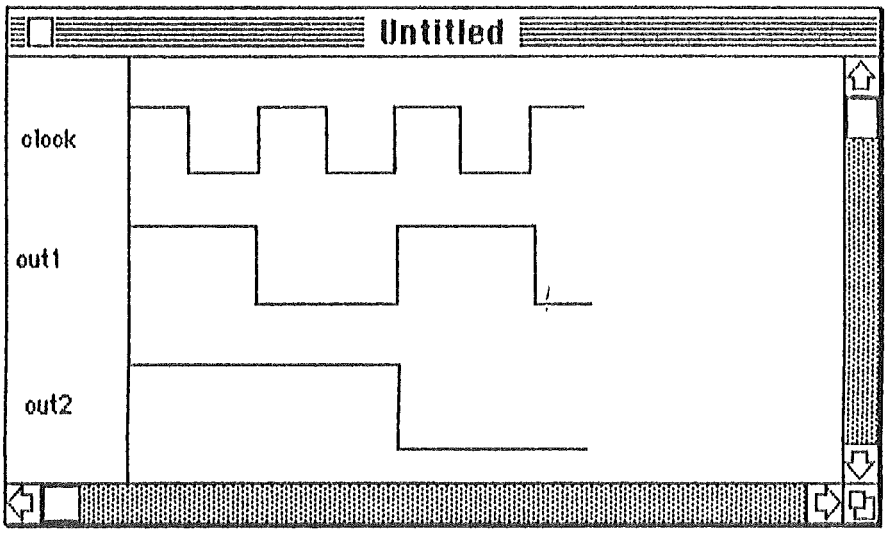

figure 4.1 Impression of trace window

Usually the trace would be displayed in a waveform format, as it is in figure 4.1. However, users may wish to see it in some other format, such as a string of binary digits. This can be done by selecting one of several possible formats from a menu. The format that the trace is displayed in will be checked in the menu.

The trace will be drawn as the simulation proceeds. The trace view maintains a list of all probes in the circuit view. After each gate cycle this list is traversed; all probes are updated and the new values are drawn into the trace window. At the end of a step the trace should be advanced, so that all steps are the same width in the trace. When the simulation has finished or has been stopped the user can scroll backwards and forwards through the trace. The name panel is used as a reference to the names of the probes that the signals are from. If a probe is unnamed then the name of the wire that it monitors is used.

5. Summary and Conclusions

In the introduction to this report it was stated that there were two parts to this project. These were the circuit view, described in section 3, and the simulator described in section 4. The code for the circuit view is almost completely written, though not completely debugged, but the simulator and the trace view exist in a detailed outline form only. Problems with a damaged disc and a corrupted source file occurred at a crucial stage, and there was not enough time to complete the project. The major sections of work remaining to be done are summarised below. As well as these, some of the existing code has to be tested and debugged. The drawing methods for some of the symbols may also have to be changed to make the symbols look right.

At the moment a beep is sounded when the user attempts to do something illegal, such as joining two output wires together. Alert boxes, windows giving a warning message, should be used to tell him or her exactly what has gone wrong.

Wire routing has yet to be implemented properly. At present the two endpoints of the wire are specified. This could be extended to a multi point approach where the user clicks at the end of each line segment until the second endpoint is reached.

The naming of components has not been implemented. The idea here is to carry on a dialog with the user, asking for the name of the component to be typed in. The name is then drawn next to the symbol that it is associated with.

The specification of signal sources should also be done by carrying on a dialog with the user. Several options could be presented, and the user would select one of these or type in the series of values to be used. The input function will have to set the state of the wire leaving the signal source according to this.

The remaining code for the simulator and trace view has to be filled in. Most of this has been written in an outline form. The simulation algorithm has not been tested, and the trace view

drawing routines will need some adjustment to make everything line up correctly. One question that needs to be answered is whether the Macintosh is powerful enough to simulate large circuits in a reasonable time. This would have to be answered by running a large number of tests, and may involve changing the simulation algorithm.

Once the basic system has been fully implemented there are a number of enhancements that could be made. At the moment symbols are always drawn facing in the same direction. By transforming the coordinates used in drawing the symbol it would be possible to draw them in any orientation. One limitation of the gate components is that they are restricted to having exactly two inputs. Since these components are all subtypes of the TGate object, most of the changes required to allow different numbers of inputs would be to the methods of this object type. The main problem involved in doing this is the location of connection points. These points would probably have to change according to how many inputs the gate had. The transform method for each component would also need to know how many inputs it had. Another enhancement that could be made is the ability to define a macrocircuit. This is a circuit element that is represented internally as a collection of components that perform some function, but appears as a single component. For example, a data register could be designed using several flip-flops, then used as a single component.

The development of the system was done on an Apple Lisa using the Pascal Workshop. This provided a compiler and linker as well as a good interactive editor. The use of MacApp saved a great deal of effort in the early stages of the implementation, but did present some obstacles. However, the MacApp system is still under development and a later version may be easier to use. The object oriented approach used was found to lend itself readily to the development of the system.

Although the project has not been completed it is hoped that the work done forms a good basis on which a fully functional system can be built. The intended system has been described in this report, and the work needed to finish it has been outlined in this section. If completed the system could be very useful as a both a logic design and an education tool, used to help students to understand digital electronics.

References

[1] Adries van Dam

"Some Implementation Issues Relating to Data Structures For Interactive Graphics"

International Journal of Computer and Information Sciences, vol. 1, no. 4. 1972

[2] Steve McCusker

"Software Tools For Teaching Computer Organisations"

The Australian Computer Journal, vol. 8, no. 1. March 1976. pp. 2-6.

[3] Robert M. McDermott

"The design of an Advanced Logic Simulator"

Byte, vol. 8, no. 4. April 1983. pp. 398-438.

[4] William M. Newman and Robert F. Sproull

Principles of Interactive Computer Graphics. Second edition. McGraw-Hill, 1981.

[5] Larry Tesler

"An Introduction to MacApp 0.1", (MacApp Documentation)

Apple Corporation, February 1985.

[6] Larry Tesler

"Object Pascal Report", (MacApp Documentation)

Apple Corporation, February 1985.

[7] Various Authors

"Macintosh Users Guide", (Macintosh Documentation)

Apple Corporation, 1983 - 1985.

Appendix A

This appendix gives a listing of the declaration part of the program. Object Pascal programs consist of an interface part and an implementation part. The implementation of the project covers nearly four thousand lines of code, and was considered too long to include as part of this report. The interface section contains declarations of all objects used. The declaration of an object includes both its fields and its methods. Comments indicate the purpose of each method.

```
UNIT ULocs;

{$M+}
{$X-}

INTERFACE

USES
  {the files containing external procedures, etc.}
    {$U-}
    {$U obj/MemTypes}       MemTypes,
    {$U obj/QuickDraw}      QuickDraw,
    {$U obj/OSIntf}         OSIntf,
    {$U obj/ToolIntf}       ToolIntf,
    {$U obj/PackIntf}       PackIntf,
    {$U UObject}            UObject,
    {$U UMacApp}            UMacApp,
    {$U UPrinting}          UPrinting,
    {$U UFiling}            UFiling;


{=========================================================}

CONST
    myFileType = 'QUUX';

    numShapes =  10;          {in the palette}


   {command numbers - used to identify commands within MacApp}
    cNewShape     =       1010;
    cMoveShape    =       1011;

    cName         =       1021;

    cSimulate     =       1030;
    cHalt         =       1031;
    cRun          =       1032;
    cStep         =       1033;
    cReset        =       1034;

    cWaveFormat   =       1041;
    cBinFormat    =       1042;
    cToggleTrace  =       1046;

    cManWires     =       1051;
    cAutoWires    =       1052;


   {shape ID numbers}
    IDAndGate     =       1;
    IDNandGate    =       2;
    IDOrGate      =       3;
    IDNorGate     =       4;
    IDNotGate     =       5;
    IDDFlipFlop   =       6;
    IDJKFlipFlop  =       7;
    IDPin         =       8;
    IDWire        =       9;
    IDProbe       =      10;
    IDLineSegment =      11;

    probeCursor   =     128;
```

```
    paletteWidth    =   32; {parameters of boxes in palette}
    paletteHeight   =   20;
    selectionGrain  =   5;  {how close have to click to an object to hit it}
    gridGrain       =   5;  {spacing of grid}
    lineOffset      =  -3;  {extent rect of line segments is offset}
    high            =  '1'; {possible logic values}
    low             =  '0';
    undefined       =  '/'; {char before 0}

{=====================================================}

TYPE


    {format of a shape in the file}
    TFiledShape =   RECORD
                        theId:      INTEGER;
                        theRect:    Rect;
                    END;

      {box within palette}
    choiceRect      =  RECORD
                         theRect : Rect;     {Bounds of the square}
                         theIcon : Handle    {Icon for the shape}
                         END;

       {pointer types used in probes for storing record of values}
    BytePtr         =  ^Byte;
    ByteHandle      =  ^BytePtr;

    drawingMode     =  (automatic, manual);  {for wires}
    traceFormats    =  (waveform, binary);   {what trace can be viewed as}
    logicValue      =  undefined..high;      {range of logic values}


{=====================================================}

        {the application object - responsible for handling top level events}

    TShapeApplication = OBJECT(TFileApplication)

      {Creation & Initialization}
        PROCEDURE TShapeApplication.IShapeApplication(itsSignature: OSType);
        FUNCTION  TShapeApplication.DoMakeDocument(itsDocKind:INTEGER):
                                                        TDocument; OVERRIDE;
        PROCEDURE TShapeApplication.PerformCommand(command: TCommand);
                                                          OVERRIDE;
                {overridden so can use a command object several times}
        END;


{=====================================================}

        {contains main data structures; responsible for file handling}

    TShapeDocument = OBJECT(TFileDocument)
        fFirstShape:     TShape;     {first element in the list of shapes}
        fLastShape:      TShape;     {last element in the list of shapes}
        fTraceView:      TTraceView; {the traceview holding the probes list}
```

```
        PROCEDURE TShapeDocument.IShapeDocument;

            {add an object to the list}
        PROCEDURE TShapeDocument.AddShape(shape: TShape);
            {set up windows and views for the document}
        FUNCTION  TShapeDocument.DoLaunchWindows: BOOLEAN; OVERRIDE;
            {set up the main view}
        FUNCTION  TShapeDocument.DoMakeView(forPrinting: BOOLEAN): TView;
                                                                OVERRIDE;
            {reading and writing procedures}
        PROCEDURE TShapeDocument.DoNeedDiskSpace(VAR dataForkBytes,
                                        rsrcForkBytes: LONGINT); OVERRIDE;
        PROCEDURE TShapeDocument.DoRead(aRefNum: INTEGER;
                            forPrinting: BOOLEAN; VAR err: OsErr); OVERRIDE;
        PROCEDURE TShapeDocument.DoWrite(aRefNum: INTEGER; VAR err: OsErr);
                                                                OVERRIDE;
            {returns a reference to the object nearest the point given}
        FUNCTION  TShapeDocument.HitDetect(aPoint: Point): TShape;
        END;


{=====================================================}

      {contains a range of symbols that can be selected and
        placed in the circuit view}

    TPalette = OBJECT(TView)
        fCurrShape:      INTEGER; {currently selected shape, 0 for the arrow}

      {Creation/Destruction}
        PROCEDURE TPalette.IPalette(itsDocument: TDocument);

           {handle mouse events}
        FUNCTION  TPalette.DoMouseCommand(VAR downLocalPoint: Point;
                                    VAR info: EventInfo;
                                    VAR hysteresis: Point):TCommand;OVERRIDE;
            {draw the view}
        PROCEDURE TPalette.Draw(area: Rect); OVERRIDE;
            {highlight the current selection}
        PROCEDURE TPalette.HighlightSelection(turnOn: BOOLEAN); OVERRIDE;
    END;


{=====================================================}

        {the circuit view object}

    TShapeView = OBJECT(TPrintableView)
        fShapeSelected:  BOOLEAN;       {are one or more shapes is selected}
        fDragging:       BOOLEAN;       {tells us if the mouse is dragging}
        fWireSketcher:   TWireSketcher; {the command object for drawing wires}
        fPalette:        TPalette;      {corresponding palette}
        fTraceView:      TTraceView;    {the trace for this circuit}
        fSimulator:      TSimulator;    {the simulation command object}
        fRunningSimulation: BOOLEAN;    {are we simulating?}

      {Creation/Destruction}
        PROCEDURE TShapeView.IShapeView(itsDocument: TShapeDocument);

            {called when there is nothing else to do}
        PROCEDURE TShapeView.DoIdle(phase: IdlePhase); OVERRIDE;
            {deselect the active selection}
```

```
        PROCEDURE TShapeView.Deselect;
            {set up a circuit view window}
        FUNCTION  TShapeView.DoMakeWindow: TWindow; OVERRIDE;

            {procedures to handle events}
        FUNCTION  TShapeView.DoKeyCommand(ch: Char; aKeyCode: Integer;
                                          VAR info: EventInfo): TCommand;
                                                                  OVERRIDE;
        FUNCTION  TShapeView.DoMenuCommand(aCmdNumber: CmdNumber): TCommand;
                                                                  OVERRIDE;
        FUNCTION  TShapeView.DoMouseCommand(VAR downLocalPoint: Point;
                                            VAR info: EventInfo;
                                            VAR hysteresis: Point): TCommand; OVERRIDE;
        PROCEDURE TShapeView.DoSetupMenus; OVERRIDE;
            {draw the view}
        PROCEDURE TShapeView.Draw(area: Rect); OVERRIDE;
            {call the procedure for all symbols in the list}
        PROCEDURE TShapeView.EachShapeDo(PROCEDURE DoThis(shape: TShape));
            {highlight the active selection}
        PROCEDURE TShapeView.HighlightSelection(turnOn: BOOLEAN); OVERRIDE;
        PROCEDURE TShapeView.PrepareToTrack;
        PROCEDURE TShapeView.SetMinViewExtent(VAR minExtent: VRect); OVERRIDE;
    END;


{=====================================================}

        {the view for drawing trace of simulation}

    TTraceView = OBJECT(TPrintableView)
        fShapeView:     TShapeView;  {the circuit that this traces}
        fProbesList:    TObjList;    {list of all probes used}
        fLastStep:      INTEGER;     {the last step of simulation so far}
        fFormat:        traceFormats;{formats for drawing of trace}
        fPanelWidth,
        fSignalSize,
        fWaveStepWidth,
        fBinStepWidth,
        fStepWidth,
        fWaveHeight:    INTEGER;      {parameters used in drawing the view}

      {Creation/Destruction}
        PROCEDURE TTraceView.ITraceView(itsDocument:   TShapeDocument;
                                        itsShapeView: TShapeView);

            {procedures to handle menus}
        FUNCTION  TTraceView.DoMenuCommand(aCmdNumber: CmdNumber): TCommand;
                                                                  OVERRIDE;
        PROCEDURE TTraceView.DoSetupMenus; OVERRIDE;
            {draw the view}
        PROCEDURE TTraceView.Draw(area: Rect); OVERRIDE;
            {draw a single step}
        PROCEDURE TTraceView.DoDrawStep(step: INTEGER);
            {do this for all probes in circuit}
        PROCEDURE TTraceView.EachProbeDo(PROCEDURE DoThis(probe: TProbe));
            {procedures to add/remove probes in list}
        PROCEDURE TTraceView.AddToProbesList(probe: TProbe);
        PROCEDURE TTraceView.RemoveFromProbesList(probe: TProbe);
            {update all probes for the step}
        PROCEDURE TTraceView.UpdateProbes(step: INTEGER);
            {advance trace, get ready for next step}
        PROCEDURE TTraceView.Advance;
    END;
```

```
{=====================================================}

        {all component objects are sub-classes of this. Defines
        the methods used in all of them}

    TShape = OBJECT(TObject)

        fExtentRect:      Rect;    {size of object}
        fOffSet:          Point;   {offset of corners of extent from centre}
        fNextShape:       TShape;  {successor in the linked list of shapes}
        fPrevShape:       TShape;  {predecessor in the linked list of shapes}
        fIsSelected:      BOOLEAN; {is this shape selected ?}
        fWasSelected:     BOOLEAN; {old selection status}
        fNextSignalState,
        fSignalState:     logicValue; {state of the component; used for wires}
        fName:            Str255;


        PROCEDURE TShape.IShape;

            {draw the symbol}
        PROCEDURE TShape.DoDrawShape;
        PROCEDURE TShape.Draw(area: Rect);

            {call the procedure for all wires connected to the component}
        PROCEDURE TShape.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
            {find the closest connection point to the point given}
        FUNCTION  TShape.FindConnectPoint(aPoint: Point): Point;
            {connect the wire to the given connection point. toOutput
            tells us if we are connecting directly to a component output}
        FUNCTION  TShape.ConnectWire(aPoint: Point; aWire: TWire;
                                     VAR toOutput: BOOLEAN): BOOLEAN;
            {disconnect the wire}
        PROCEDURE TShape.DisconnectWire(whatWire: TWire);

            {performs the logic function of the component}
        PROCEDURE TShape.Transform;
            {inverts the extent rectangle}
        PROCEDURE TShape.Highlight(turnOn: BOOLEAN);
            {returns the id number of the symbol}
        FUNCTION  TShape.ID: INTEGER;
            {write info to file}
        FUNCTION  TShape.WriteTo(aRefNum: INTEGER): OSErr;
    {$IFC qDebug}
            {gives some debugging information}
        PROCEDURE TShape.ShowDebugInfo;
    {$ENDC}
        END;


{=====================================================}

        {template for and,nand,or,nor gates}

    TGate = OBJECT(TShape)

        fInput1,
        fInput2,
        fOutput:          TWire;   {the wires connected to the component}
        fInput1Connect,
```

```
        fInput2Connect,
        fOutputConnect:   Point;  {the connection points}

        PROCEDURE TGate.IShape;  OVERRIDE;

        PROCEDURE TGate.DoDrawShape; OVERRIDE;
        PROCEDURE TGate.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
                                                                    OVERRIDE;
        FUNCTION  TGate.FindConnectPoint(aPoint: Point): Point; OVERRIDE;
        FUNCTION  TGate.ConnectWire(aPoint: Point; aWire: TWire;
                                    VAR toOutput: BOOLEAN): BOOLEAN; OVERRIDE;
        PROCEDURE TGate.DisconnectWire(whatWire: TWire); OVERRIDE;
      END;

{=====================================================}


    TAndGate = OBJECT(TGate)

        PROCEDURE TAndGate.IAndGate;

        PROCEDURE TAndGate.DoDrawShape; OVERRIDE;
        PROCEDURE TAndGate.Transform; OVERRIDE;
        FUNCTION  TAndGate.ID: INTEGER; OVERRIDE;
        END;


{=====================================================}

    TNandGate = OBJECT(TGate)

        PROCEDURE TNandGate.INandGate;

        PROCEDURE TNandGate.DoDrawShape; OVERRIDE;
        PROCEDURE TNandGate.Transform; OVERRIDE;
        FUNCTION  TNandGate.ID: INTEGER; OVERRIDE;
        END;


{=====================================================}

    TOrGate = OBJECT(TGate)

        PROCEDURE TOrGate.IOrGate;

        PROCEDURE TOrGate.DoDrawShape; OVERRIDE;
        PROCEDURE TOrGate.Transform; OVERRIDE;
        FUNCTION  TOrGate.ID: INTEGER; OVERRIDE;
        END;


{=====================================================}

    TNorGate = OBJECT(TGate)

        PROCEDURE TNorGate.INorGate;

        PROCEDURE TNorGate.DoDrawShape; OVERRIDE;
        PROCEDURE TNorGate.Transform; OVERRIDE;
```

```
        FUNCTION  TNorGate.ID: INTEGER; OVERRIDE;
        END;


{=========================================================}

    TNotGate = OBJECT(TShape)

        fInput,
        fOutput:          TWire;
        fInputConnect,
        fOutputConnect:   Point;

        PROCEDURE TNotGate.INotGate;

        PROCEDURE TNotGate.DoDrawShape; OVERRIDE;
        PROCEDURE TNotGate.Transform; OVERRIDE;
        PROCEDURE TNotGate.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
                                                              OVERRIDE;
        FUNCTION  TNotGate.FindConnectPoint(aPoint: Point): Point; OVERRIDE;
        FUNCTION  TNotGate.ConnectWire(aPoint: Point; aWire: TWire;
                                  VAR toOutput: BOOLEAN): BOOLEAN; OVERRIDE;
        PROCEDURE TNotGate.DisconnectWire(whatWire: TWire); OVERRIDE;
        FUNCTION  TNotGate.ID: INTEGER; OVERRIDE;
        END;


{=========================================================}

        {template for flip-flops}

    TFlipF = OBJECT(TShape)

        fSet,
        fReset,
        fDataJ,
        fEnableK,
        fQ,
        fQBar:          TWire;
        fSetConnect,
        fResetConnect,
        fDataJConnect,
        fEnablKConnect,
        fQConnect,
        fQBarConnect:  Point;

        PROCEDURE TFlipF.IShape; OVERRIDE;

        PROCEDURE TFlipF.DoDrawShape; OVERRIDE;
        PROCEDURE TFlipF.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
                                                              OVERRIDE;
        FUNCTION  TFlipF.FindConnectPoint(aPoint: Point): Point; OVERRIDE;
        FUNCTION  TFlipF.ConnectWire(aPoint: Point; aWire: TWire;
                                  VAR toOutput: BOOLEAN): BOOLEAN; OVERRIDE;
        PROCEDURE TFlipF.DisconnectWire(whatWire: TWire); OVERRIDE;
        END;


{=========================================================}

    TDFlipFlop = OBJECT(TFlipF)
```

```
        PROCEDURE TDFlipFlop.IDFlipFlop;

        PROCEDURE TDFlipFlop.DoDrawShape; OVERRIDE;
        PROCEDURE TDFlipFlop.Transform; OVERRIDE;
        FUNCTION  TDFlipFlop.ID: INTEGER; OVERRIDE;
        END;


{=====================================================}

    TJKFlipFlop = OBJECT(TFlipF)

        fClock:         TWire;
        fClockConnect:  Point;
        fLastClock:     logicValue;  {value of clock in last cycle;
                                        - used for detecting edges in clock}

        PROCEDURE TJKFlipFlop.IJKFlipFlop;

        PROCEDURE TJKFlipFlop.DoDrawShape; OVERRIDE;
        PROCEDURE TJKFlipFlop.Transform; OVERRIDE;
        PROCEDURE TJKFlipFlop.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
                                                                        OVERRIDE;
        FUNCTION  TJKFlipFlop.FindConnectPoint(aPoint: Point): Point; OVERRIDE;
        FUNCTION  TJKFlipFlop.ConnectWire(aPoint: Point; aWire: TWire;
                                    VAR toOutput: BOOLEAN): BOOLEAN; OVERRIDE;
        PROCEDURE TJKFlipFlop.DisconnectWire(whatWire: TWire); OVERRIDE;
        FUNCTION  TJKFlipFlop.ID: INTEGER; OVERRIDE;
        END;


{=====================================================}

    TPin = OBJECT(TShape)

        fWire:          TWire;
        fIsInputPin:    BOOLEAN;  {is this a signal source?}

        PROCEDURE TPin.IPin;
        PROCEDURE TPin.DoDrawShape; OVERRIDE;
        PROCEDURE TPin.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
                                                                    OVERRIDE;
        FUNCTION  TPin.FindConnectPoint(aPoint: Point): Point; OVERRIDE;
        FUNCTION  TPin.ConnectWire(aPoint: Point; aWire: TWire;
                                VAR toOutput: BOOLEAN): BOOLEAN; OVERRIDE;
        PROCEDURE TPin.DisconnectWire(whatWire: TWire); OVERRIDE;
            {ask user to set up input function}
        PROCEDURE TPin.SetUpInput;
            {start the input function}
        PROCEDURE TPin.StartIt;
        FUNCTION  TPin.ID: INTEGER; OVERRIDE;
        END;


{=====================================================}

    TWire = OBJECT(TShape)
```

```
        fInput,
        fOutput:          TShape;   {the components the wire joins}
        fStart,
        fFinish:          Point;    {start/finish points of wire}
        fLines:           TObjList; {the list of line segments in wire}
        fWiresOff:        TObjList; {wires joined to this one}
        fHasOutput,                 {connected directly to an output?}
        fMarked:          BOOLEAN;  {used in EachConnectionDo, to make sure
                                     only look at each connected wire once}


        PROCEDURE TWire.IWire;

        PROCEDURE TWire.Free; OVERRIDE;
        PROCEDURE TWire.DoDrawShape; OVERRIDE;
        PROCEDURE TWire.EachConnectionDo(PROCEDURE DoThis(aWire: TWire));
                                                                    OVERRIDE;
        FUNCTION  TWire.FindConnectPoint(aPoint: Point): Point; OVERRIDE;
        FUNCTION  TWire.ConnectWire(aPoint: Point; aWire: TWire;
                                    VAR toOutput: BOOLEAN): BOOLEAN; OVERRIDE;
        PROCEDURE TWire.DisconnectWire(whatWire: TWire); OVERRIDE;
            {called after disconnecting a wire in wires list}
        PROCEDURE TWire.RemoveFromWiresList(whatWire: TWire);
            {set up wire when reading from a file}
        PROCEDURE TWire.FixWire(aShapeDocument: TShapeDocument);
            {is the wire connected to an output (directy or indirectly)?}
        FUNCTION  TWire.WireHasOutput: BOOLEAN;
        PROCEDURE TWire.Highlight(turnOn: BOOLEAN); OVERRIDE;
            {set the next state of the wire}
        PROCEDURE TWire.SetSignalState(signal:logicValue);
        FUNCTION  TWire.ID: INTEGER; OVERRIDE;
        FUNCTION  TWire.WriteTo(aRefNum: INTEGER): OSErr; OVERRIDE;
    {$IFC qDebug}
        PROCEDURE TWire.ShowDebugInfo; OVERRIDE;
    {$ENDC}
        END;


{=================================================}

    TLineSegment = OBJECT(TShape)

        fStart,
        fFinish:          Point;

        FUNCTION  TLineSegment.ID: INTEGER; OVERRIDE;
        END;


{=================================================}

    TProbe = OBJECT(TShape)

        fIcon:            Handle;  {symbol drawn for this object}
        fMonitoredShape:  TShape;  {shape it is connected to}
        fhSize:           Size;    {size of memory pointed to by fhByte}
        fhByte:           ByteHandle; {pointer to the location in memory
                                       where the probes recorded values are}


        PROCEDURE TProbe.IProbe(theIcon: Handle);
```

```
        PROCEDURE TProbe.Free; OVERRIDE;
            {connect to a point on a wire. Returns false if this cannot be done}
        FUNCTION  TProbe.ConnectTo(aPoint:         Point;
                                   aShapeDocument: TShapeDocument): BOOLEAN;
        PROCEDURE TProbe.DoDrawShape; OVERRIDE;
        FUNCTION  TProbe.ID: INTEGER; OVERRIDE;
        END;


{=========================================================}

        {used in TObjList, below}

    TListObject = OBJECT(TObject)
        fShape:  TShape;       {Shape associated with the object}
        fNext:   TListObject;  {next in list}

    END;



{=========================================================}

        {used in several places to keep a list of shape objects}
    TObjList = OBJECT(TObject)

        fFirst,
        fLast:    TListObject; {Start and finish of list}

      {Creation/Destruction}
        PROCEDURE TObjList.IObjList;
        PROCEDURE TObjList.Free;  OVERRIDE;
        PROCEDURE TObjList.ZeroObjList;  {reset to empty list}

            {add,remove, manipulate objects in list}
        PROCEDURE TObjList.AddObj(shape: TShape);
        PROCEDURE TObjList.DeleteObj(obj: TListObject);
        PROCEDURE TObjList.EachListObjectDo(PROCEDURE DoThis(obj: TListObject));

    END;



{=========================================================}

        {Command object used when selecting symbols within an area}

    TShapeSelector = OBJECT(TCommand)
        fShapeView:  TShapeView;  {the view we are in}

        PROCEDURE TShapeSelector.IShapeSelector(aShapeView: TShapeView);

            {follow the mouse, give feedback to user}
        PROCEDURE TShapeSelector.TrackFeedback(anchorPoint, nextPoint: Point;
                                               turnItOn, mouseDidMove: BOOLEAN);
                                                                    OVERRIDE;
        FUNCTION  TShapeSelector.TrackMouse(aTrackPhase: TrackPhase;
                        VAR anchorPoint, previousPoint, nextPoint: Point):
                                                          TCommand; OVERRIDE;
        END;
```

```
{================================================}

        {Command object used when placing a new symbol}

    TShapeSketcher = OBJECT(TCommand)
        fShape:     TShape;     {the shape that will be drawn}
        fShapeView: TShapeView;

        PROCEDURE TShapeSketcher.IShapeSketcher(aShapeView: TShapeView;
                                                protoShape: TShape);
        PROCEDURE TShapeSketcher.Free; OVERRIDE;

            {add/remove from data structures}
        PROCEDURE TShapeSketcher.AddShape;
        PROCEDURE TShapeSketcher.DeleteShape;
        PROCEDURE TShapeSketcher.DoIt; OVERRIDE;
        PROCEDURE TShapeSketcher.RedoIt; OVERRIDE;
        PROCEDURE TShapeSketcher.TrackFeedback(anchorPoint, nextPoint: Point;
                                  turnItOn, mouseDidMove: BOOLEAN); OVERRIDE;
        FUNCTION  TShapeSketcher.TrackMouse(aTrackPhase: TrackPhase;
                        VAR anchorPoint, previousPoint, nextPoint: Point):
                                                          TCommand; OVERRIDE;
        PROCEDURE TShapeSketcher.UndoIt; OVERRIDE;

        END;


{================================================}

        {Command object used when drawing a new wire
        This object is intended to be reused, getting the points that
        it needs in building up the wire}

    TWireSketcher = OBJECT(TCommand)

        fShapeView:         TShapeView;
        fReleasePoint,                      {point where mouse was released}
        fStartPoint,
        fFinishPoint,                       {start/finish of wire}
        fOldStartPoint,
        fOldFinishPoint:    Point;          {Oldxxxx used for undo/redo command}
        fFirstShape,
        fLastShape,
        fOldFirstShape,
        fOldLastShape:      TShape;         {components connected to}
        fWire:              TWire;          {the new wire}
        fHaveOutput:        BOOLEAN;        {start of wire was an output}

        PROCEDURE TWireSketcher.IWireSketcher(aShapeView: TShapeView);
        PROCEDURE TWireSketcher.Free; OVERRIDE;

        PROCEDURE TWireSketcher.TrackFeedback(anchorPoint, nextPoint: Point;
                                  turnItOn, mouseDidMove: BOOLEAN); OVERRIDE;
        FUNCTION  TWireSketcher.TrackMouse(aTrackPhase: TrackPhase;
                        VAR anchorPoint, previousPoint, nextPoint: Point):
                                                          TCommand; OVERRIDE;

            {add/remove from data structures}
        PROCEDURE TWireSketcher.AddWire;
```

```
        PROCEDURE TWireSketcher.DeleteWire;

            {used for routing the wire in automatic mode}
        PROCEDURE TWireSketcher.AutoRouter;
            {set up the fields etc. in the wire when it has been placed}
        PROCEDURE TWireSketcher.SetUpWire;

        PROCEDURE TWireSketcher.Doit; OVERRIDE;
        PROCEDURE TWireSketcher.Redoit; OVERRIDE;
        PROCEDURE TWireSketcher.Undoit; OVERRIDE;

        END;


{=====================================================}

         {command object created when dragging a symbol}

    TShapeDragger = OBJECT(TCommand)
        fShapeView:  TShapeView;
        fBounds:     Rect;          {Union of fExtentRect of all shapes being
                                     dragged, before the move}
        fDeltaH:     INTEGER;       {distances moved, horizontal/vertical}
        fDeltaV:     INTEGER;
        fCutLeavers,                {should wires out of selection be deleted?}
        fWiresLeaving: BOOLEAN;     {are there any wires leaving selection?}


        PROCEDURE TShapeDragger.IShapeDragger(aShapeView: TShapeView);

        PROCEDURE TShapeDragger.Doit; OVERRIDE;
        PROCEDURE TShapeDragger.FixSelection;
            {do the moving}
        PROCEDURE TShapeDragger.MoveBy(deltaH, deltaV: INTEGER);
        PROCEDURE TShapeDragger.Redoit; OVERRIDE;
        PROCEDURE TShapeDragger.TrackFeedback(anchorPoint, nextPoint: Point;
                                              turnItOn, mouseDidMove: BOOLEAN); OVERRIDE;
        FUNCTION  TShapeDragger.TrackMouse(aTrackPhase: TrackPhase;
                    VAR anchorPoint, previousPoint, nextPoint: Point): TCommand; OVERRIDE;
        PROCEDURE TShapeDragger.Undoit; OVERRIDE;

        END;


{=====================================================}

         {command object created when deleting symbols}

    TCutCmd = OBJECT(TCommand)
        fShapeView:       TShapeView;
        fShapeDocument:   TShapeDocument; {document we are working in}
        fObjList:         TObjList;       {list of symbols to be cut}

        PROCEDURE TCutCmd.ICutCmd(itsCmdNumber: INTEGER;
                                  aShapeView:   TShapeView);
        PROCEDURE TCutCmd.Free; OVERRIDE;

        PROCEDURE TCutCmd.Commit; OVERRIDE;
        PROCEDURE TCutCmd.Doit; OVERRIDE;
        PROCEDURE TCutCmd.Redoit; OVERRIDE;
        PROCEDURE TCutCmd.Undoit; OVERRIDE;
```

```
        END;


{=========================================================}

         {command object created when naming a symbol}

    TNameCmd = OBJECT(TCommand)
        fShapeView:       TShapeView;
        fShape:           TShape;     {the symbol being named}
        fNewName,
        fOldName:         Str255;    {names saved for undo/redo commands}

        PROCEDURE TNameCmd.INameCmd(itsCmdNumber: CmdNumber;
                                    aShapeView:   TShapeView);

        PROCEDURE TNameCmd.DoIt; OVERRIDE;
        PROCEDURE TNameCmd.RedoIt; OVERRIDE;
        PROCEDURE TNameCmd.UndoIt; OVERRIDE;

        END;


{=========================================================}

         {command object created for simulation.
          Intended to be reused from within a circuit view}

    TSimulator = OBJECT(TCommand)
        fShapeView:       TShapeView;
        fTraceView:       TTraceView;{the view associated with the simulator}
        fStep:            INTEGER;    {what step we are up to}
        fNoInputs:        BOOLEAN;    {no input functions return a value}
        fChangeCount:     INTEGER;    {used to see if circuit has been changed
                        since the last step - if so, reset before simulating}

        PROCEDURE TSimulator.ISimulator(aShapeView:   TShapeView);
        PROCEDURE TSimulator.Free; OVERRIDE;

        PROCEDURE TSimulator.ResetSimulator;
            {do the command passed - run/step/halt/reset}
        PROCEDURE TSimulator.PerformCommand(itsCmdNumber: CmdNumber);
            {get the next set inputs and stimulate the input wires}
        FUNCTION  TSimulator.GetNextInput: BOOLEAN;
            {do a single step of the simulation}
        PROCEDURE TSimulator.PerformStep;

        END;


VAR
    shapesArray:      ARRAY [1..numShapes] OF TShape;     {prototype shapes from the palette}
    choiceArray:      ARRAY [0..numShapes] OF choiceRect; {possible shapes in the palette}
    paletteExtent:    Rect;                               {view extent for the palette (same f
    crossHdl:         CursHandle;                         {standard cross cursor}
    watchHdl:         CursHandle;
    probeHdl:         CursHandle;                         {cursor shown when using a probe}
    gWireDrawingMode: drawingMode;
```

```
IMPLEMENTATION

{implementation section is in another file.
   The files shapes.text and UTraceview.text are also used; these
   are included from within ULocs2.}

   {the program can be compiled and loaded by running the Lisa exec file
      "fast(ulocs)". This also copies the application onto Macintosh disc}

{$I ULocs2.TEXT}

END.
```