TOKEN BUS LOCAL AREA NETWORK SIMULATOR

by

Gregg Guarnera

Project submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

APPROVED:

Emile Haddad, Chairman

Thomas Reid

William May

December, 1991

Blacksburg, Virginia
TOKEN BUS LOCAL AREA NETWORK SIMULATOR

by

Gregg Guarnera

Committee Chairman: Emile Haddad
Computer Science

(ABSTRACT)

This project is a token bus local area network simulator written in Pascal on an IBM PC compatible. The simulator is written for the Microsoft Windows operating environment and makes use of a graphical user interface for controlling the simulation. The program is object-oriented to make use of the Borland ObjectWindows Windows interface and because of the suitability of object-oriented programming to graphics and simulation applications. All basic token bus network functionalities specified by the Institute Of Electrical And Electronics Engineers (IEEE) 802.4 token bus standard are implemented in the simulation plus an added function to resolve duplicate node addresses.

The network nodes and bus are drawn using Windows' color graphics. The state of each node is represented by text as well as the color and style each node is drawn in. The frame being transmitted is shown as large text within the bus object on the screen. The direction of data transfer on the bus is shown graphically as is the current location of the token among the nodes.

The user of the simulation has the ability make any node active, inactive, or passive, or to make any node fail. The user may make a
node send data to one other node or many other random nodes. The addresses of the nodes may also be changed. The user may pause, step through, or continue the simulation, control the simulation speed, control the error rate of data on the bus, and produce a lost token scenario.
ACKNOWLEDGEMENTS

I would like to acknowledge the support and encouragement that my father and my brothers have given me for so long. My father stressed the importance of education since I was in diapers and continued to from then on. I don't know if I would have gotten this far if he hadn't been there to help financially and morally through the chaotic experience that was my college career. I know that 'thanks' doesn't quite say it all, but thanks Dad.

As for my brothers, I would like to acknowledge their support for just helping me get through life with my sanity still in check and a smile on my face. Any of us doing anything major (a master's degree for example) without the others being 100% involved would completely go against everything we have ever done together or been to each other. Thanks guys.
# TABLE OF CONTENTS

Acknowledgements ................................................................. iv
Table Of Contents ........................................................................ v
Introduction .................................................................................. 1
Materials And Methods .................................................................. 4
  Token Bus Local Area Networks .................................................. 4
  Addition To The Ring ............................................................... 6
  Deletion From The Ring ............................................................ 7
  Initialization Of The Token Ring ............................................... 8
  Fault Management ...................................................................... 8
  Duplicate Addresses/Tokens ...................................................... 9
  Loss of Token ............................................................................ 9
  Token Passing ........................................................................... 10
The Language and The Windows Environment ............................. 10
Object-Oriented Programming ..................................................... 12
The Simulation Display And User Interface ................................. 13
The Structure of The Program ..................................................... 24
The Objects .................................................................................. 27
  The Application ......................................................................... 27
  The Main Window ..................................................................... 28
    The Menu System .................................................................... 28
    The Control Bar ...................................................................... 30
  The TSimObject Object ............................................................ 31
  The Bus Object ......................................................................... 31
    The Bus Display ..................................................................... 32
    The Bus Operational Routines ................................................ 33
  The Link Object ...................................................................... 34
  The Node Object ...................................................................... 35
    The Node Display ................................................................... 36
    The Status Update Routines ................................................... 37
    The Input Buffer Routine ...................................................... 38
    Contention Resolution .......................................................... 39
    Logical Ring Initialization ..................................................... 40
    Node Addition To The Ring ................................................... 41
    Node Deletion From the Ring .................................................. 42
    The Solicit Successor Routine ................................................ 43
    The Data Sending Routine ..................................................... 45
    The Token Passing Routine .................................................... 46
    Duplicate Node Address Resolution ....................................... 48
Results ....................................................................................... 50
  Scenario 1: Logical Ring Initialization With Contention ............ 50
  Scenario 2: Token Passing ....................................................... 58
  Scenario 3: Token Passing To A Crashed Node ......................... 63
  Scenario 4: Data Sending ....................................................... 68
  Scenario 5: Node Deletion From The Logical Ring .................... 72
  Scenario 6: Duplicate Address Resolution ................................ 76
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discussion</td>
<td>31</td>
</tr>
<tr>
<td>Programming Approach</td>
<td>81</td>
</tr>
<tr>
<td>Simulator Uses</td>
<td>82</td>
</tr>
<tr>
<td>Program Expansion: Performance Monitoring</td>
<td>83</td>
</tr>
<tr>
<td>Program Extension: CSMA/CD Simulator</td>
<td>84</td>
</tr>
<tr>
<td>Conclusions</td>
<td>86</td>
</tr>
<tr>
<td>Summary</td>
<td>87</td>
</tr>
<tr>
<td>Literature Cited</td>
<td>88</td>
</tr>
<tr>
<td>Appendix: Program Code</td>
<td>89</td>
</tr>
</tbody>
</table>
INTRODUCTION

There are several technologies involved in this project. These technologies include object-oriented programming, Microsoft Windows programming, token bus local area networks, and simulation design and construction.

The simulator program is written in Borland's Turbo Pascal for Windows. This language is designed to operate in the Microsoft Windows operating environment, a graphical user interface environment for IBM PC compatible computers. The language is object-oriented, a programming style that groups data and code into objects. Objects are created in such a language to represent real-world objects that have properties and actions associated with them. Borland includes a collection of Microsoft Windows interface objects, called ObjectWindows, with its Pascal compiler. ObjectWindows makes it much easier to write Windows applications compared to conventional Windows programming.

The specification for the behavior of the token bus network is taken from the Institute Of Electrical And Electronics Engineers (IEEE) 802.4 token bus network standard. This behavior is modeled in the node object definition in the program. This behavior determines the current status and action being performed by each node. Each node's status and current action is displayed graphically as well as textually to give the user of the simulation a clear picture of what each node is doing and why.

The user of the simulation has control over each node's status, the data it sends, and its address on the network by choosing from the
header bar menu. In addition, the user controls the speed of the simulation and the error rate of network transmissions by manipulating scroll bars. The user can choose to pause, step through, or continue the simulation at the defined speed by clicking the mouse on push buttons on the main window's control bar. The user can also simulate an lost token scenario by choosing the appropriate menu selection.

Each of the major functional areas of the program is coded as a separate object in Pascal. These major functional areas include the user interface, the network bus, and the network nodes. There are three additional object definitions: the application object, the node-bus link object, and the SimObject object which is used to derive the bus, link, and node objects.

Most of the programming work is put into the node object because most of the network functionality resides with the network nodes in a decentralized network access scheme such as token bus. Included in the node object description are routines to perform data transmission, token passing, network initialization, network maintenance, contention resolution, addition and deletion from the logical ring, and user interface support. The user interface support consists of responding to requests by the user, such as changing a node's address, and drawing the node on the screen to accurately reflect the node's status and current activity being performed.

The user interface object is the second most complicated of the objects in the simulation. This object is responsible for processing all of the menu commands from the menu bar and the simulation controls
from the control bar. It is also responsible for drawing the window and responding to the timer ticks from the window's system timer. The bus and link objects are much simpler than the node and user interface objects. The routines for these objects are concerned mostly with user interface support. The bus object does, however, contain several routines that are responsible for simulating the actions of a network bus.
MATERIALS AND METHODS

Token Bus Local Area Networks

The IEEE 802.4 token bus network standard is used to define the actions performed by the nodes in the simulation. The token bus algorithm is relatively new and is not as commonly used as other protocols. It is more complicated, and therefore more expensive, than other Medium Access Control (MAC) methods for bus architectures, most notably Carrier Sense Multiple Access with Collision Detection (CSMA/CD), but offers high throughput when there is a high offered load by the network nodes. The reason for the high throughput is that the token bus MAC method does not suffer from the contention problems experienced by CSMA/CD and other contention based protocols. Therefore, more time is spent sending data and less time is spent resolving contention.

With the token bus protocol, nodes gain access to transmit on the network bus by receiving the token. After receiving the token, the node is free to send any data it has to send. Since there is only one token, only one node transmits on the bus at a time so there is no contention with other nodes. Each active node on the bus has a predecessor node and a successor node so that a logical ring is formed as shown in Figure 1.
Figure 1. Token Bus Logical Ring
After sending any data, the node passes the token to its successor in the logical ring. The token is passed from node to node in the logical ring giving each node a fair chance to send data without contending with other nodes.

Nodes on the network can be one of three status types: active, passive, or inactive. Active nodes are nodes that are actively part of the logical ring and are allowed to receive the token and send data. Passive nodes are not part of the logical ring so they do not receive the token but they are allowed to receive and acknowledge transmissions on the bus. Inactive nodes are nodes that are physically connected to the bus but are not involved in any way with transmissions on the bus. The simulation also recognized a status class called crashed nodes. These are nodes that experienced a system failure and became inactive as a result.

There are several functions that must be performed by each node of a token bus network. These functions are addition to the logical ring, deletion from the ring, ring initialization, and fault management.

**Addition To The Ring**

Every so often members of the logical ring are required to allow other nodes the opportunity to add into the ring. A token owning node does this by issuing a *solicit-successor* frame that allows nodes with an address between itself and its successor to add to the ring. When a node receives a *solicit-successor* frame and wants to add to the ring it responds by setting its successor to the address of the token holding node's successor and
returning a *set-successor* frame to the token holding node. When
the token holding node receives the *set-successor* frame it makes
the new node its successor and passes the token to the new node.
Whenever a node receives a token it sets its predecessor address
to the address of the token sender. This rule ensures that the
new node and its successor have the correct predecessor addresses.

If more than one node responds to a *solicit-successor* frame,
the token holding node issues a *resolve-contention* frame to let
the sending nodes know that there is a contention situation that
must be resolved. Each of the contending nodes waits either 0, 2,
4, or 6 clock counts to resend their *set-successor* frame depending
on the first two bits of their address. If a node hears another
transmission on the bus before it is supposed to transmit, it
backs off and enters a listener state. If there is contention
again then the contending nodes repeat the process using the next
pair of address bits. This process continues until there is no
more contention.

**Deletion From The Ring**

Deletion from the ring is much simpler than addition. When a
node wants to remove itself from the logical ring, it sends a *set-
successor* frame to its predecessor containing the address of its
successor. This has the effect of making the predecessor's
successor equal to the current node's successor. The current node
then sends the token to its successor as usual. When the current
node's predecessor gets the token on the next round, it sends it
to the current node's old successor and the successor updates the pointer to its predecessor as normal when receiving the token. At this point, the current node is completely removed from the ring.

**Initialization Of The Token Ring**

Each node is responsible for initializing the ring after a certain amount of time passes with no activity on the bus. This situation can occur when a token holding node fails or during network start-up. A node performs initialization by issuing a *claim-token* frame on the bus. If there is no contention with other nodes, the claiming node considers itself the token holder and issues a *solicit-successor* frame inviting the full range of addresses to enter the logical ring. A new node is added using the process described previously and a new ring is created. If there are no responses to the *claim-token* frame then the node quits and enters a listener state. If there is contention when the node issues the *claim-token* frame, the node detects it by listening to the bus and resolves the contention using the scheme used for contention during addition to the ring.

**Fault Management**

There are problems that can arise in a token bus network that must be accounted for. These problems are duplicate node addresses, the loss of the token, and token passing faults.
Duplicate Addresses/Tokens

The problem of multiple tokens in the ring is usually caused by two nodes having the same address. The previous node in the ring sends the token to its next node but if there are two or more nodes with that next address then all of those nodes become token holders. The multiple token state is resolved by a token holding node. When a token holding node hears another station issuing a token, sending data, or soliciting a successor, it drops the token and enters an listener state as a normal, active node. If there are two token holders then there is either one or none left when one hears the other issue one of the above frame types.

Loss of Token

The lost token scenario occurs when none of the active nodes considers itself the token owner. This scenario can occur when a token holding node crashes before it gets the chance to pass the token. When this scenario does occur there is a long period of inactivity on the bus due to the lack of token. After a certain amount of time goes by without any activity, each node is responsible for attempting to re-initialize the logical ring using the normal ring initialization strategy.
Token Passing

When a node receives a token, it is expected to immediately release a token, data, or control frame to let the token sender know that the token transmission is successful. The token sender waits two time slots for this frame to ensure that its successor is active and that it receives the token. If the sender hears nothing then it resends the token. If it again hears nothing after waiting two more time slots, the sender assumes that its successor is no longer active and issues a who-follows frame prompting the remaining nodes to find the successor of the receiving node. The receiver’s successor sends a set-successor frame to the sender and the sender sets its successor address to match. If there is no response then the sender re-issues the who-follows frame. If there is still no response after the second who-follows frame, the sender issues a solicit-successor frame inviting the full range of addresses to add to the ring. If this fails, the sender assumes that there has been a catastrophic failure of the network and reverts to a listener mode.

The Language and The Windows Environment

Borland’s Turbo Pascal for Windows is the language chosen for this project. Turbo Pascal for Windows is an easy to read and write language that is also very powerful. Turbo Pascal differs from standard Pascal
in that is allows for separate compilation by breaking the program into units. Units allow the programmer to work on and compile one small section of the program without having to constantly save and recompile the whole program. The two biggest advantages of Turbo Pascal for Windows, however, are that it allows object-oriented programming and allows Microsoft Windows programming. In fact, this version of Pascal comes packaged with a collection of objects, called ObjectWindows, that simplifies the programming interface with Microsoft Windows.

Windows is an operating system shell written by Microsoft corporation for the IBM PC and compatibles. Windows is an operating environment that provides the user with a graphical user interface similar to that of the Apple Macintosh's. Version 3.0 of Windows is used for this project. Besides providing the user a complete shell for the operating system, Windows also provides the programmer with a large variety of graphical user interface services and resources. The framework provided by Windows allows the programmer to create windows, menus, scroll bars, buttons, dialogue boxes, color graphics and much more using a programming language like Turbo Pascal for Windows. It provides a simple means for dealing with asynchronous events like mouse clicks, button selections, and scroll bar movements by providing system messages in response to these events and allowing subroutines to act on them. Windows also allows graphics and text to be mixed easily and provides a system timer that can be controlled through the program.
Object-Oriented Programming

Object-oriented programming is a programming style that groups data and the actions that are performed on that data together as an object. The actions are called methods and are described as subroutines. It is assumed that methods have access to any data included in their object but cannot directly access data from another object. When objects are declared, they may inherit the data and methods from a previously described object and then add any additional data and methods desired. For example, an object named shape may have a size and a method called draw that draws the shape on the screen. Another object named circle can inherit all of the attributes of the shape object and add data for color and update the draw method to correctly draw the object as a circle. Using this methodology, real life objects are described in code by their attributes (data) and the actions that they perform (methods).

An added benefit of object-oriented programming is that it allows programs to be extended easily through inheritance. New programs can simply inherit objects from the current one and add any needed methods or data without having to rewrite any of the code or worry about any of the details associated with the original objects.

Object-oriented programming is used in this project for two reasons. The first is to take advantage of the ObjectWindows object library included with Turbo Pascal for Windows. By making the user interface portion of the program object-oriented, the ObjectWindows user interface objects can be used to simplify the task of creating an application in the Windows environment. ObjectWindows includes objects
for creating windows, buttons, scroll bars, static text, dialogue boxes, Windows display contexts, pens, brushes, and the Windows applications themselves. The object-oriented programming style also allows an object's methods to be invoked directly in response to a Windows system message such as the system timer or an asynchronous event.

The second reason that object-oriented programming is used is to represent the different parts of the token bus network so that they can all work independently. The bus, the nodes, and the links connecting the nodes to the bus are all represented as objects. The bus methods do not perform any of the node functions and vice versa. Furthermore, each of the eight nodes in the simulation is created as a separate but identical object so each node is able to see and change only the data associated with itself. It can not see any of the data associated with any other node or the bus. Besides providing a certain sense of reality to the mapping between the real world and computer program representing it, object-oriented programming also provides a safeguard to ensure that the wrong code does not accidently alter the wrong data.

The Simulation Display And User Interface

The simulation display consists of a main window with a mouse selectable menu bar across the top. On the menu bar, there is a selection for each of the eight nodes in the simulation plus an additional selection to invoke a no token scenario. Each of the node menu selections invokes a pull-down menu that contains selections for changing node status, data transmission options, and node address. Below the menu bar is a control bar that contains the simulation
controls. These controls include mouse activated push buttons to pause, step through, and continue the simulation and scroll bars to control the simulation speed and data transmission error rate.

Below the control bar is the graphical simulation display that shows the network nodes and bus. The screen display of a network node is a rectangle with a solid border. Each rectangle has four sections that help to show its status. The top section is the header and it shows the name of the node. The color of the header changes with the node's status. The header is white inside when the node is inactive or crashed, gray filled when the node is a passive listener, green when the node is active, and orange when the node holds the token. The section below the header is the address section and it displays the node's current address as well as the address of the predecessor and successor nodes. Below the address section is the node status section. This section shows the current status of the node, the current action being performed, and the current step of the action being performed.

The bottom section of the node display is the counter section and it contains three counters. The first counter, Cs, is the current contention step that the node is on. This corresponds to the pair of bits the node is using to determine its retransmission wait time to resolve the contention. The second counter, Cw, is the retransmission wait time that the node is using to resolve the contention. The third counter, Cnt, is a general counter that shows the value of the counter that corresponds to the current activity of the node. If the current
activity is contention resolution then the counter shows the number of
time slots the node has waited since the last transmission.

The bus is drawn as a long open rectangle that connects each of the
network nodes. Within the bus rectangle is the current frame type being
sent on the bus by the nodes. Connecting each node to the bus is a link
that is shown as a small rectangle with an arrow inside to show the
direction of data transmission between the node and the bus. A sample
of the actual simulator display is shown in Figure 2 followed by a
description of the simulator controls in Table 1.
### Figure 2. Token Bus Simulator Display

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 1</td>
<td>Address: 2</td>
<td>Address: 3</td>
<td>Address: 4</td>
</tr>
<tr>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
</tr>
<tr>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
</tr>
<tr>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
</tr>
<tr>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
</tr>
<tr>
<td>Cw: 0</td>
<td>Cw: 0</td>
<td>Cw: 0</td>
<td>Cw: 0</td>
</tr>
<tr>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
</tr>
</tbody>
</table>

---

**Idle**

<table>
<thead>
<tr>
<th>Node 5</th>
<th>Node 6</th>
<th>Node 7</th>
<th>Node 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 5</td>
<td>Address: 6</td>
<td>Address: 7</td>
<td>Address: 8</td>
</tr>
<tr>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
</tr>
<tr>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
</tr>
<tr>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
</tr>
<tr>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
</tr>
<tr>
<td>Cw: 0</td>
<td>Cw: 0</td>
<td>Cw: 0</td>
<td>Cw: 0</td>
</tr>
<tr>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
</tr>
</tbody>
</table>
**Table 1. Token Bus Simulator Controls**

<table>
<thead>
<tr>
<th>Screen Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node Menu Items</td>
<td>Pops up a menu for node status changing, data sending, and address changing.</td>
</tr>
<tr>
<td>No Token Menu Item</td>
<td>Creates a lost token scenario on the network.</td>
</tr>
<tr>
<td>Pause Button</td>
<td>Freezes the current state of the simulation.</td>
</tr>
<tr>
<td>Step Button</td>
<td>Steps the simulation through one clock count at a time.</td>
</tr>
<tr>
<td>Go Button</td>
<td>Resumes normal operation of the simulation after the Pause or Step button is pressed.</td>
</tr>
<tr>
<td>Speed Scroll Bar</td>
<td>Controls the speed at which the simulation runs.</td>
</tr>
<tr>
<td>Bus Error Scroll Bar</td>
<td>Controls the error rate at which data is transmitted on the bus.</td>
</tr>
</tbody>
</table>
Each of the node menus on the menu bar are identical. Clicking on one of the node selections causes a pull-down menu to appear. Each pull-down menu contains three sections: the status section, the data sending section, and the address change section.

The status section of the node menu contains four options of which only one may be selected at a time. These options allow the user to change the status of the corresponding node to either active, inactive, passive, or crashed. The currently chosen option is highlighted with a check mark next to it. The active option makes the node an active member of the ring. The inactive option makes the node inactive and causes a graceful removal from the ring if the node is active. The passive option does the same thing but leaves the node in the passive listener stated so it may receive data from other nodes. The crash option causes the node to crash or fail. If the node is currently active when this happens, the node crashes without the opportunity to remove itself from the ring gracefully. A sample of a node pull-down menu is shown in Figure 3.
Figure 3. Node Menu Bar Selections
The data sending section of the menu has two options. The AutoSend option causes the current node to send data to a random node on the network each time it has the token. A check mark is placed next to the AutoSend option when it is active. The other option in the data sending section is the Send To option. The Send To option results in a pop-up dialogue box being displayed as shown in Figure 4. The user specifies the node that the current node is to send data to by typing the address of the receiver node in the dialogue box. The current node responds by attempting to send data to the given receiver the next time it has the token.
Figure 4. The Send To Function Dialogue Box
The last section of the node pull-down menu is the address change option. This option allows the user of the simulator to change the address of any node on the network regardless of the node's status. Choosing the address change option causes a dialogue box to be displayed as shown in Figure 5. The user enters the node's new address in the dialogue box and the node's address is changed instantly, regardless of whether or not another node has the same address.
Figure 5. The Address Change Function Dialogue Box
The Structure of The Program

The program is written as a group of objects and a group of global variables written in Pascal. The code is stored in six separate files, one for the main program and the other five coded as Pascal units that are separately compiled. The program is run either from within the Turbo Pascal For Windows programming environment or directly from the Windows operating environment by double clicking the mouse over the token bus icon.

There are six objects that make up simulator. The main program application object, TMyApplication, is the object that opens the application in the Windows environment, invokes the main window object, and provides the interface with Windows. This object is stored in the Token1 file which is the file that is compiled and linked as the main program. The main window object, TMyWindow, is stored as the GGUI unit file. This object contains all of the graphical user interface definitions and performs many small tasks required of a main window by the Windows environment. There is an object, TSimObject, that is used as a parent object that is inherited by all other objects that are displayed as part of the simulation. This object is stored in unit file GSimObj and is inherited by the bus, link, and node objects. The object that represents the bus, TBus, is stored in the GBusObj unit file with the object that connects the bus to the nodes, TLink. The last object is the node object, TNode, and is stored in unit file GNodeObj. This object contains all of the code that dictates the behavior of the
network nodes. In addition to the objects' files there is an additional file that contains global data definitions, GGlobal.

There are eight node object instances and eight node to bus link instances, one of each for each node in the network. These object instances are grouped in what is called a collection. A collection is a data structure that allows methods to act on many instances at a time, similar to an array or linked list. The collection of nodes is a global variable called TNodeCollection. This collection is global to allow the main window object definition access to the node methods while allowing the bus object access to the bus buffers within each node object in the collection. The link object collection is defined within the bus object since the bus controls their behavior. In fact, the bus is the only object that needs to worry about the links. The rest of the objects view the links as part of the bus and are not concerned with how they are manipulated.

The simulation is controlled by a clock pulse provided by Windows. Each clock pulse is actually a Windows message sent to the application for the application to process any way it wants. The rate of the clock pulse is initially set by the TMyWindow object initialization method and is controlled during the simulation by TMyWindow methods that respond to messages from the speed control scroll bar. The TMyWindow object has a method named TimerTick that processes each clock pulse. This routine calls the TimerTick methods of each of the nodes and of the bus upon receipt of each clock pulse message from Windows. The node and bus
TimerTick methods in turn call the remaining methods in the object to perform the processing needed for one time slot on the network.

The global unit contains a variety of values including all of the Windows control and message identifiers. These identifiers are used to internally represent scroll bars, buttons, static display elements, and Windows messages. The Windows messages are sent from Windows itself to communicate the happening of events such as mouse clicks. The messages also are sent from buttons, scroll bars and menu selections to signify that they have been activated. The global data unit also contains screen data and positions as well as algorithm controlling constants and global variables and data types.
The Objects

The six objects that make up the simulator are the application, main window, TSimObject, bus, link, and node objects. Each object performs specific tasks that are independent of the tasks performed by the other nodes. Two types of methods are common among all of the objects, however, the constructor and the destructor. The constructor method, named Init in each object, performs all needed object initialization functions. The destructor method, named Done in each object, performs all clean up duties needed when the object is discarded. The application and main window objects perform all Windows and user interface functions. The TSimObject, bus, link, and node objects perform all of the simulation functions. Most of the code for the Windows and user interfaces lies in the main window object while most of the simulation code is contained in the node object.

The Application

The application object, TMyApplication, contains the definition for an application in the Windows environment and handles the application interface with Windows. A part of this definition is the name of the main window that represents the application to the user. The main window name TMyWindow is passed as a parameter into the initialization method for the application object. When the application is invoked by the user in Windows, the application object starts up and creates a main window described by the TMyWindow object definition. When the
user decides to end the application, the application object removes itself from the Windows environment through its Done method.

The Main Window

The main window object supplies a window for the application to run in and performs all of the user interface functions. The object's Init method provides the window name, creates the node object collection, the buttons, the scroll bars, and initializes variables. The SetupWindow method is the method that initializes the checkmarks in the node menus, sets up the Windows system timer, and initializes the nodes in the node collection. This method is also responsible for creating and initializing the bus object.

Windows holds the application's main window responsible for being able to recreate its contents on the screen. To uphold this responsibility, the Paint method is described to redraw the screen. This function is performed by redrawing the interface objects and then calling the bus and node Draw methods to make them draw themselves. Most of the main window object's other functionality deals with supporting the user interface menu and control bar.

The Menu System

The main menu for the simulator is a Windows resource that is created using the Whitewater Resource Tool-kit and
is precompiled and automatically linked into the main window object. The menu resource is created in the tool-kit by describing each level of the menu in order and associating a message identifier with each selection. The identifier for the menu is supplied within the TMyWindow Init method to link the precompiled menu resource into the main window.

When a menu selection is chosen by the user, Windows returns the message identifier associated with the menu selection along with a generic command message to the application. The WMCommand method is defined in TMyWindow to respond to the generic command message and process the message identifier returned. Another option for this part of the code is to create a separate method to respond to each of the 56 node menu selection message identifiers. Rather than have such a large number of simpler methods, a single more complicated method handles menu selections.

The WMCommand method begins by decoding the message identifier to determine which node and menu selection is selected. A case statement ensures that the correct procedure for the menu selection is performed. The procedures for the menu selections update the checkmarks on the menu, display any needed dialogue boxes to get user input, and then call the corresponding node's methods to perform the desired function from the menu.
The No Token option from the menu bar is handled by a method that is invoked directly from the message identifier. The method calls methods in the bus and all of the nodes to drop all traces of the token.

The Control Bar

The control bar is located on the screen just under the menu bar and contains the pause, step, and go buttons as well as the simulation speed and bus error scroll bars. Each of these controls has its own identifier and sends its own message identifier when selected. In the TMyWindow object, a separate method is described to handle each of the control's events.

The pause, step, and go button methods simply update boolean values describing the pause state of the simulation when activated. These booleans are used by the TimerTick method to determine whether to perform another step of the simulation or to ignore the current clock pulse from the Windows timer and keep the simulation frozen in its present state.

The scroll bars' methods work similarly. The speed scroll bar method, IDSPEEDScroll, gets the current scroll bar position and updates the displayed speed value to match. The bus error scroll bar method, IDErrorScroll, uses the current scroll bar position to compute an error value based on an exponentially rising function so that the granularity
of the scroll bar error value is very small at low error values and large at large error values. The bus error display is updated and a bus method is then called so the bus can use the new error rate.

The TSimObject Object

The TSimObject object is not used directly in the simulator but is inherited by the bus, link, and node objects. This object provides descendent objects with basic variables such as pen style, color, location, size, and address on the network. It also provides the basic Init and Done methods along with default TimerTick and Draw methods. By inheriting the TSimObject object, descendent objects do not have to have extra, duplicate code and variables to be part of the simulation and be drawn on the screen. These basics are handled by TSimObject and inherited into the more useful objects.

The Bus Object

The bus object simulates the bus of the token bus network and contains two major functionalities. The first is the display of the bus and the data being sent on it. The second functionality is the transfer of transmissions from a transmitting node to all nodes connected to the bus. The TBus object’s TimerTick method coordinates the activities of the bus. Each time TimerTick is called (each active clock pulse of the simulation) it calls the main operational method to perform the bus activities and then
calls the display update method to show the results of the operational activities.

The Bus Display

There are two methods that are called to display the bus on the screen. The UpdateBusDisplay method is called by the TBus object’s TimerTick method to update the display by redrawing any part of the bus that may have changed since the last drawing of the bus. The other method, Draw, is called by TMyWindow’s Paint method to completely redraw the entire bus on the screen.

The UpdateBusDisplay method exists to update the display of the bus as quickly as possible. It first displays the current frame being sent on the bus in its text display area using a large font defined in the Init method. It then calls a subroutine, UpdateLinkDisplay, for each link object in the link object collection. UpdateLinkDisplay checks the output buffer of the node object that corresponds to the current link object. This correspondence is facilitated by the fact that both the node and link collections are sorted by address so a node has the same position in its collection as its corresponding link does. If the node’s transmission status is different than it was on the last clock pulse, the link object’s Draw method is called to update the display of the link to reflect the new transmission status.
The TBus object's Draw method draws the entire bus. To do this, it calls its parent object's, TSimObject, Draw method to draw its outline, calls the Draw method of each of the link objects in the link collection, and then writes the current frame data into the bus's frame display in the large font.

**The Bus Operational Routines**

The bus object is designed to represent the functionalities performed by a real network bus in the simulation. This means that the bus should receive any transmissions being sent by any of the nodes, indicate a contention if more than one node transmits, clear the output buffers of the nodes, possibly insert error into a data transmission, and fill the input buffers of all of the nodes on the bus with the data that is on the bus.

These functions are handled by two methods in the TBus object, UpdateBusState and ClearNodeBuffers. The second method, ClearNodeBuffers, clears out the output buffers of all of the nodes and is called after the bus display update method since the display method needs access to the nodes' display status before the buffers are cleared.

The UpdateBusState method handles all bus activities other than clearing output buffers. It checks the output buffer of each node and transfers the data in it to the bus's buffer if the node is transmitting. If more than one
node transmits, this method sets a contention flag and the data in the bus buffer is set to be garbled and unreadable by the nodes. The method then decides whether or not the transmission on the bus contains an error if the transmission is a data frame. This is done by choosing a random number between 1 and 100. If the random number is less than the current bus error rate controlled by the bus error scroll bar on the screen, then the transmission error flag is set to indicate bus error. The method completes its duties by setting each of the nodes' input buffers to match the bus's buffer.

The Link Object

The link object is called TLink and its main purpose is to show the connection between each node and the bus. Besides its initialization and destruction methods, TLink's only other function is to draw itself using its Draw method.

TLink's Draw method is called from the TBus display methods. During initialization, each link object is given the address of its corresponding node and an orientation value to depict its orientation to the bus on the screen. A link can either be above the bus or below the bus on the display. The TLink Draw method uses this orientation information along with a parameter describing the transmission status of its corresponding node (sending, receiving, or idle) to draw either an arrow pointing toward the bus for the sending status, and arrow pointing toward
the node for the receiving status, or an empty rectangle for the idle status. The orientation information is used to determine which direction is toward the bus and which direction is toward the node, up or down.

The Node Object

The node object contains the majority of the program's code and is responsible for dictating the actions of the network nodes in the simulation. Since token bus networks rely on a distributed control scheme, the behavior of the nodes dictates the behavior of the network. Like the bus object, the node object, TNode, has an operational part and a display part that is coordinated through its TimerTick method. TimerTick calls the operational methods for the node object and then calls the display update method to show the changes brought about by the operational methods.

The operational methods are broken into several subroutines. The UpdateNodeState method is called to update the current state of the node based on the data in the input buffer from the bus and the current action the node is trying to accomplish. UpdateNodeState calls UpdateNodeStatus to process node status changes first. It then calls the generic input processing and counter increment routines, CheckInputBuffer and UpdateCounters, followed by the active node state update routine, UpdateActiveState. UpdateActiveState calls the specialty routines such as the node adding and initialization routines when it is appropriate to do so. UpdateActiveState also calls the token
holding state processing routine, UpdateTokenState, if the node is the token holder.

There are several different counters that are incremented at each clock count. The routine UpdateCounters performs the incrementation. These counters are used for various special purposes such as counting the number of time slots since the bus was last active or counting the number of time slots since data was sent to another node. The counters are used by the special purpose routines to keep track of time intervals without worrying about the details of when to increment the counters.

The Node Display

Like the bus display methods, the node display methods are divided into an update method, UpdateNodeDisplay, called from the TNode TimerTick method and a Draw method that is called from the TMyWindow Paint Method to redraw the entire node.

The UpdateNodeDisplay method redraws only those parts of the node that changed since the last clock pulse. If this method were to simply redraw the whole node each time instead, the simulation would run much slower. Keeping track of the display variable values from the previous iteration and redrawing only those that change increases the speed of the simulation by about four times.

The UpdateNodeDisplay method is responsible for drawing every part of the node except the static text and section
separation lines. This includes the node border and header. The method checks the current status of the node and if it is different from the previous status the border and header may be redrawn in the appropriate color and style to reflect the new status. When UpdateNodeDisplay finishes drawing it updates the previous display variables to reflect the current ones.

The Draw method draws the static text and section separator lines and then calls UpdateNodeDisplay. The Draw method sets a flag before calling UpdateNodeDisplay to tell it to redraw every part of the node, not just those parts that have changed. Doing this ensures that the entire node is redrawn when Draw is called from the TMyWindow Paint method.

The Status Update Routines

The status update routines are part of the UpdateNodeStatus method in the TNode object. Their purpose is to perform the transition of node status from the current status to the status selected by the user from the node's pull-down menu. The routines also set up default activities for the node depending on the node's current action and status. If the node is involved in any special activities such as data sending or token passing, the default activities are overwritten in later processes.
The current state of a node at any time is represented in the program by a series of boolean variables that indicate, for example, whether or not the node owns the token, is trying to add in to the logical ring, is sending data, or is resolving a contention. The status update routines compare the current status (active, inactive, passive, or crashed) of the node to the one selected from the menu and manipulate the state booleans appropriately. Some statuses are changed to the newly requested ones immediately while others cannot be. For example, a node is instantly changed to a crashed status regardless of the current status when the user selects "Crash" from the menu. However, a node is not immediately changed to the inactive status if it is currently an active member of the logical ring. The node must first delete itself from the ring gracefully. In such a case, the active status routine sets a boolean that instructs upcoming routines that the node needs to delete itself from the ring. Once the deletion has occurred, the active status routine confirms that the node is no longer a ring member and sets its current status to inactive.

The Input Buffer Routine

The input buffer routine, CheckInputBuffer, is one of several routines belonging to the UpdateNodeState method. The CheckInputBuffer routine is called before the state
update routines in the `UpdateNodeState` method to perform generic input buffer processing. Any reaction that the node can make to an input without having to worry about the value of special counters such as the contention or token passing counters is done in this routine. The routine uses a case structure to process the input buffer contents based on the current status of the node.

As an example of the type of processing performed by `CheckInputBuffer`, assume that the node is currently active and trying to add itself to the logical ring. The node addition routine keeps track of how many clock counts have expired since the last transmission, how long it should wait for a response, and what the next step should be. However, when the node receives the token, the `CheckInputBuffer` is smart enough to know what to do with the state of the node based on the current status and the receipt of the token. `CheckInputBuffer` calls a routine called `GetToken` at this point to make the node a ring member and the token owner. This action automatically makes the flow of the program pass through the token owner routine instead of the ring addition routine because the boolean variables now indicate that the node is no longer trying to add into the ring.

**Contention Resolution**

The contention resolution routine is part of the `UpdateActiveState` routine which is itself part of the
UpdateNodeState method. Contention occurs on the network when two or more nodes attempt to transmit at the same time. This can occur when the ring is being initialized or when several nodes are trying to add into the logical ring at the same time. When contention occurs, the contention resolution routine, ResolveContention, is called to resolve it.

The ResolveContention routine keeps track of the current pair of address bits being checked to determine the number of clock counts to wait before retransmitting and another counter that keeps track of how many clock counts have gone by since the last transmission. The routine waits either 0, 2, 4, or 6 clock counts before retransmitting depending on whether or not the current pair of address bits is 00, 01, 10, or 11.

The ResolveContention routine is called directly from the node addition and ring initialization routines when they are having contention problems. The contention resolution strategy ends for the node either when ResolveContention hears another node transmit before it or the calling routine sees a successful transmission without contention.

**Logical Ring Initialization**

The ring initialization routine, CheckInitialize, performs ring initialization steps either when the bus has been inactive for more than a constant number of clock
counts (tracked by a special counter), or the node was unable to confirm a successful token pass and is initializing the ring as a last resort. The CheckInitialize routine is called every clock count until the ring is initialized so it checks to see if the node has already started initializing the ring or is just starting on this clock count.

If the node is just starting the initialization process, the routine transmits a claim-token frame and adjusts the state booleans and status variables. If the node is already in the process of initializing it checks for contention on the bus. If there is contention then the contention resolution routine is called each clock cycle until it is resolved. If there is no contention then the node claims the token for itself by calling the GetToken routine.

**Node Addition To The Ring**

When the user of the simulation requests that a node become active, the node must wait for the next solicit-successor frame that it is eligible to respond to and return a set-successor frame to add itself into the logical ring. This function is handled by the CheckAddNode procedure in the UpdateActiveState routine. The IsAddNode boolean state variable is set when the node is trying to add to the ring and stays set until the node receives the token and is part
of the ring. The CheckAddNode routine is called whenever the IsAddNode boolean is set.

CheckAddNode checks the node's input buffer each iteration until it sees a solicit-successor frame. It checks to make sure that the node's address falls within the legal address range for the frame. If it is able to respond to the frame, it responds by transmitting a set-successor frame back to the originating node and updating its predecessor and successor pointers to those in the solicit-successor frame. If the originating node returns the token to the current node, then the CheckInputBuffer routine calls GetToken and CheckAddNode is not called again. If CheckAddNode receives a resolve-contention frame from the originator then there is contention resulting from the set-successor frame. CheckAddNode then sets a contention resolution boolean and calls the ResolveContention routine to resolve the contention problem. If the node ends up losing the contention battle to add to the ring, the predecessor and successor pointers are reset to null. If no reply is heard, CheckAddNode gives up and resets the IsAddNode boolean and predecessor and successor pointers.

**Node Deletion From the Ring**

Node deletion from the logical ring is performed whenever an active node that is a member of the ring is requested to become either passive or inactive. When this
happens, the IsDeleteNode boolean is set to indicate that the node needs to delete itself gracefully from the ring. This boolean is checked in the CheckDeleteNode routine within the UpdateTokenState routine. Therefore, this condition is only checked when the node is the token owner.

The CheckDeleteNode routine sends a set-successor frame to its predecessor with the address of its successor as the set address. On the next clock count the CheckDeleteNode routine is called again but this time it sets the state booleans so that UpdateTokenState can simply pass the token to its successor. Once the token is successfully passed, the UpdateNodeStatus routine updates the node's status to its requested passive or inactive status.

There is one other situation that causes the CheckDeleteNode to delete the node from the logical ring. If the user requests a node address change then the node is deleted from the logical ring before the address is changed.

**The Solicit Successor Routine**

The solicit successor routine, CheckSolicitSucc, is called from the UpdateTokenState routine to solicit a successor whenever the node is beginning ring initialization or is performing a routine solicit to allow other nodes the chance to add to the ring. The first condition is determined by the setting of a boolean in the ring initialization routine while the second condition is kept
track of through the use of a special counter that keeps track of the number of token possessions since the last solicit-successor frame was sent.

The CheckSolicitSucc routine is called on each token possession by the node. If the routine determines that a solicit-successor frame should be sent it updates the state booleans to reflect the fact that the node is soliciting a successor and sends a solicit-successor frame to all nodes using the appropriate range of addresses for a legal response. After waiting two time slots, the CheckSolicitSucc routine either sees a set-successor frame addressed to it, sees no frame on the bus, or senses contention.

If there is a set-successor frame addressed to it, the routine calls the token passing routine to pass the token to the sender of the set-successor. The CheckInputBuffer routine adjusts the node's successor pointer. If nothing is heard on the bus and the solicit was a scheduled one, then the routine resets the booleans to stop soliciting a successor. If nothing is heard on the bus and the node is initializing the bus then the routine calls the DropToken routine to update the state boolean flags to indicate that the node is no longer the token owner.

If there is contention resulting from the solicit responses then the CheckSolicitSucc routine sends a resolve-
contention frame to all nodes on the bus. At this point it is up to the other nodes to resolve the contention. The current node just keeps track of how long it is taking for the other nodes to resolve the contention using a special counter. If there is still a contention problem after four contention rounds the CheckSolicitSucc routine retransmits the solicit-successor frame. If the contention is not resolved after four more contention rounds the routine either calls DropToken if the node is initializing the ring or calls PassToken if it was a scheduled solicit.

The Data Sending Routine

The data sending routine is called by the UpdateTokenState routine to send a data frame to another node when the user requests the node to do so either through the Send To or AutoSend option from the node pull-down menu. The routine responsible for sending data and ensuring that the data is successfully received by the destination node is the SendData routine.

The SendData routine relies on the CheckInputBuffer routine to process acknowledgements from the destination node. After sending the data to the destination node, the SendData routine waits two time slots for an acknowledgement. If one is received, the CheckInputBuffer routine updates the state booleans and SendData is not called again. If a negative acknowledgement is received or
the constant number of time slots has expired since the transmission, the routine resends the data frame and waits again. sendData keeps track of how many times it has retried the data transmission. If the number of retransmissions exceeds a constant number, then the sendData routine gives up and resets the state booleans to reflect that it is no longer trying to confirm a data transmission.

The Token Passing Routine

Whenever the node is the token holder and does not need to send data, solicit a successor, or delete from the logical ring it passes the token to its successor. The function of passing the token to the successor and confirming that the successor properly receives the token is the job of the PassToken routine. PassToken is mainly called from the UpdateTokenState routine but may be called by other specialty routines that need to pass the token.

The PassToken routine first checks to see whether it needs to send the token to the successor or whether it needs to confirm a previous transmission. Whenever it needs to transmit the token to the successor it calls the PutToken subroutine to perform the transmit and update the state booleans. When confirming a token pass, the PassToken routine updates the state booleans to reflect that no more confirmation is needed after noticing a valid frame from the successor frame. If a set-successor frame is received
after PassToken issues a who-follows frame, the CheckInputBuffer routine notices the frame on the bus and resets the state variables to pass the token to the originator of the set-successor frame. Therefore, whenever the PassToken routine is being executed, it is because the token pass has not yet been confirmed.

There is a special counter that keeps track of how many time slots have expired since the token was first transmitted. The PassToken routine is structured as a case statement based on the counter value. After two time slots, the PassToken routine resends the token to the successor. If there is no confirmation after two more time slots, when the counter is equal to four, PassToken sends a who-follows frame to all nodes to find the successor to the current node's successor. If the counter reaches six it is because CheckInputBuffer did not receive a set-successor frame so PassToken reissues the who-follows frame once more.

After two more time slots with no valid response, PassToken sets the state booleans and the solicit successor counter to make the UpdateTokenState routine solicit a successor using the complete range of addresses for respondents. If there is no valid response after three more time slots PassToken assumes that something major has gone wrong, makes itself the token owner again, and attempts
solicit a successor through the ring initialization procedures.

**Duplicate Node Address Resolution**

The user of the simulator has the option of changing the address of any node using the address option from the node pull-down menus. This option allows the user to create duplicate addresses between different nodes which can, in turn, create a multiple token scenario. When two nodes with the same address both receive the token, then they both consider themselves token owners creating multiple tokens on the network.

The multiple token problem is resolved by the CheckInputBuffer routine. When it senses another node transmitting while the current node owns the token it calls DropToken to change the node's state booleans so the node no longer owns the token and is no longer part of the logical ring. This leaves the two or more nodes with duplicate addresses outside of the logical ring trying to get back in. When a solicit-successor frame is issued that the nodes can respond to they all contend because they all have the same address. This contention can not be resolved and continues past the four legal contention rounds. If a fifth contention round is entered while in the CheckAddNode routine and there is still a contention problem, each node resets its current address to equal a random number from 1
to 99. Each of the nodes then has a new chance to enter the
ring when a solicit-successor frame allowing their address
is issued. If the random address selection results in
further duplicate addresses, they are resolved in the same
manner when the nodes try to add into the logical ring
again.
RESULTS

The results consist of a series of sample screens from the program that demonstrate the actions of the simulation during different scenarios. The scenarios shown are logical ring initialization with contention, token passing, token passing to a crashed node, data sending, node deletion from the logical ring, and duplicate address resolution. Node addition into the logical ring is shown in the first scenario for ring initialization when the token owner solicits a successor to form the ring.

Scenario 1: Logical Ring Initialization With Contention

These sample screens show four active nodes trying to initialize the logical ring. Since they all try at the same time there is a contention problem to resolve. After resolving the contention, Node 2 gets the token and solicits a successor. The remaining nodes try to respond to the solicit-successor frame and there is another contention problem to resolve. The token owner issues a resolve-contention frame to do this. The winner of the contention round, Node 5, issues its set-successor frame and becomes part of the newly formed ring. This scenario is depicted in Figures 6 through 12.
Figure 6. Scenario 1 - Ring Initialization Attempt With Contention
### Figure 7. Scenario 1 - Claim Token Frame
Figure 8. Scenario 1 - Solicit-Successor Frame
Figure 9. Scenario 1 - Set-Successor Contention
Figure 10. Scenario 1 - Resolve-Contention Frame
Figure 11. Scenario 1 - Resolve-Contention Frame

Contention
Figure 12. Scenario 1 - Set-Successor Frame
Scenario 2: Token Passing

This scenario shows a simple round of normal token passing where none of the nodes has any data to send. This scenario is shown in Figures 13 through 16.
Figure 13. Scenario 2 - Token Pass Screen 1
Figure 14. Scenario 2 - Token Pass Screen 2
**Figure 15. Scenario 2 - Token Pass Screen 3**
**Figure 16. Scenario 2 - Token Pass Screen 4**
Scenario 3: Token Passing To A Crashed Node

In this scenario, Node 5 is trying to pass the token to Node 7 but Node 7 is crashed and cannot respond. Node 5 issues a who-follows frame to find out who follows Node 7 in the logical ring. Node 8 is Node 7's successor and answers the who-follows frame with a set-successor frame. Node 5 responds by passing the token to node 8 and resetting its successor pointer. This scenario is shown in Figures 17 through 20.
### Figure 17. Scenario 3 - Token Pass To Crashed Node
Figure 18. Scenario 3 - Who-Follows Frame
**Set Successor**

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 1</td>
<td>Address: 2</td>
<td>Address: 3</td>
<td>Address: 4</td>
</tr>
<tr>
<td>Next Addr: 0</td>
<td>Next Addr: 5</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
</tr>
<tr>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
</tr>
<tr>
<td>Status: Inactive</td>
<td>Status: Active</td>
<td>Action: Wait</td>
<td>Action: Wait</td>
</tr>
<tr>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node 5</th>
<th>Node 6</th>
<th>Node 7</th>
<th>Node 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 5</td>
<td>Address: 6</td>
<td>Address: 7</td>
<td>Address: 8</td>
</tr>
<tr>
<td>Next Addr: 7</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 2</td>
</tr>
<tr>
<td>Prev Addr: 2</td>
<td>Prev Addr: 0</td>
<td>Status: Inactive</td>
<td>Prev Addr: 0</td>
</tr>
<tr>
<td>Status: Active</td>
<td>Status: Inactive</td>
<td>Status: Down</td>
<td>Status: Active</td>
</tr>
<tr>
<td>Action: Pass Token</td>
<td>Action: Wait</td>
<td>Action: Crash</td>
<td>Action: Listen</td>
</tr>
<tr>
<td>Step: Confirm</td>
<td>Step: Wait</td>
<td>Step: Wait</td>
<td>Step: Set Successor</td>
</tr>
<tr>
<td>Cs: 0 Cw: 0 Cnt: 5</td>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
<td>Cs: 0 Cw: 0 Cnt: 0</td>
</tr>
</tbody>
</table>

*Figure 19. Scenario 3 - Set-Successor Frame*
**Figure 20. Scenario 3 - Token Pass To Successor**
Scenario 4: Data Sending

In this scenario Node 3 has data to send to Node 6, a passive listener. Node 6 receives the data successfully and returns an acknowledgement. Node 3 sees the acknowledgement and passes the token to its successor. This scenario is shown in Figures 21 through 24.
Figure 21. Scenario 4 - Data Sent To Receiving Node
Figure 22, Scenario 4 - Acknowledgement Sent From Receiver
Figure 23. Scenario 4 - Token Passed To Successor
Scenario 5: Node Deletion From The Logical Ring

In this scenario, node 2 wants to become inactive so it deletes itself from the logical ring. To do this, it sends a set-successor frame to its predecessor with the address of its successor. It then sends the token to its successor and changes its status to inactive. This scenario is shown in Figures 24 through 26.
Figure 24. Scenario 5 - Set-Successor Frame
Figure 25. Scenario 5 - Token Pass
**Figure 26. Scenario 5 - Change To Inactive Status**

![Token Bus LAN Simulator Diagram](image)

<table>
<thead>
<tr>
<th>Node 1</th>
<th>Node 2</th>
<th>Node 3</th>
<th>Node 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 1</td>
<td>Address: 2</td>
<td>Address: 3</td>
<td>Address: 4</td>
</tr>
<tr>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
<td>Next Addr: 0</td>
</tr>
<tr>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
<td>Prev Addr: 0</td>
</tr>
<tr>
<td>Status: Inactive</td>
<td>Status: Inactive</td>
<td>Status: Taken</td>
<td>Status: Inactive</td>
</tr>
<tr>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
</tr>
<tr>
<td>Cw: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node 5</th>
<th>Node 6</th>
<th>Node 7</th>
<th>Node 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address: 5</td>
<td>Address: 6</td>
<td>Address: 7</td>
<td>Address: 8</td>
</tr>
<tr>
<td>Next Addr: 7</td>
<td>Next Addr: 0</td>
<td>Next Addr: 8</td>
<td>Next Addr: 3</td>
</tr>
<tr>
<td>Status: Active</td>
<td>Status: Passive</td>
<td>Status: Active</td>
<td>Status: Active</td>
</tr>
<tr>
<td>Action: Listen</td>
<td>Action: Listen</td>
<td>Action: Listen</td>
<td>Action: Pass Token</td>
</tr>
<tr>
<td>Step: Listen</td>
<td>Step: Listen</td>
<td>Step: Listen</td>
<td>Step: Confirm</td>
</tr>
<tr>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
<td>Cs: 0</td>
</tr>
<tr>
<td>Cw: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
<td>Cnt: 0</td>
</tr>
</tbody>
</table>

---
Scenario 6: Duplicate Address Resolution

In this scenario, Node 1 and Node 6 have the same address and both receive the token. When they both send the token they notice the contention and drop the token and get out of the logical ring. When trying to add back into the ring they run into a contention that they cannot resolve since their addresses are the same. At this point they get new, random addresses and continue. This scenario is shown in Figures 27 through 30.
Figure 27. Scenario 6 - Duplicate Addresses Both Send Token
Figure 28. Scenario 6 - Duplicate Addresses Back Off
Figure 29. Scenario 6 - Duplicate Addresses Contend Trying

To Add
Figure 30. Scenario 6 - Duplicate Address Nodes Get New Addresses
DISCUSSION

Programming Approach

There are several other programming styles and environments that would be appropriate for a token bus simulator. The program does not need to be made object-oriented, does not need to use Windows, or even be written in Pascal. A token bus simulator could even be written as a text based simulator without the use of graphics.

The object-oriented approach is chosen to make Windows programming easier and to divide the different parts of the simulation into logically separate entities within the program. The bus, links and the nodes could have been written using Pascal or C record structures to contain the data associated with each of the entities. The link and node records could be maintained in arrays that could be accessed using For loops instead of storing the node and link objects in sorted object collections and accessing them via the collections’ ForEach methods. Instead of using object methods, separate subroutines could be used to access the record data and perform the tasks associated with each part of the simulation.

Even if Microsoft Windows is not used as a basis for the graphical user interface and a graphical user interface is still desired for the simulation, Borland’s Turbo Pascal version 6.0 contains a set of objects called TurboVision which allows the programmer to create a mouse based graphical user interface using regular IBM DOS. The interface is not nearly as elaborate or aesthetically pleasing as the Windows interface
and can be at least as hard to program. Another option for a user interface is to have a second screen that contains a series of menus to make selections from. From a programming standpoint, this option is much simpler to implement than either of the graphical user interfaces. However, for a simulator such as this one, it is much more convenient from a user standpoint to have a graphical user interface incorporated into the simulation.

**Simulator Uses**

There are many other uses for a simulator such as this one besides using it as a vehicle for demonstrating an understanding of token bus local area networks by the programmer. The simulator is an excellent educational tool for demonstrating how a token bus network works. The students get a chance to see the workings, scenarios, problems, and functions associated with a token bus first hand in a visual, interactive format.

This educational aid would be useful in the classroom for a data communications, computer architecture, or local area network type of class. The instructor could explain the workings of a token bus local area network and then demonstrate those concepts with a personal computer running the simulator. This really becomes an exciting learning tool when students wish to know how a token bus LAN handles certain situations and are able to see the answers to their questions being acted out in a simulation. It would also be useful to the students for the professor to reserve a little time at the end of the class to allow students to try out the simulator for themselves and get
a first hand feel for the concepts at work in a token bus local area network.

Another practical use for such a simulator is as a testbed for new ideas. For example, the scheme included in the simulation to resolve the duplicate address problem is not part of the IEEE 802.4 standard. It was included after the rest of the simulation was complete to test out a possible solution to the duplicate address problem. Not only does the simulator prove the feasibility of such an idea, it lets the programmer see the idea in action to analyze the strengths and weaknesses of the design. Changes to the design can be made quickly and the results of the changes can immediately be viewed in operation in the simulation.

**Program Expansion: Performance Monitoring**

The fact that this simulator is written using an object-oriented design allows for the design to be expanded upon easily by writing new objects that inherit the traits of the current ones. One addition to the current simulator that could prove very useful is the addition of network performance monitoring. The propagation delay between pairs of nodes could be taken into account and the clock pulse in the simulation could represent the time it takes a transmission to go from one node to an adjacent node on the bus instead of having one propagation for all pairs of nodes. Performance measures such as node specific and network throughput and utilization could be measured under different load conditions.
The displays for the node throughput and utilization could be added to the status display area of each of the nodes by making each node take up a slightly larger portion of the screen. The network throughput and utilization display could be added at the bottom of the screen or next to the bus display. The load provided by each node could be controlled through the AutoSend function. The current version of the AutoSend function sends data to other nodes on the network on every token possession of the current node. This could be changed so the user could specify the percentage of time that the node would send data to other nodes. In fact, a scroll bar could even be added to the control bar to control the minimum percentage of the time that every node must send data to other nodes. In this case, if all nodes were active and the scroll bar were set to 80%, then each node would have data to send 80% of the time. If this scroll bar were set to 100%, then every active node would send data at each token possession.

Program Extension: CSMA/CD Simulator

Another extension that could be made to the simulator is to create a similar CSMA/CD network simulator using the current bus, node and link objects as the basis for the new objects. The current SimObject traits would be inherited as would many of the variables and methods added by the token bus network node such as the counter and display related variables and the data sending methods. The bus object may be able to be inherited by a CSMA/CD bus object with little modification since the bus still performs the same basic functions such as getting all node transmissions, determining contention status and data error, filling the
node input buffers, and updating the bus display to show the current frame on the bus. The largest change would be, of course, in the node's UpdateNodeState method since this is the method that determines the actions of the node and would make it behave like a CSMA/CD node as opposed to a token bus node. The UpdateNodeDisplay method would have to be overridden also to display the information relevant to a CSMA/CD node.

The simulator could be written so the user could switch back and forth between the two token bus and CSMA/CD network types to compare their actions. Doing this would most likely mean that only one type of network could be shown on the screen at a time unless each network type were shown as a four node network on the screen or each of the eight nodes of each network were made much smaller showing much less status information. If the performance monitoring enhancement is included in the CSMA/CD enhancement, the simulator could give performance comparisons between the two network protocols and even produce graphs or charts to display the differences. This type of option would probably be made easier if both simulations were run concurrently with each being shown on the screen at once. A performance comparison menu could be added to the pull-down menus to control the comparison choices and allow the user to pick from a selection of graph or chart types.
CONCLUSIONS

The token bus simulator does a very good job of recreating the operations and scenarios that exist in an actual token bus local area network. It simulates all scenarios described in the IEEE 802.4 token bus standard including multiple tokens, lost token, node failure, and normal operational scenarios. While simulating these scenarios, it gives the user a clear picture of what actions are being taken on the network.

The object-oriented approach to programming the simulator provided many advantages that made the effort needed to learn the object-oriented programming style worthwhile. The object-oriented technique helped to define the functionalities of each part of the simulation and also helped to keep the data and processing duties of the different objects separate. The Microsoft Windows programming environment also took a good deal of effort to learn but it was worth the effort to gain the advantages it provided to the final product.

Lastly, one of the most important parts of the simulation program turns out to be its overall design and flow. One of the more difficult parts of creating the simulator was deciding how to break up the responsibilities of the nodes into easily workable, logical subroutines. There are so many functions that a node performs that rely on or share common functionality with other functions that the overall design of the simulator has a great effect on the overall ease of programming, expanding and maintaining the simulator program. The design for this project seems to separate the node functionalities quite well.
SUMMARY

This project includes the design and implementation of an IBM PC based simulator for a token bus local area network. The program makes use of the Microsoft Windows programming environment, Borland's Turbo Pascal for Windows, and an object-oriented style of programming. The simulator performs all of the basic functionalities for a token bus network as described by the IEEE 802.4 standard for token bus local area networks. It allows the user complete control of the graphically based simulation through a mouse controlled graphical user interface that includes control over node status, data sending, address changing, and lost token scenarios. The user interface also controls the pause state the simulation, the simulation speed, and the error associated with data transmitted on the bus.

Although there are many other ways a token bus simulation could be designed, it is deemed that the choice of using Microsoft Windows with object-oriented programming combined with the special attention to ensure a clean separation of functionalities in the program enhanced the simulator design and operation considerably.
LITERATURE CITED


APPENDIX: PROGRAM CODE

The Appendix contains the Pascal code that makes up the simulator program. Comments and pseudocode are included with the Pascal code to help clarify the logic of the program.
program TokenBusSimulator;

uses GGUI,   {Graphical User Interface unit for simulation}
    WObjects;   {ObjectWindows Object definitions}

{
  TApplication Object Description

  type
      TMyApplication = object(TApplication)
          procedure InitMainWindow; virtual;
  end;
}

{*******************************************************************************
  TMyApplication’s Method Implementations
*******************************************************************************}

{
  Procedure TMyApplication.InitMainWindow
  *
  *    Defines the main window to be TMyWindow with the Token Bus title
  ******************************************************************************
}
procedure TMyApplication.InitMainWindow;
begin
    MainWindow := New(PMyWindow, Init(nil, 'Token Bus LAN Simulator'));
end; {TMyApplication.InitMainWindow}

{*******************************************************************************
  Main Program
*******************************************************************************}

var
    MyApp : TMyApplication;

begin
    MyApp.Init('TokenBusSimulator');
    MyApp.Run;

MyApp.Done;
end.
unit GGUI;

{*******************************************************************************}

Unit GGUI
This unit contains the graphical user interface for the Token Bus Simulation program. It links in a menu resource called Token.RES as the menu for the interface. There is one object described in this unit, TMyWindow, the main window for the Token Bus Application. TMyWindow calls the node and bus methods to communicate current user choices and timer ticks to these objects.
{*******************************************************************************}

interface
uses Strings, {Turbo String handling routines}
    WinTypes, {Windows Type declarations}
    WinProcs, {Windows Procedure declarations}
    WObjects, {ObjectWindows Window object definitions}
    StdDIs, {ObjectWindows Dialogue object definitions}
    GGlobal, {Global data definitions}
    GNodeObj, {Node object definition}
    GBusObj; {Bus object definition}

{$R \Token\Token.RES} {The token bus menu bar resource}

{*******************************************************************************}
| TMyWindow Object Description  // |
{*******************************************************************************}
type
TMyWindow = ^TMyWindow;
TMyWindow = object(TWindow)
    PaintBrush: HBrush; {Brush used to paint control bar}
    PaintPen: HPen; {Pen used to paint control bar}
    TheDC: HDC; {The window's display context}
    PauseChosen, StepChosen: boolean; {The Pause button was pushed}
    SpeedValue: integer; {Value from the Speed scroll bar}
    ErrStr, SpeedStr: array[0..5] of char; {Error Display Text}
    array[0..5] of char; {Speed Display Text}
    PTempButton: PButton; {Temp pointer to create buttons}
    PSpeedScroll, PErrorScroll: PScrollBar; {Pointer to the Speed scroll bar}
    PTempStatic, PSpeedStatic, PErrorStatic: PStatic; {Pointer to Error display static}
PTTheBus: \{Pointer to the Bus\}

PTBus;

constructor Init(AParent: PWindowsObject; ATitle: PChar);

destructor Done; virtual;

procedure SetupWindow; virtual;

function GetClassName: PChar; virtual;

procedure GetWindowClass(var AWndClass: TWndClass); virtual;

procedure Paint(PaintDC: HDC; var PaintInfo: TPaintStruct); virtual;

procedure WMTimer(var Msg: TMessage);

virtual wm_First + wm_Timer;

procedure WMCommand(var Msg: TMessage);

virtual wm_First + wm_Command;

procedure IDSSpeedScroll(var Msg: TMessage);

virtual id_First + id_SpeedScroll;

procedure IDErrorScroll(var Msg: TMessage);

virtual id_First + id_ErrorScroll;

procedure Pause(var Msg: TMessage);

virtual id_First + id_Pause;

procedure Step(var Msg: TMessage);

virtual id_First + id_Step;

procedure Go(var Msg: TMessage);

virtual id_First + id_Go;

procedure CMNoToken(Msg: TMessage);

virtual cm_First + cm_NoToken;

end;

implementation

{xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
******** TMyWindow's Method Implementations ********
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx}

{

******** TMyWindow's Method Implementations ********
*
* Constructor TMyWindow.Init
*
* Initializes main window variables, buttons, statics, and scrollers.*
********
}

constructor TMyWindow.Init(AParent: PWindowsObject; ATitle: PChar);
var
    Result: Integer;

begin
{
    |----------------------------------------------|
    | Call parent Init method |
    | Initialize random number generator |
Initialize Speed value, and button choice variables
Load the menu definition into the window
Create the node collection and allow duplicate addresses

TWindow.Init(AParent, ATitle);
Randomize;

SpeedValue := 0;
StepChosen := false;
PauseChosen := false;
StrCopy(ErrStr, '0.0%');
StrCopy(SpeedStr, '0%');
Attr.Menu := LoadMenu(HInstance, PChar(id_Menu));
PNodeCollection := New(PSortedAddressCollection, Init(8, 0));
PNodeCollection^.Duplicates := true;
PaintBrush := GetStockObject(LtGray_Brush);
PaintPen := GetStockObject(Null_Pen);
{
| Create the Pause, Step, and Go buttons
|---------------------------------------------------------------|
PTempButton := (New(PButton,
    Init(@Self, id_Pause,'Pause \', C_ButtonPos, 3, 70, 26, False)));
PTempButton := (New(PButton,
    Init(@Self, id_Step,'Step >>', C_ButtonPos+70, 3, 70, 26, False)));
PTempButton := (New(PButton,
    Init(@Self, id_Go,'Go >>', C_ButtonPos+140, 3, 70, 26, False)));
{
| Create the Speed and Error scroll bars
|---------------------------------------------------------------|
PSpeedScroll := (New(PScrollBar,
    Init(@Self, id_SpeedScroll, C_ScrollPos, 3, 120, 26, True)));
PErrorScroll := (New(PScrollBar,
    Init(@Self, id_ErrorScroll, C_ErrScrollPos, 3, 120, 26, True)));
{
| Create divider line static
|---------------------------------------------------------------|
PTempStatic := New(PStatic,
    Init(@Self, id_SepStatic, '', 0, 32, 700, 1, 0));
PTempStatic^.Attr.Style := PTempStatic^.Attr.Style
    and not SS_LEFT or SS_BLACKRECT;
end;

***************

* Destructor TMyWindow.Done
* Called upon close of main window to delete Node Collection, Bus,
* and timer objects. It then calls Done to close window.
*
{                                                                                   
  destructor TMyWindow.Done;                                                       
  begin                                                                              
    Dispose(PNodeCollection, Done);                                                 
    KillTimer(HWindow, 1);                                                          
    PTheBus^.Done;                                                                  
    TWindow.Done;                                                                   
  end; {TMYWindow.Done}                                                             
                                                                                   
{                                                                                   
  */ Function TMyWindow.GetClassName                                                 */
  /* Returns the class name of this window. */                                       
  */                                                                                   
  function TMyWindow.GetClassName: PChar;                                           
  begin                                                                              
    GetClassName := 'TokenWindow';                                                   
  end; {TMyWindow.GetClassName}                                                     
                                                                                   
{                                                                                   
  */ Procedure TMyWindow.GetWindowClass                                             */
  /* Override to link in token bus icon. */                                         
  */                                                                                   
  procedure TMyWindow.GetWindowClass(var AWndClass: TWndClass);                    
  begin                                                                              
    TWindow.GetWindowClass(AWndClass);                                              
    AWndClass.IIcon := LoadIcon(Hinstance, 'TokenIcon');                            
  end; {TMyWindow.GetWindowClass}                                                   
                                                                                   
{                                                                                   
  */ Procedure TMyWindow.SetupWindow                                               */
  /* Called after TMyWindow.Init to initialize menu selection checkboxes */        
  /* setup the window timer, and to initialize the Node Collection. */             
  */                                                                                   
  procedure TMyWindow.SetupWindow;                                                 
  var                                                                                
    Result: integer;                                                                
  begin                                                                              

{  
  Call the parent object SetupWindow method
  Remove checks next to all of the Active, Passive, Crash AutoSend/
  menu selections for all of the nodes' pull-down menus
  
  TWindow.SetupWindow;
  CheckMenuItem(Attr.Menu, cm_N1_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N2_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N3_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N4_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N5_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N6_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N7_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N8_Active, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N1_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N2_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N3_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N4_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N5_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N6_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N7_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N8_Passive, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N1_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N2_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N3_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N4_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N5_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N6_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N7_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N8_Crash, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N1_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N2_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N3_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N4_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N5_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N6_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N7_AutoSend, mf_Unchecked);
  CheckMenuItem(Attr.Menu, cm_N8_AutoSend, mf_Unchecked);
  
  {  
    Set the Windows System Timer to send Timer messages to the main /
    window at MaxTimerDelay intervals.
    While the timer setup was not successful
    give the user the option of retrying or quitting
  }  
}

Result := IDRetry;
while (SetTimer(hWnd, 1, C_MaxTimerDelay, nil) = 0) and (Result = IDRetry) do
  Result := MessageBox(GetFocus, 'Could not Create Timer', 'Token Bus Simulator',
    mbRetryCancel);
if Result = IDCANCEL then PostQuitMessage(0);
{
  Initialize each of the Nodes and place each one into the / Node Collection.
  Create the Bus object and initialize it.

  With PNodeCollection^ do
  begin
    Insert(New(PTNode, Init(@Self, N1_X, N1_Y, N1_Address, '1')));
    Insert(New(PTNode, Init(@Self, N2_X, N2_Y, N2_Address, '2')));
    Insert(New(PTNode, Init(@Self, N3_X, N3_Y, N3_Address, '3')));
    Insert(New(PTNode, Init(@Self, N4_X, N4_Y, N4_Address, '4')));
    Insert(New(PTNode, Init(@Self, N5_X, N5_Y, N5_Address, '5')));
    Insert(New(PTNode, Init(@Self, N6_X, N6_Y, N6_Address, '6')));
    Insert(New(PTNode, Init(@Self, N7_X, N7_Y, N7_Address, '7')));
    Insert(New(PTNode, Init(@Self, N8_X, N8_Y, N8_Address, '8')));
  end; {with}
  PTheBus := New(PTBus, Init(@Self));
end; {TMyWindow.SetupWindow}

{

*****************************************************************************
* Procedure TMyWindow.WMTimer                                      *
* Calls node's TimerTick method for each wm_Timer message received  *
* from the Window's System                                           *
*****************************************************************************
}
procedure TMyWindow.WMTimer(var Msg: TMessage);
  procedure NodeTimers(PNode: PTNode); far;
  begin
    PNode^.TimerTick(TheDC);
  end; {NodeTimers}
begin
{
  If the (speed value is positive or the Step button was pushed) / and we are not paused
    Get a display context for the window
    Call each of the nodes' TimerTick methods
    Call the Bus's TimerTick method
Release the display context
EndIf
If the Step button was pushed
Pause the simulation
EndIf

if ((SpeedValue > 0) or (StepChosen)) and (not PauseChosen) then
begin
begin
TheDC := GetDC(HWNDWindow);
PNODECollection^ .ForEach (@NodeTimers);
PTheBus^ .TimerTick(TheDC);
ReleaseDC(HWNDWindow, TheDC);
end; {if}
end; {if}
if StepChosen then
PauseChosen := true;
end; {TMyWindow.WMTimer}

************

* Procedure TMyWindow.WMCommand *
* This procedure is being redefined to handle the Node Command menu *
* selections since there are 56 of them that would otherwise need to *
* be handled by 56 separate procedures. *

************

procedure TMyWindow.WMCommand(var Msg: TMessage);
var
Temp: array[0..3] of char;
InText: array[0..2] of char;
PTThisNode: PTreeNode;
NodeIndex,
ErrorPos,
ReturnVal: integer;
CommandIndex: T_NodeCommand;
Found: boolean;
StatFlag: word;

----------------------------------------

Function AValidNode
Determines if N is a valid node to 'SendTo'.
A node is valid if its address belongs to a member of the node collection but does not match the current node's address.

----------------------------------------
TempInt: integer;
PNode: PNode;

begin
IsValid := false;
if N <> PThisNode^.Address then
    IsValid := PNodeCollection^.Search(@N, TempInt);
AValidNode := IsValid;
ed;

{--------------------------------------------------}

begin
{

*** The menu selection results in an integer of the menu choice.
*** What we need, though, is the index of the node (1 - 8) and the
*** name of the command (Active, Inactive, etc) that goes with the
*** integer. To do this, we loop through all the node indices and
*** command names until we find a set that uses the integer for a
*** menu identifier.

Call the parent object's WMCommand method
If the menu item chosen is in one of the node menus
    Set node index to zero and the command index to first command
    While we haven't found the command no. matching the one chosen/
        and we still have node menus to look through
        Increment the node index
        Set the command Index to the first command on the node menu
        While we have not found command no. matching the one chosen/
            and there are still commands on the menu to search
            Set the command index to its successor
    EndWhile
EndWhile

TWindowsObject.WMCommand(Msg);

if Msg.WParam < cm_NoToken then
begin
    NodeIndex := 0;
    CommandIndex := nc_Active;
    Found := false;

    while ((not Found) and (NodeIndex <= 8)) do
begin
    inc(NodeIndex);
    CommandIndex := nc_Active;
    Found := (Msg.WParam = NodeCommand[NodeIndex, CommandIndex]);
    while ((not Found) and (CommandIndex < nc_Address)) do
    begin

CommandIndex := succ(CommandIndex);
  Found := (Msg.WParam = NodeCommand[NodeIndex, CommandIndex]);
end; {while CommandIndex}
end; {while NodeIndex}
{
--------------------------------------------------------------------------
  *** At this point we have the command name (since it is the same as
  *** its index) and a node index. We need to go through all
  *** nodes in the collection to find one with a matching address and
  *** use the pointer to that node to call its methods.
--------------------------------------------------------------------------

If the node and command indices were found
Get pointer to the node in the node collection with this index (-1)
  Case command index of
    Case command index = Active:
      Check Active menu selection and uncheck other commands
      Call SetCommand method for this node to set it to Active
    Case command index = Inactive:
      Check Inactive selection and uncheck other commands
      Call SetCommand method for this node to set it to Inactive
--------------------------------------------------------------------------;

if Found then
begin
  PThisNode := PNodeCollection^.(NodeIndex - 1);
  case CommandIndex of
    nc_Active: begin
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Active], mf_Checked);
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Inactive], mf_Checked);
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Passive], mf_Checked);
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Crash], mf_Checked);
      PThisNode^.SetCommand(CommandIndex);
    end; {Active}

    nc_Inactive: begin
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Active], mf_Checked);
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Inactive], mf_Checked);
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Passive], mf_Checked);
      CheckMenuItem(Attr.Menu,
      NodeCommand[NodeIndex, nc_Crash], mf_Checked);
      PThisNode^.SetCommand(CommandIndex);
    end; {Inactive}
{
--------------------------------------------------------------------------;
  Case command index = Passive:

Check Passive selection and uncheck other commands
Call SetCommand method for this node to set it to Passive

Case command index = Crash:
Check Crash menu selection and uncheck other commands
Call the SetCommand method for this node to set it to Crash

\[
\text{nc\_Passive: begin}
\]
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Active], mf\_Unchecked);
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Inactive], mf\_Unchecked);
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Passive], mf\_Checked);
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Crash], mf\_Unchecked);
PThisNode\^\^\_.SetCommand(CommandIndex);
end; \{Passive\}

\[
\text{nc\_Crash: begin}
\]
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Active], mf\_Unchecked);
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Inactive], mf\_Unchecked);
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Passive], mf\_Unchecked);
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_Crash], mf\_Checked);
PThisNode\^\^\_.SetCommand(CommandIndex);
end; \{Crash\}

\[
\text{Case command index = AutoSend:
Get current status (checked or unchecked) of AutoSend / menu selection}
\]
If menu item is checked (We are changing to unchecked)
Uncheck the menu item
Call node's MakeAutoSend procedure to turn off AutoSend
Else (we are changing it to checked)
Check the menu item
Call node's MakeAutoSend procedure to turn on AutoSend
EndIf

\[
\text{nc\_AutoSend: begin}
\]
StatFlag := GetMenuState(Attr.Menu, Msg\_WParam, mf\_ByCommand);
StatFlag := StatFlag and mf\_Checked;
if StatFlag <> 0 then
begin
CheckMenuItem(Attr.Menu,
NodeCommand[NodeIndex, nc\_AutoSend], mf\_Unchecked);
PThisNode\^\^\_.MakeAutoSend(false);
end {if}
else
begin
CheckMenuItem(Attr.Menu, 
NodeCommand[NodeIndex, nc_AutoSend], mf_Checked);
PThisNode^.MakeAutoSend(true);
end; {else}
end; {AutoSend}

Case command index = SendTo
PopUp DialogueBox asking user to enter node to send to
Convert the number entered from text to integer
Call AValidNode to ensure node is valid one to send to
If the node is valid
Call node's MakeSendTo method to make node send data/
to the node entered by the user
EndIf

nc_SendTo:begin
StrCopy(InText, ' '); 
if Application^.ExecDialog(New[PInputDialog, 
Init(@Self, 'Send To', 'Enter The ADDRESS Of
Node To SEND TO',
    InText, SizeOf(InText)))) = id_Ok then
begin
Val(InText, ReturnVal, ErrorPos);
if not AValidNode(ReturnVal) then
    ErrorPos := 1;
if ErrorPos = 0 then
    PThisNode^.MakeSendTo(ReturnVal);
end {if}
else
    ErrorPos := 0;
end {while number entered is not valid and the user did not/
choose to cancel out of the dialogue box
Pop Up a Dialogue Box asking the user to enter a node
Convert the number entered from text to integer
Call AValidNode to ensure node's valid to send to
If the node is valid
Call node's MakeSendTo method to make node send/
data to the node entered by the user
EndWhile

while ErrorPos <> 0 do
begin
if Application^.ExecDialog(New[PInputDialog, 
Init(@Self, 'Send To', 'Must Be VALID
ADDRESS Other Than Sender',
...
InText, SizeOf(InText))) = id_OK then
  begin
    Val(InText, ReturnVal, ErrorPos);
    if not AValidNode(ReturnVal) then
      ErrorPos := 1;
    if ErrorPos = 0 then
      PThisNode^.MakeSendTo(ReturnVal);
  end; {if}
else
  ErrorPos := 0;
end; {while}
end; {SendTo}

{-------------------------------------------------------------------------}
<p>| Case command index = Address |
| Pop Up a Dialogue Box asking the user to enter the new address |
| Convert the number entered from text to integer |
| Verify that the number is from 1 to 99 |
| If the node is valid |
| Call this node's ChangeAddress method to have the node / |</p>
<table>
<thead>
<tr>
<th>change its address to the one entered</th>
</tr>
</thead>
</table>

nc_Address: begin
  Str(PThisNode^.Address, InText);
  if Application^.ExecDialog(New(PInputDialog, |
    Init(@Self, 'New Address', 'Enter NEW ADDRESS |
  For This Node (1..99)', |
    InText, SizeOf(InText))) = id_OK then
  begin
    Val(InText, ReturnVal, ErrorPos);
    if (ReturnVal < 1) or (ReturnVal > 99) then
      ErrorPos := 1;
    if ErrorPos = 0 then
      PThisNode^.ChangeAddress(ReturnVal);
  end; {if}
else
  ErrorPos := 0;
end; {nc_Address}

{-------------------------------------------------------------------------}
<p>| While the number entered is not valid and user did not / |
| choose to cancel out of the dialogue box |
| Pop Up a Dialogue Box asking user to enter a new address |
| Convert the number entered from text to integer |
| Verify that the number is from 1 to 99 |
| If the node is valid |
| Call this node's ChangeAddress method to make the node/ |</p>
<table>
<thead>
<tr>
<th>change its address to the one entered</th>
</tr>
</thead>
</table>

EndWhile
while ErrorPos <> 0 do begin
   if Application^.ExecDialog(New(PInputDialog,
      Init(@Self, 'New Address', 'NEW ADDRESS Must Be From 1 To 99',
      InText, SizeOf(InText)))) = id_Ok then begin
      Val(InText, ReturnVal, ErrorPos);
      if (ReturnVal < 1) or (ReturnVal > 99) then
         ErrorPos := 1;
      if ErrorPos = 0 then
         PThisNode^.ChangeAddress(ReturnVal);
      end; (if)
   else
      ErrorPos := 0;
   end; (while)
end; (Address)

end; (case)
end; (if Found)
end; (if Msg.WParam)
end; [TMyWindow.WMCommand]

{
************************************************************************************************************
* Procedure TMyWindow.IDSpeedScroll
* * Responds to a change in the simulation speed scrollbar.
* * Gets the new value from the scrollbar and updates the timer delay
* * and then updates the speed display.
************************************************************************************************************
}
procedure TMyWindow.IDSpeedScroll(var Msg: TMessage); begin
   {---------------------------------------------------------------
   | Set the current speed value to the Speed scroll bar position
   | If the speed value is greater than zero
   | Set Windows timer delay to a constant divided by the speed value
   | EndIf
   | Update Speed static display with the new value of the speed value
   ---------------------------------------------------------------}
SpeedValue := PSpeedScroll^.GetPosition;
if SpeedValue > 0 then
   SetTimer(HWindow, 1, C_MaxTimerDelay div (SpeedValue), nil);
WYSPrintf(SpeedStr, '%d%%', SpeedValue);
TheDC := GetDC(HWindow);
PaintBrush := GetStockObject(LtGray_Brush);
   .
SelectObject(TheDC, PaintPen);
SelectObject(TheDC, PaintBrush);
Rectangle(TheDC, C_ScrollPos-47, 14, C_ScrollPos+1, 31);
SetBkMode(TheDC, Transparent);
TextOut(TheDC, C_ScrollPos-47, 14, SpeedStr, StrLen(SpeedStr));
SetBkMode(TheDC, Opaque);
ReleaseDC(HWindow, TheDC);
end; {IDSSpeedScroll}

{
***********************************************************************
* Procedure TMyWindow.IDLErrorScroll                      *
*  Responds to a change in the transmission error scrollbar        *
*  Gets the new value from the scrollbar and updates the error value* 
*  and then updates the error display.                           *
***********************************************************************
}
procedure TMyWindow.IDLErrorScroll(var Msg: TMessage);
var
  i: array[0..2] of integer;
  ErrorValue: integer;
  RErrorValue: real;

begin
  ErrorValue := FErrorScroll^.GetPosition;
  RErrorValue := 100 - (ln(100 - ErrorValue + 1) * 21.66790653);
  i[0] := Trunc(RErrorValue);
  i[1] := Trunc(100 * (RErrorValue - i[0]));
  if (i[1] = 0) or (i[1] >9) then
    WVSPrintf(ErrStr,'%d.%d%%', i)
  else
    begin
      i[2] := i[1];
      i[1] := 0;
      WVSPrintf(ErrStr,'%d.%d%d%%', i);
    end; {else}
  TheDC := GetDC(HWindow);
  PaintBrush := GetStockObject(LtGray_Brush);
  SelectObject(TheDC, PaintPen);
  SelectObject(TheDC, PaintBrush);
Rectangle(TheDC, C_ErrScrollPos-52, 14, C_ErrScrollPos+1, 31);
SetBkMode(TheDC, Transparent);
TextOut(TheDC, C_ErrScrollPos-52, 14, ErrStr, StrLen(ErrStr));
SetBkMode(TheDC, Opaque);
ReleaseDC(HWND, TheDC);

PTTheBus^.UpdateError(RErrorValue);
end; {IDErrorScroll}

{
  *************************************************************************************************
  * Procedure TMyWindow.Pause
  *    Responds to the Pause button being pressed.
  *    Causes the simulation to pause.
  *  *************************************************************************************************
}
procedure TMyWindow.Pause;
begin

  {-----------------------------------------------------------------------
  | Set the Step button pushed flag to false
  | Set the Pause button pushed flag to true
  {-----------------------------------------------------------------------}
  StepChosen := false;
  PauseChosen := true;
end; {TMyWindow.Pause}

{
  *************************************************************************************************
  * Procedure TMyWindow.Step
  *    Responds to the Step button being pushed
  *    Causes the simulation to run one step then pause.
  *  *************************************************************************************************
}
procedure TMyWindow.Step;
var
  NullMsg: TMessage;
begin

  {-----------------------------------------------------------------------
  | Set the Step button pushed flag to true
  | Set the Pause button pushed flag to false
  {-----------------------------------------------------------------------}
  StepChosen := true;
  PauseChosen := false;
  WMTimer(NullMsg);
end; {TMyWindow.Step}
procedure TForm1.Go;
  begin
    {------------------}
    | Set the Pause button and Step button pushed flags to false |
    {------------------}
    PauseChosen := false;
    StepChosen := false;
  end; {TMyWindow.Go}

procedure TMyWindow.Paint(PaintDC: HDC; var PaintInfo: TPaintStruct);

  procedure DrawPaintNode(PNode: PTreeNode); far;
  begin
    PNode^.Draw(PaintDC);
  end; {DrawPaintNode}

begin
  {------------------}
  | Select a light gray brush with a clear pen into display context |
  | Draw the light gray rectangle for the control bar background |
  | Draw a dark gray shadow under the light gray rectangle |
  | Set background mode to use current background color of control bar |
  | Write the Speed and Error text and values |
  | Reset the background mode to opaque |
  {------------------}
  PaintBrush := GetStockObject(LtGray_Brush);
  SelectObject(PaintDC, PaintBrush);
  SelectObject(PaintDC, PaintPen);
  Rectangle(PaintDC, 0, 1, 700, 32);
  PaintBrush := GetStockObject(Gray_Brush);
  SelectObject(PaintDC, PaintBrush);
Rectangle(PaintDC, 0, 31, 700, 33);
SetBkMode(PaintDC, Transparent);
TextOut(PaintDC, C_ScrollPos-50, 0, 'Speed:', 6);
TextOut(PaintDC, C_ErrScrollPos-57, 0, 'Bus Err:', 8);
TextOut(PaintDC, C_ScrollPos-47, 14, SpeedStr, StrLen(SpeedStr));
TextOut(PaintDC, C_ErrScrollPos-52, 14, ErrStr, StrLen(ErrStr));
SetBkMode(PaintDC, Opaque);
{

 Call the Draw method for each node in the node collection
 Call the Bus's Draw method

}

PNodeCollection^.ForEach(@DrawPaintNode);
PTheBus^.Draw(PaintDC);
end; {TMyWindow.Paint}

{

************************************************************************

* Procedure TMyWindow.CMNoToken
* Removes the token from the network.
************************************************************************

}

procedure TMyWindow.CMNoToken(Msg: TMessage);
 procedure KillToken(PNode: PTNode); far;
 begin
 PNode^.KillToken;
 end;

begin
{

 Call the KillToken method for each node in the node collection
 Call the Bus's KillToken method

}
PNodeCollection^.ForEach(@KillToken);
PTheBus^.KillToken;
end; {TMyWindow.CMNoToken}

end. {unit GGUI}
unit GSImObj;
{******************************************************************************

Unit GSImObj
This unit contains the definition for a Sim object. A Sim object is any object that is displayed as part of the token bus simulator. The node, bus, and link objects are all derived from a Sim object. A Sim object supplies basic pens, line type, sizes, colors, and locations of the objects.
******************************************************************************}

interface

uses WObjects, {ObjectWindows Window object definitions}
   WinTypes, {Windows type definitions}
   WinProcs; {Windows procedure definitions}

{******************************************************************************/
  \ TSimObject Object Description  //
  /******************************************************************************}


  type
      PFSimObject = ^TSimObject;
      TSimObject = object(TObject)
          ThePen,
          object
              ThePen2:
              HPen;
          {The normal drawing pen for this}
          {The alternate pen for this object}
          {This object's parent window handle}
          HWindow:
          HWnd;
          {Style of line for this object}
          LineType,
          {X and Y coordinates of this object}
          X, Y,
          {Width and Height of this object}
          Width, Height,
          {PenSize for this object}
          PenSize,
          {Address of this object}
          Address:
          integer;
          {The color of the object's frame}
          FrameColor:
          longint;

      constructor Init(pParent: PWindow; pX, pY, pWidth, pHeight, pPenSize: integer; pColor: longint);
      destructor Done; virtual;
      procedure TimerTick(DC: HDC); virtual;
      procedure Draw(DC: HDC); virtual;
      end; {TSimObject}

implementation
**TSimObject's Method Implementations**

* Constructor TSimObject.Init  *
  * Creates and initializes a SimObject.  *
  
  constructor TSimObject.Init(pParent: PWindow; pX, pY, pWidth, pHeight,  
  pPenSize: integer; pColor:  
  longint);  
  begin  
  {  
  \------------------------------  
  \ Call the parent object's Init method  
  \ Set the objects variables to the input parameters  
  \ Set the pen to one of the object's line type, pen size, framecolor  
  \ Set the alternate pen to null (clear pen)  
  \ Set the parent window to the calling object's parent window handle  
  \------------------------------  
  TObject.Init;  
  X := pX;   \ Y := pY;  
  Width := pWidth;   \ Height := pHeight;  
  LineType := ps_Solid;  
  PenSize := pPenSize;  
  FrameColor := pColor;  
  ThePen := CreatePen(LineType, PenSize, FrameColor);  
  ThePen2 := GetStockObject(Nul_pen);  
  HWindow := pParent^.HWindow;  
  end;  
  {TSimObject.Init}
  
  }  
  
  \------------------------------  
  \ Destructor TSimObject.Done  
  \ Destroys a SimObject.  
  \------------------------------  
  destructor TSimObject.Done;  
  begin  
  DeleteObject(ThePen);  
  TObject.Done;  
  end;  
  {TSimObject.Done}
  
  }  
  
  \------------------------------  
  \ Procedure TSimObject.Draw  
  \ Draws the Object on the screen.  
  \------------------------------  

procedure TSimObject.Draw(DC: HDC);
begin
  // Select the normal pen into the display context
  // Draw a rectangle of this object's size at its coordinates
  SelectObject(DC, ThePen);
  Rectangle(DC, X, Y, X+Width, Y+Height);
end; {TSimObject.Draw}

{
  Procedure TSimObject.TimerTick
  * Responds to a tick of the timer clock.
  * Should be overridden by the inheriting object.

procedure TSimObject.TimerTick(DC: HDC);
begin
end; {TSimObject.TimerTick}

end. {GSimObj}
unit GBusObj;

{*******************************************************************************

Unit GBusObj
This unit contains the object definitions for the bus and the link
objects. The link object methods are called by the bus object
making them 'part of' the bus. The purpose of the bus is to get
all of the nodes' transmissions and place them in every other
node's input buffer while keeping track of any bus contention
and data transmission error.
*******************************************************************************}

interface

uses
GGlobal, {Global data definitions}
GSimObj, {Object definition for a TSimObject object}
GNodeObj, {Object definition for a Node object}
WObjects, {Object Windows Window object definitions}
WinProcs, {Windows procedure definitions}
WinTypes; {Windows type definitions}

 type
PTBus = ^ TBus;

 TBus = object(TSimObject)
  TheFont: HFont; {The font used by the bus display}
  RErrorPct: real; {The % chance of error for bus data}
  BusBuffer: T_Buffer; {The buffer for data on the bus}
  PLinkCollection: PSortedAddressCollection;

 constructor Init(pParent: PWindow);
 destructor Done; virtual;
 procedure TimerTick(DC: HDC); virtual;
 procedure Draw(DC: HDC); virtual;
 procedure KillToken; virtual;
 procedure UpdateError(pNewErrorPct: real); virtual;
 procedure UpdateBusState; virtual;
 procedure UpdateBusDisplay(DC: HDC); virtual;
 procedure ClearNodeBuffers; virtual;
 end; (TBus)

{*******************************************************************************

\\ TLink Object Description ///
*******************************************************************************}
type
  PTLink = ^ TLink;
  TLink = object(TSimObject)
    Direction: T_Direction;  // The direction the link is pointing
    Side: T_Side;            // Which side of the bus the link is on
    LinkDidSend, LinkDidReceive: boolean;  // The link was sending last cycle, received last cycle

constructor Init(pParent: PWindow; px, py, pAddress: integer;
                 pSide: T_Side);
  procedure Draw(DC: HDC); virtual;
end; {TLink}

implementation

{**************************************************
 TLink's Method Implementations  ************
**************************************************}
{
**********************************************************************
* Constructor TLink.  *
* Creates and initializes a Link Object.  *
**********************************************************************
}constructor TLink.Init(pParent: PWindow; px, py, pAddress: integer;
                        pSide: T_Side);
begin
{ *******************************************************
  Call the parent object's Init method
  Set the alternate pen to a white pen (for blanking out old links)
  Initialize side and address variables from the input parameters
  Set the direction and previous send variables for an idle bus
}TSimObject.Init(pParent, px, py, LinkWidth, LinkHeight, 2,
RGB(0,0,0));
  Side := pSide;
  ThePen2 := GetStockObject(White_Pen);
  Address := pAddress;
  Direction := Idle;
  LinkDidSend := false;
  LinkDidReceive := false;
end; {TLink.Init}
procedure TLink.Draw(var UpDown: (IdleAbove, IdleBelow, Up, Down); Points: array[0..3] of TPoint;

begin
{
  Select the white pen into the display context
  Draw a white rectangle over the old link
  Select the normal black pen into the display context
  Determine if link should point up/down depending on which side/
    of the bus it is on (above or below) and whether or not the link is transmitting, receiving, or idle
{
SelectObject(DC, ThePen2);
Rectangle(DC, X, Y, X+Width, Y+Height);
SelectObject(DC, ThePen);
case Direction of
  Idle  : if Side = AboveBus then
    UpDown := IdleAbove
  else
    UpDown := IdleBelow;
  Receive: if Side = AboveBus then
    UpDown := Up
  else
    UpDown := Down;
  Transmit: if Side = AboveBus then
    UpDown := Down
  else
    UpDown := Up;
end;  {case}
{
  Case: Draw the link according to its up/down orientation
  IdleAbove links and IdleBelow links are both parallel lines,
    one is just drawn a little higher than the other, '/'.
    Down links are drawn as down pointing arrow heads like '/'.
    Up links are drawn as up pointing arrow heads like '/'.
EndCase
{
case UpDown of
  IdleAbove: begin

Points[0].X := X;
Points[0].Y := Y;
Points[1].X := X;
Points[1].Y := Y + Height + 1;
Points[2].X := X + Width;
Points[2].Y := Y + Height + 1;
Points[3].X := X + Width;
Points[3].Y := Y;
PolyLine(DC, Points, 4);
end; {IdleAbove}

IdleBelow: begin
Points[0].X := X;
Points[0].Y := Y + Height - 1;
Points[1].X := X;
Points[1].Y := Y;
Points[2].X := X + Width;
Points[2].Y := Y;
Points[3].X := X + Width;
Points[3].Y := Y + Height - 1;
PolyLine(DC, Points, 4);
end; {IdleBelow}

Down : begin
Points[0].X := X;
Points[0].Y := Y;
Points[1].X := X + (Width div 2);
Points[1].Y := Y + Height;
Points[2].X := X + Width;
Points[2].Y := Y;
PolyLine(DC, Points, 3);
end; {Down}

Up : begin
Points[0].X := X;
Points[0].Y := Y + Height;
Points[1].X := X + (Width div 2);
Points[1].Y := Y;
Points[2].X := X + Width;
Points[2].Y := Y + Height;
PolyLine(DC, Points, 3);
end;
end; {case}
end; {TLink.Draw}
constructor TBus.Init(pParent: PWindow);
begin
{
  // Call the parent object's Init method
  // Create the link collection and allow duplicate address entries
  // Initialize each of the links and insert into the link collection
  TSimObject.Init(pParent, BusX, BusY, BusWidth, BusHeight, 3,
  RGB(0,0,0));
  PLinkCollection := New(TSortedAddressCollection, Init(8,0));
  PLinkCollection^.Duplicates := true;
  PLinkCollection^.Insert(New(PTLink, Init(pParent, LinkMargin, 200,
  N1_Address, AboveBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent,
  LinkMargin+HSpacing, 200, N2_Address, AboveBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent,
  LinkMargin+2*HSpacing, 200, N3_Address, AboveBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent,
  LinkMargin+3*HSpacing, 200, N4_Address, AboveBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent, LinkMargin, 255,
  N5_Address, BelowBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent,
  LinkMargin+HSpacing, 255, N6_Address, BelowBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent,
  LinkMargin+2*HSpacing, 255, N7_Address, BelowBus)));
  PLinkCollection^.Insert(New(PTLink, Init(pParent,
  LinkMargin+3*HSpacing, 255, N8_Address, BelowBus)));
{
  // Clear the values of the bus buffer
  // Create the large font used in the bus display
  // Initialize the error percent to zero
}{
with BusBuffer do
begin
  FromAddress := 0;
  ToAddress := 0;
  FrameType := ft_Null;
  FrameArg1 := 0;
  FrameArg2 := 0;
  Contention := false;
  Error := false;
  IsTransmitting := false;
end; {with}
TheFont := CreateFont(25, 16, 0, 0, fw_ExtraLight, 0, 0, 0,
  OEM CharSet, Out_CharSet, Precis, Clip_Default_Precis,
Draft_Quality, Variable_Pitch or ff_Modern,
'GreggFont');

{TheFont := CreateFont(60, 180, 0, 0, fw_heavy, 0, 0, 0,
OEM_CharSet, Out_String_Precis, Clip_Default_Precis,
Proof_Quality, Fixed_Pitch, 'GreggFont');

RErrorPct := 0.0;
end; {TBus.Init}

{
****************************************************************************************
* Destructor TBus.Done
*    Deletes the bus and link objects
****************************************************************************************
}
destructor TBus.Done;
beg
   TSimObject.Done;
   Dispose(PLinkCollection,Done);
   DeleteObject(TheFont);
end; {TBus.Done}

{
****************************************************************************************
* Procedure TBus.UpdateBusDisplay
*    Updates the Bus display after a state update.
****************************************************************************************
}
procedure TBus.UpdateBusDisplay(DC: HDC);
procedure UpdateLinkDisplay(PLink: PTLink); far;
var
   Changed: boolean;
   PMatchingNode: PTNode;
begin
{
   __________________________________________________________
   *** The links and nodes are both in sorted collections, sorted by /
   *** address so nodes and links of the same address have the same
   *** index and can be cross referenced. This is done to find out if
   *** a given link's node is transmitting or not.
   __________________________________________________________

   Set the changed flag to false (link does not need to be updated)
   Find the node that corresponds to this link
   If the node is transmitting this time
   Set the link direction to transmit
   If this link did not send last cycle
      Set the changed flag to true
   EndIf
   Set the new values of the previous status flags to Send

   __________________________________________________________
Changed := false;

PMatchingNode :=
NodeCollection^.At(PLinkCollection^.IndexOf(PLink));
if PMatchingNode^.OutputBuffer.IsTransmitting then
  begin
    PLink^.Direction := Transmit;
    if not PLink^.LinkIdSend then
      Changed := true;
    PLink^.LinkIdSend := true;
    PLink^.LinkIdReceive := false;
  end {if}

{------------------------------------------------------------------}
Else (node is not transmitting)
If the To address on the bus is equal to this node’s address
  Set the link direction to receive
  If this link did not receive last cycle
    Set the changed flag to true
  EndIf
  Set the new values of the previous status flags to Receive
{------------------------------------------------------------------}
else
  if PMatchingNode^.Address = BusBuffer.ToAddress then
    begin
      PLink^.Direction := Receive;
      if not PLink^.LinkIdReceive then
        Changed := true;
      PLink^.LinkIdReceive := true;
      PLink^.LinkIdSend := false;
    end {else if}

{------------------------------------------------------------------}
Else (node is not transmitting or receiving)
  If link was transmitting or receiving last time
    Set the changed flag to true
  EndIf
  Set the link direction to idle
  Set the new values of the previous status flags to Idle
  EndIf (node is receiving)
  EndIf (node is transmitting)
{------------------------------------------------------------------}
else
  begin
    if PLink^.LinkIdSend or PLink^.LinkIdReceive then
      Changed := true;
    PLink^.Direction := Idle;
    PLink^.LinkIdSend := false;
    PLink^.LinkIdReceive := false;
  end {else}
{
If the changed flag was set to true (the display needs updating)
  Call the link's Draw method
EndIf

if Changed then
  PLink^.Draw(DC);
end; {UpdateLinkDisplay}

{------------------------}
begin

  {------------------------}
  *** Beginning of UpdateBusDisplay main procedure
  {------------------------}
  Select the bus's large font into the display context
  Write the current frame into the main bus
  Call UpdateLinkDisplay for each link to show current link status
  {------------------------}
  SelectObject(DC, TheFont);
  TextOut(DC, X+70, Y+10, BusMsg[BusBuffer.FrameType], 30);
  PLinkCollection^.ForEach(@UpdateLinkDisplay);
end; {TBus.UpdateBusDisplay}

{***********************
* Procedure TBus.Draw
*   Draws the Bus on the screen.
***********************
} procedure TBus.Draw(DC: HDC);

  procedure DrawLink(PLink: PTLink); far;
  begin
    PLink^.Draw(DC);
  end; {DrawLink}

  {------------------------}
begin
  {------------------------}
  Call the parent object Draw method to draw the main rectangle
  Call each link's Draw method
  Call UpdateBusDisplay to show the current bus status
  {------------------------}
  TSimObject.Draw(DC);
PLinkCollection^.ForEach(@DrawLink);
UpdateBusDisplay(DC);
end; {TBus.Draw}

{

******************************************************************************
* Procedure TBus.UpdateBusState  *
*  Gets transmissions from node output buffers and transfers*  
*  them to the nodes input buffers. This routine also   *
*  determines whether or not there is contention and error  *
******************************************************************************
}
procedure TBus.UpdateBusState;

{=============================-----------------------------------------------
 procedure ReceiveNodeTransmission 
  Transfers the transmitting node's output buffer to the bus. 
  If more than one node is transmitting, the contention flag 
  is set.
{=============================-----------------------------------------------

procedure ReceiveNodeTransmission(PNode: PTNode); far;
begin
{

if the node is transmitting

  If there is already a transmission on the bus
    Set the bus frame type to contention
    Set the bus contention flag to true
  Else (there is no contention)
    Copy the Addresses, frametype, and arguments from node buffer / 
    to the bus buffer
    Set the bus's transmit flag to true
  EndIf
EndIf (node is transmitting)


{-----------------------------------------------
with PNode^.OutputBuffer do
  if IsTransmitting then
    if BusBuffer.IsTransmitting or BusBuffer.Contention then
       begin
          BusBuffer.FrameType := ft_Contention;
          BusBuffer.Contention := true;
       end {if}
    else
       begin
          BusBuffer.FromAddress := FromAddress;
          BusBuffer.ToAddress := ToAddress;
          BusBuffer.FrameType := FrameType;
          BusBuffer.FrameArg1 := FrameArg1;
          BusBuffer.FrameArg2 := FrameArg2;
          BusBuffer.IsTransmitting := true;
      end {else}
end; {else}
end; {ReceiveNodeTransmission}

\begin{verbatim}
procedure FillInputBuffer
  Transfers the contents of the bus to all of the nodes' input buffers.
\end{verbatim}

\begin{verbatim}
procedure FillInputBuffer(PNode: PTNode); far;
begin
  PNode^.InputBuffer := BusBuffer;
end; {FillInputBuffer}
\end{verbatim}

\begin{verbatim}
begin {UpdateBusState}
{----------------------------------------------------------}
  *** Beginning of UpdateBusState main procedure
----------------------------------------------------------
| Clear the values in the bus buffer |
| Call ReceiveNodeTransmission for all nodes in the node collection / |
|   to transfer any transmissions to the bus buffer              |
| Calculate whether or not there is any error in a data transmission|
| Transfer the contents of bus buffer into each node's input buffer|
{----------------------------------------------------------}
with BusBuffer do
begin
  FromAddress := 0;
  ToAddress := 0;
  FrameType := ft_Null;
  FrameArg1 := 0;
  FrameArg2 := 0;
  Contention := false;
  Error := false;
  IsTransmitting := false;
end; {with}
PNodeCollection^.ForEach(@ReceiveNodeTransmission);

if RErrorPct > 100 * Random then
  BusBuffer.Error := true;

PNodeCollection^.ForEach(@FillInputBuffer);
end; {TBus.UpdateBusState}
\end{verbatim}

\begin{verbatim}
{*******************************************************************************
  * Procedure TBus.ClearNodeBuffers                                          *
  *   Clears the Output Buffers of all of the nodes.                         *
*******************************************************************************
\end{verbatim}
procedure TBus.ClearNodeBuffers;


Procedure ClearBuffer
Sets all of the values in an output buffer to null values

procedure ClearBuffer(PNode: PTreeNode); far;
begin
  with PNode^.OutputBuffer do
  begin
    FromAddress := 0;
    ToAddress := 0;
    FrameType := ft_Null;
    FrameArg1 := 0;
    FrameArg2 := 0;
    Contention := false;
    Error := false;
    IsTransmitting := false;
  end; {with}
end; {ClearBuffer}


begin
  {----------------------------------}

  begin
    {-----------------------------}
    | Clear the output buffer of each node in the node collection |
    {-----------------------------}
    PNodeCollection^.ForEach(@ClearBuffer);
  end; {TBus.TimerTick}


begin


begin


procedure TBus.TimerTick(DC: HDC);
begin
  {----------------------------------}
  | Call UpdateBusState to receive and transmit to/from all the nodes
  | Call UpdateBusDisplay to show the new bus contents and state
  | Call ClearNodeBuffers to clear the nodes' output buffers
  {----------------------------------}
  UpdateBusState;
  UpdateBusDisplay(DC);
  ClearNodeBuffers;
  {----------------------------------}


end; {TBus.TimerTick}

{
.Designer.
******************************************************************************
* Procedure TBus.KillToken  *
*   Removes the token from the bus if it exists   *
******************************************************************************
}
procedure TBus.KillToken;
begin
  {
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the buffer frame type to null</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
  }
  BusBuffer.FrameType := ft_Null;
end; {TBus.KillToken}

{
.Designer.
******************************************************************************
* Procedure TBus.UpdateError  *
*    Updates the percent transmission error on the bus.                     *
******************************************************************************
}
procedure TBus.UpdateError(pNewErrorPct: real);
begin
  {
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Set the error percentage to the value of the input parameter</td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
  }
  RErrorPct := pNewErrorPct;
end; {TBus.UpdateError}

end. {GBus}
unit GNodeObj;

{**************************************************************************}

Unit GNodeObj
This is the unit that describes a TNode Object.
The node is responsible for performing all token bus network
related maintenance and data transmission to/from the bus.

**************************************************************************}

interface
uses GGlobals, {Global constants, types, vars}
    WObjects, {ObjectWindows Objects}
    WinTypes, {windows type declarations}
    WinProc, {Windows procedure declarations}
    GSimObj, {SimObject object definition}
    Strings; {Turbo String functions}

{**************************************************************************}

\b TNode Object Description
\\
**************************************************************************

type
    PTNode = ^TNode;
    TNode = object(TSimObject)
        TokenBrush, {Brush’s token, active, and passive }
        ActiveBrush, { node header rectangles (contains }
        PassiveBrush: { the color of the node header ) }
        HBrush;
        NodeNumber: {Char node number for display } char;
        Status, {Current status of node }
        XStatus: {Status of node on last clock count }
        T_Status;
        Action, {Current action node is taking }
        XAction: {Action of node on last clock count }
        T_Action;
        Step, {Current step of the current action }
        XStep: {Step of node on last clock count }
        T_Step;
        XAddress, {Address of node on last clock count }
        XXAddress, {Address if node two counts ago }
        PrevAddress, {Address of previous node in ring }
        XPrevAddress, {PrevAddress on last clock count }
        XNextAddress, {NextAddress on last clock count }
        SendToAddress, {Address of node to send to }
        DeadBusCount, { # clock counts since bus was active }
        ContentionCount, {" " " " started resolving conten. }
        SolicitSuccCount, {" " " " soliciting a successor}
        PassingTokenCount, {" " " " passing the token }
        NextSolicitCount, { # token possessions since last }
        SendingCount, { # clock counts since started sending }
        AddingCount,
constructor Init(pParent: PWindown; pX, pY: integer; pAddress: integer; pNodeNumber: char);
destructor Done; virtual;
procedure UpdateNodeState(DC: HDC); virtual;
procedure UpdateNodeStatus; virtual;
procedure UpdateNodeDisplay(DC: HDC); virtual;
procedure TimerTick(DC: HDC); virtual;
procedure KillToken; virtual;
procedure Draw(DC: HDC); virtual;
procedure SetCommand(pCommand: T_NodeCommand); virtual;
procedure MakeAutoSend(pIsChecked: boolean); virtual;
procedure MakeSendTo(pToAddress: integer); virtual;
procedure ChangeAddress(pNewAddress: integer); virtual;
end; {TNode}

implementation

{TNode's Method Implementations}

constructor TNode.Init(pParent: PWiindow; px, py: integer; pAddress: integer; pNodeNumber: char);
begin

{---------------------------------------------}
| Call object's parent Init procedure
| Set remaining node attributes from input parameters
| Initialize special pens and brushes
| Initialize node status display vars.
{---------------------------------------------}
TSimObject.Init(pParent, px, py, NodeWidth, NodeHeight, 3,
C_NormalColor);
Address := pAddress;
XAddress := Address;
XXAddress := Address;
NodeNumber := pNodeNumber;
ThePen2 := CreatePen(ps_Solid, 1, FrameColor);
ActiveBrush := CreateSolidBrush(C_HeaderColor);
PassiveBrush := CreateSolidBrush(C_PassiveColor);
TokenBrush := CreateSolidBrush(C_TokenColor);
Command := nc_Inactive;
Status := st_Inactive;
Action := ac_Wait;
Step := sp_Wait;
PrevAddress := 0;
NextAddress := 0;
XStatus := st_Inactive;
XAction := ac_Wait;
XStep := sp_Wait;
YPrevAddress := 0;
YNextAddress := 0;
XCount := 0;
{
    Initialize all node status booleans to false
    isAddNode := false;
    isDeleteNode := false;
    isTokenOwner := false;
    isReadyToSend := false;
    isAutoSend := false;
    isSending := false;
    isResolvingContention := false;
    isResolvingContention := false;
    isSolicitContention := false;
    isInitializing := false;
    isSolicitSucc := false;
    isPassingToken := false;
    isWhoFollows := false;
    isInitSolicit := false;
    isAdding := false;
    isDeleting := false;
    isChangingAddress := false;
    isAddressNode := false;
    AlreadyAutoSent := false;
    {
        Initialize all counters to zero
        Initialize last frame sent to null
    }
    DeadBusCount := 0;
    ContentionCount := 0;
    SolicitSuccCount := 0;
    PassingTokenCount := 0;
    NextSolicitCount := 0;
    AddingCount := 0;
    SendingCount := 0;
    NumberSent := 0;
    NumSolicitContentions := 0;
    AddressBits := 0;
    WhenToSend := 0;
    DidSend := ft_Null;
    {
        Initialize node input and output buffers to null values
    }
    with InputBuffer do
    begin
        FromAddress := 0;
        ToAddress := 0;
        FrameType := ft_Null;
        FrameArg1 := 0;
        FrameArg2 := 0;
IsTransmitting := false;
Contension := false;
Error := false;
end; {with}
OutputBuffer := InputBuffer;
end; {TNode.Init}

{

*******************************************************************************
* destructor TNode.Done
* * Destroys a node object.
*******************************************************************************
}

{ TNode.Done; begin
{
| Delete all pen and brush objects allocated by this node
| Call this object's parent Done procedure
|------------------------------------------------------------------------------|
DeleteObject(ThePen2);
DeleteObject(ActiveBrush);
DeleteObject(PassiveBrush);
DeleteObject(TokenBrush);
TSimObject.Done;
end; {TNode.Done}

{

*******************************************************************************
* Procedure TNode.UpdateNodeDisplay
* * Updates the node's display to show new status.
* * This routine only changes those fields and areas of the display*
* that need to be changed to show the nodes current status. If *
* this extra care is not taken, the display update becomes much *
* much slower and slows down the speed of the simulation *
* dramatically. Windows GDI functions are used for the display. *
*******************************************************************************
}
procedure TNode.UpdateNodeDisplay(DC: HDC);
var
  WhitePen,
  BlackPen: HPen;
  StockBrush: HBrush;
  Temp: array[0..2] of char;
  ChangeCount: boolean;
Counter: integer;

begin
{
    
    If the node status has changed or the caller is Draw
    If the current status is Active, Token or Passive
        Select a null pen into the display context
        Select the brush that contains the proper color for current/
            node status
        Set text color to white
        Draw a rectangle for the node's header
    --------------------------

    if (Status <> XStatus) or (CalledFromDraw) then
        if (Status = st_Active) or (Status = st_Token) or
            (Status = st Passive) then
            begin
                WhitePen := GetStockObject(NULL_Pen);
                SelectObject(DC, WhitePen);
                if Status = st Passive then
                    SelectObject(DC, PassiveBrush)
                else
                    if Status = st_Token then
                        SelectObject(DC, TokenBrush)
                    else
                        SelectObject(DC, ActiveBrush);
                SetTextColor(DC, $00FFFF);
                Rectangle(DC, X + 3, Y + 3, X + 128, Y + 20);
            
            ---------------------------

            Set the background mode to transparent
            Write 'Node x' in white in the header
            Restore the background mode and text color (black)
    --------------------------

    SetBkMode(DC, Transparent);
    TextOut(DC, X+40, Y+3, 'Node ', 5);
    Temp[0] := NodeNumber;
    TextOut(DC, X+80, Y+3, Temp, 1);
    SetBkMode(DC, Opaque);
    SetTextColor(DC, $00000000);
    end if
    {
        
        Else (not Passive, Active, or Token)
            Select a null pen and white brush into current display context
            Set the text color to black
            Draw a white rectangle in the header
            Write 'Node x' in the header
        EndIf
    EndIf (Status changed)
}
else
begin
    WhitePen := GetStockObject(Null_Pen);
    StockBrush := GetStockObject(White_Brush);
    SelectObject(DC, WhitePen);
    SelectObject(DC, StockBrush);
    SetTextColor(DC, $00000000);
    Rectangle(DC, X + 2, Y + 2, X + 129, Y + 20);
    TextOut(DC, X+40, Y+3, 'Node ', 5);
    Temp[0] := NodeNumber;
    TextOut(DC, X+80, Y+3, Temp, 1);
end; {else}

{-------------------------------------------------------------------;

    If node's current status is Down
    Set the frame color to constant DownColor
    Else
    Set the frame color to constant NormalColor
    EndIf
{-------------------------------------------------------------------;}

if Status = st_Down then
    FrameColor := C_DownColor
else
    FrameColor := C_NormalColor;

{-------------------------------------------------------------------;

    If Draw is the caller or
    / node status has changed and the new or old status is Down
    Select a null brush into the display context
    Select a solid pen of width 3 of this node's frame color
    Draw a rectangle for the node's inner border
    If the status is currently Down
    Select a Dash-Dot-Dot line style for the outer border
    Else
    Select a solid black line for the outer border of the node
    EndIf
    Draw a rectangle for the node's outer border
    EndIf
{-------------------------------------------------------------------;}

if (CalledFromDraw) or ((Status <> XStatus) and
((Status = st_Down) or (XStatus = st_Down)) ) then

begin
    StockBrush := GetStockObject(Null_Brush);
    DeleteObject(ThePen);
    ThePen := CreatePen(ps_Solid, 3, FrameColor);
    SelectObject(DC, ThePen);
    SelectObject(DC, StockBrush);
    Rectangle(DC, X, Y, X + Width, Y + Height - 1);
    if Status = st_Down then
        BlackPen := CreatePen(ps_DashDotDot, 1, $00000000)
else
  BlackPen := GetStockObject(Black_Pen);
  SelectObject(DC, BlackPen);
  Rectangle(DC, X - 2, Y - 2, X + Width + 2, Y + Height + 1);
  If Status = st_Down then
    DeleteObject(BlackPen);
  end; {if}

{--------------------------------------------------------------------------}
Select a white brush into the current display context
Select a white pen into the current display context
If Draw is the caller or the node's address in the ring has changed
  Display the new value of the node's address
EndIf
If Draw is the caller or the next address in the ring has changed
  Display the new value of the next address
EndIf
If Draw is the caller or previous address in the ring has changed
  Display the new value of the previous address
EndIf
{--------------------------------------------------------------------------}
StockBrush := GetStockObject(White_Brush);
WhitePen := GetStockObject(White_Pen);
SelectObject(DC, WhitePen);
SelectObject(DC, StockBrush);

  if (CalledFromDraw) or (Address <> XAddress) or
     (Address <> XXAddress) then begin
    Str(Address, Temp);
    TextOut(DC, X+90, Y+22, ' ', 5);
    TextOut(DC, X+90, Y+22, Temp, StrLen(Temp));
    XXAddress := XAddress;
    XAddress := Address;
  end; {if}

  if (CalledFromDraw) or (NextAddress <> XNextAddress) then begin
    Str(NextAddress, Temp);
    TextOut(DC, X+90, Y+41, ' ', 5);
    TextOut(DC, X+90, Y+41, Temp, StrLen(Temp));
    XNextAddress := NextAddress;
  end; {if}

  if (CalledFromDraw) or (PrevAddress <> XPrevAddress) then begin
    Str(PrevAddress, Temp);
    TextOut(DC, X+90, Y+60, ' ', 5);
    TextOut(DC, X+90, Y+60, Temp, StrLen(Temp));
    XPrevAddress := PrevAddress;
  end; {if}
If `Draw` is the caller or the node's current status has changed
   Display the new value of the current status
EndIf
If `Draw` is the caller or the node's current action has changed
   Display the new value of the current action
EndIf
If `Draw` is the caller or the node's current step has changed
   Display the new value of the current step
EndIf

if (CalledFromDraw) or (Status <> XStatus) then
begin
   Rectangle(DC, X+53, Y+80, X+127, Y+95);
   TextOut(DC, X+53, Y+80, NodeStatusMsg[Status], 10);
   XStatus := Status;
end; {if}

if (CalledFromDraw) or (Action <> XAction) then
begin
   Rectangle(DC, X+52, Y+98, X+127, Y+113);
   TextOut(DC, X+52, Y+98, NodeActionMsg[Action], 10);
   XAction := Action;
end; {if}

if (CalledFromDraw) or (Step <> XStep) then
begin
   Rectangle(DC, X+40, Y+116, X+127, Y+131);
   TextOut(DC, X+40, Y+116, NodeStepMsg[Step], 12);
   XStep := Step;
end; {if}

If we are or were resolving a contention problem or `Draw` is caller
   Display Address bits being used and the send delay being used /
   to resolve the contention
EndIf

if IsResolvingContention or XIsResolvingContention or
   CalledFromDraw then
begin
   Str(AddressBits, Temp);
   TextOut(DC, X+27, Y+135, Temp, StrLen(Temp));
   Str(WhenToSend, Temp);
   TextOut(DC, X+66, Y+135, Temp, StrLen(Temp));
end; {if}

Display the value of active counter. If none are active, write 0.
Update the values of the previous state variables to equal the current state variables

Counter := 0;
if IsResolvingContention then
  Counter := ContentionCount;
if IsSolicitSucc then
  Counter := SolicitSuccCount;
if IsPassingToken then
  Counter := PassingTokenCount;
if IsSending then
  Counter := SendingCount;
if (Counter <> XCount) or (CalledFromDraw) then
  begin
    Str(Counter, Temp);
    Rectangle(DC, X+110, Y+135, X+127, Y+150);
    TextOut(DC, X+110, Y+135, Temp, StrLen(Temp));
  end; {if}

XCount := Counter;
XIsResolvingContention := IsResolvingContention;
end; {TNode.UpdateNodeDisplay}

************

* Procedure TNode.Draw
* Draws the Node on the screen.

************

procedure TNode.Draw(DC: HDC);
var
  Temp: array[0..11] of char;
  DividerPen: HPen;
begin
  {----
  Select the pen of width 1, light gray into the display context
  Draw a line of width 2 under the header
  Write the Address, Prev Address, and Next Address text
  ----} SelectObject(DC, ThePen2);

  MoveTo(DC, X+2, Y+20);
  LineTo(DC, X+Width-2, Y+20);
  MoveTo(DC, X, Y+19);
  LineTo(DC, X+Width, Y+19);
TextOut(DC, X+4, Y+22, 'Address: ', 9);
TextOut(DC, X+4, Y+41, 'Next Addr: ', 11);
TextOut(DC, X+4, Y+60, 'Prev Addr: ', 11);

{-------------
| Select a solid dark gray pen, width 1, into display context
| Draw a line under the Address section of the node display
| Write the Status, Action, and Step text
| Draw a line under the status text
| Write the contention status and counter text
| Set CalledFromDraw to let UpdateDisplayNode know Draw is caller
| Call UpdateDisplayNode
| Reset CalledFromDraw to false

DividerPen := CreatePen(ps_Solid, 1, $00404040);
SelectObject(DC, DividerPen);
MoveTo(DC, X, Y+77);
LineTo(DC, X+Width, Y+77);
TextOut(DC, X+4, Y+80, 'Status: ', 8);
TextOut(DC, X+4, Y+98, 'Action: ', 8);
TextOut(DC, X+4, Y+116, 'Step: ', 6);
MoveTo(DC, X+2, Y+133);
LineTo(DC, X+Width-2, Y+133);
TextOut(DC, X+4, Y+135, 'Cs: Cw: Cnt: ', 19);
DeleteObject(DividerPen);

CalledFromDraw := true;
UpdateNodeDisplay(DC);
CalledFromDraw := false;
end; [TNode.DrawNode]

{
********************************************************************
* Procedure TNode.UpdateNodeStatus                                *
* Perform status update for network node.                        *
* Updates node status and status flags in response to the menu    *
* command. It also assigns the node default status values         *
* when there are no other special activities node is performing.  *
********************************************************************
}
procedure TNode.UpdateNodeStatus;

{===============================================================
 Procedure UpdateActiveStatus
 Performs status update for an Active network node.


procedure UpdateActiveStatus;
begin
{

*** Update current display variables. These may have changed due
*** to node acknowledging a data transmission, trying to add to the
*** ring, etc.

If node needs to add into ring (it is not a member now)
  Set current Action to Add
    If this node is not in the process of adding into the ring
      Set the previous and next addresses to zero
      Set the current Step to Listen
    Else
      Set the current Step to Listen
    EndIf
  Else
    (this node is already a member of the ring)
    If we are not currently passing the token, soliciting a successor
      sending data, or resolving contention
      Set the current Action and Step to Listen
    EndIf
  EndIf
EndIf
If node needs to delete or is currently deleting from ring
  Set the current Action to delete
EndIf

if IsAddNode then
begin
  Action := ac_Add;
  if not IsAdding then
    begin
      PrevAddress := 0;
      NextAddress := 0;
      Step := sp_Listen;
    end {if}
  else
    Step := sp_Listen;
  end {if}
else {Is a member of the ring}
  if (not IsPassingToken) and (not IsSolicitSucc) and
    (not IsSending) and (not IsResolvingContention) then
    begin
      Action := ac_Listen;
      Step := sp_Listen;
    end {if}

If IsDeleteNode or IsDeleting then
  Action := ac_Delete;
Case the current menu command of
    Active   : Set the needs to delete flag to false
       Set the is currently deleting flag to false
    Inactive : If node needs to add to ring or its next and prev /
               addresses are the same
       Set status, action, and step for a passive node
       Set status flags so node is not trying to add,/
       is not trying to delete, is not now deleting,/
       is not passing token, is not token owner.
EndIf
    Set flag saying node needs to delete
    Set flag that node is not currently adding

Case Command of
    nc_Active : begin
        IsDeleteNode := false;
        IsDeleting := false;
    end; {nc_Active}
    nc_Inactive: begin
        if (IsAddNode) or
           (NextAddress = PrevAddress) then
            begin
                Status := st_Inactive;
                Action := ac_Listen;
                Step := sp_Listen;
                IsAddNode := false;
                IsDeleteNode := false;
                IsDeleting := false;
                IsPassingToken := false;
                IsTokenOwner := false;
            end; {if}
            IsDeleteNode := true;
            IsAdding := false;
        end; {Inactive}

Case command = Passive:
    If this node needs to add to ring or next and prev /
       addresses are the same
    Set the status, action, and step for passive node
    Set status flags so node is not trying to add,/
    is not trying to delete, is not currently deleting,/
    is not passing token, is not token owner.
EndIf
    Set flag that node needs to delete
    Set flag that node is not currently adding

    nc_Passive: begin
                  .
if (IsAddNode) or (NextAddress = PrevAddress) then begin
    Status := st_Passive;
    Action := ac_Listen;
    Step := sp_Listen;
    IsAddNode := false;
    IsDeleteNode := false;
    IsDeleting := false;
    IsPassingToken := false;
    IsTokenOwner := false;
end; {if}
IsDeleteNode := true;
IsAdding := false;
end; {Passive}

Case command = Crash:
Set Status, Action, and Step for a crashed node
Set the previous and next addresses to zero
Set status flags so node is not trying to or now/
    adding into the ring, is not trying to or/
    currently deleting from ring, is not sending /
    data, is not the token owner, is not passing /
    the token
Clear the output buffer transmit flag
EndCase

nc_Crash: begin
    Status := st_Down;
    Action := ac_Crash;
    Step := sp_Wait;
    PrevAddress := 0;
    NextAddress := 0;
    IsAddNode := false;
    IsAdding := false;
    IsDeleteNode := false;
    IsDeleting := false;
    IsSending := false;
    IsSolicitSucc := false;
    IsPassingToken := false;
    IsTokenOwner := false;
    OutputBuffer.IsTransmitting := false;
end; {case}
end; {TNode.UpdateActiveStatus}

Procedure UpdateInactiveStatus
Performs status update for an inactive network node.

procedure UpdateInactiveStatus;
begin
{
Case the current menu command of
  Active : Reset the dead bus count
    Set the status, action, and step for a new Active node
    Set status flags so node is trying to add, / is not trying to delete, is not currently deleting
  Passive Set Status, Action, and Step for a Passive node
  Crash Set Status, Action, and Step for a Crashed node
    Clear the output buffer transmission flag
EndCase
{
\case Command of
  nc_Active : begin
    DeadBusCount := 0;
    Status := st_Active;
    Action := ac_Add;
    Step := sp_Listen;
    IsAddNode := true;
    IsAdding := false;
    IsDeleting := false;
    IsDeleteNode := false;
    end; {Active}
  nc_Passive: begin
    Status := st_Passive;
    Action := ac_Listen;
    Step := sp_Listen;
    end; {Passive}
  nc_Crash : begin
    Status := st_Down;
    Action := ac_Crash;
    Step := sp_Wait;
    OutputBuffer.IsTransmitting := false;
    end; {Crash}
\end{case}
end; {TNode.UpdateInactiveStatus}

Procedure UpdatePassiveStatus
Performs status update for an Passive network node.

procedure UpdatePassiveStatus;
begin
{

Set current Action and Step to Listen
Case the current menu command of
  Active  :  Reset the dead bus count
    Set the status, action, and step for a new Active node
    Set status flags so node is trying to add, is not trying to delete, is not currently deleting
  Inactive:  Set Status, Action, and Step for an Inactive node
  Crash   :  Set Status, Action, and Step for a Crashed node
    Clear the output buffer transmission flag
EndCase

Action := ac_Listen;
Step := sp_Listen;
case Command of
  nc_Active: begin
    DeadBusCount := 0;
    Status := st_Active;
    Action := ac_Add;
    Step := sp_Listen;
    IsAddNode := true;
    IsAdding := false;
    IsDeleting := false;
    IsDeleteNode := false;
  end;  {Active}

  nc_Inactive: begin
    Status := st_Inactive;
    Action := ac_Wait;
    Step := sp_Wait;
  end;  {Inactive}

  nc_Crash: begin
    Status := st_Down;
    Action := ac_Crash;
    Step := sp_Wait;
    OutputBuffer.IsTransmitting := false;
  end;  {Crash}
end;  {case}
end;  {TNode.UpdatePassiveStatus}

{=================================}
Procedure UpdateDownStatus
  Performs status update for a Crashed network node.
{=================================}
procedure UpdateDownStatus;
begin
  Set current Action and Step to Listen
Case the current menu command of
  Active : Reset the dead bus count
          Set the status, action, and step for a new Active node
          Set status flags so node is trying to add, / is not trying to delete, is not currently deleting
  Inactive Set Status, Action, and Step for an Inactive node
  Passive  Set Status, Action, and Step for a Passive node
EndCase

---
case Command of
  nc_Active : begin
    DeadBusCount := 0;
    Status := st_Active;
    Action := ac_Add;
    Step := sp_Listen;
    IsAddNode := true;
    IsAdding := false;
    IsDeleting := false;
    IsDeleteNode := false;
  end; \{Active\}
  nc_Inactive: begin
    Status := st_Inactive;
    Action := ac_Wait;
    Step := sp_Wait;
  end; \{Inactive\}
  nc_Passive : begin
    Status := st_Passive;
    Action := ac_Listen;
    Step := sp_Listen;
  end; \{Passive\}
end; \{case\}
end; \{TNode.UpdateDownStatus\}

---
begin \{TNode.UpdateStatus\}
{
  *** Beginning of main UpdateStatus procedure
  Call the status update procedure appropriate for the current status
}
case Status of
  st_Active:  UpdateActiveStatus;
  st_Token :  UpdateActiveStatus;
  st_Inactive: UpdateInactiveStatus;
  st_Passive:  UpdatePassiveStatus;
  st_Down:    UpdateDownStatus;
end; \{case\}
end; {TNode.UpdateNodeStatus}

{
**************************************************************************************************************************
* Procedure TNode.UpdateNodeState
* Updates the node's current state.
* Receives data from the bus, acts on the data, transmits data, initiates and continues to process node functions such as resolving contention, soliciting a successor, adding to ring, passing the token, etc.
**************************************************************************************************************************
}
procedure TNode.UpdateNodeState(DC: HDC);

{==================================================================================================================
Procedure Xmit
Fills the OutputBuffer with data to transmit on the bus.
==================================================================================================================}
procedure Xmit(pFrom, pTo: integer; pFrame: T_FrameType;
                pArg1, pArg2: integer);
begin

{------------------------------------------------------------------------------------------------------------------
| Set output buffer addresses, frametype, and arguments to those given in the input parameters
| Set the output buffer transmit flag to true
------------------------------------------------------------------------------------------------------------------}
With OutputBuffer do
begin
  FromAddress := pFrom;
  ToAddress := pTo;
  FrameType := pFrame;
  FrameArg1 := pArg1;
  FrameArg2 := pArg2;
  IsTransmitting := true;
end; {with}
end; {Xmit}

{==================================================================================================================
Procedure UpdateCounters
Increments the nodes counters.
==================================================================================================================}
procedure UpdateCounters;
begin
{------------------------------------------------------------------------------------------------------------------}


Increment dead bus, adding, sending, contention, solicit successor/
        and token passing counters
        If there was activity on the bus
            Set the dead bus counter to zero
        EndIf
        If a token was sent to this node
            increment counter for when to send the next solicit successor frame
        EndIf
        inc(DeadBusCount);
        inc(AddingCount);
        inc(SendingCount);
        inc(ContentionCount);
        inc(SolicitSuccCount);
        inc(PassingTokenCount);
        if InputBuffer.IsTransmitting then
            DeadBusCount := 0;
            if (InputBuffer.ToAddress = Address)
                and (InputBuffer.FrameType = ft_Token) then
                inc(NextSolicitCount);
            end;
            {UpdateCounters}

Function GetNext2Bits
    Returns either 0, 2, 4, or 6 depending on the 2 bits of
    the address pointed to by Position.

function GetNext2Bits(Address, Position: integer): integer;
var
    Mask, Result: integer;
    TooFar: boolean;    {true if position is larger than 4}
begin
    Set the mask to represent the two bits of the address to inspect: /
        either 11000000, 00110000, 00001100, or 00000011
    Set the result to the mask AND the node's Address
    Shift result right so any '1' bits are in least significant pair
    Set the return value to twice the result
    If the bit position is greater than 4
        Set the return value to a random number in the set [2, 4, 6]
    EndIf
    TooFar := false;
    case Position of
        1: Mask := $00C0;
        2: Mask := $0030;
        3: Mask := $00C;
        4: Result := $00000010;
        5: Result := $00000002;
        6: Result := $00000004;
        else: Result := random(2, 4, 6);    {random number}
    end;
4: Mask := $0003;
else
    TooFar := true;
end; {case}

Result := Mask and Address;
Result := Result shr (2 * (4 - Position));

GetNext2Bits := 2 * Result;
If TooFar then
    GetNext2Bits := 2 * (trunc(3 * Random) + 1);
end; {GetNext2Bits}

{=================================================================================================

Procedure DropToken

Sets status when node drops the token to enter a Listen state.

===============================================

procedure DropToken;
begin
{-----------------------------------------------

Set the current Status, Action, and Step for a new Active node
Set the previous and next addresses to zero
Set the status flags so node is not passing token, not soliciting / 
a successor, not trying to add to ring, not deleting from / 
the ring, not token owner, and not sending WhoFollows frames
Clear the output buffer transmit flag
-----------------------------------------------}

Status := st_Active;
Action := ac_Add;
Step := sp_Listen;
PrevAddress := 0;
NextAddress := 0;
NextSolicitCount := 0;
IsPassingToken := false;
IsInitSolicit := false;
IsSolicitSucc := false;
IsAddNode := true;
IsAdding := false;
IsDeleting := false;
IsTokenOwner := false;
IsWhoFollows := false;
OutputBuffer.IsTransmitting := false;
end; {DropToken}
Procedure GetToken
Sets status when node receives the token.

procedure GetToken;
begin
{----------------------------------------}
    Set the current Status, Action, and Step for a new Token node
    If node is currently adding to ring
        Set the counter until next solicit successor frame to random num
    EndIf
    Set Initial Solicit Successor flag to Initialization status flag
    Set status flags so the node is not initializing, not trying to /
        Add to ring, not soliciting a successor, not already sentdata,/
        is the token owner, and not resolving contention
    Clear the contention resolving variables
{----------------------------------------}
    Status := st_Token;
    Action := ac_Token;
    Step := sp_GetToken;
    if IsAdding then
        NextSolicitCount := round(C_NextSolicitMax * Random);
        IsInitSolicit := IsInitializing;
        IsInitializing := false;
        IsAdding := false;
        IsAddNode := false;
        IsTokenOwner := true;
        IsSolicitSucc := false;
        AlreadyAutoSent := false;
        IsResolvingContention := false;
        PrevAddress := InputBuffer.FromAddress;
        AddressBits := 0;
        WhenToSend := 0;
    end; {GetToken}

Procedure PassToken
Handles token passing duties.
This routine is called for the first time when a token needs to be
passed. After that, it is called to confirm that the token made
it to the next node. In several cases, this routine relies on the
fact that CheckInputBuffer routine will see a valid frame on the
bus and set token passing status flag to false, thereby confirming
that the token was passed.

procedure PassToken;
Procedure PutToken
  Transmits a token frame and updates node status
procedure PutToken;
begin
  Xmit(Address, NextAddress, ft_Token, 0, 0);
  Action := ac_Token;
  Step := sp_PutToken;
  IsTokenOwner := false;
  IsWhoFollows := false;
end; {PutToken}

begin
begin
{------------------------}
  *** Beginning of the main PassToken procedure
begin
  If we are passing the token and are not receiving a transmission
    Set Step to Confirm
  EndIf
  if (IsPassingToken) and
    not ((InputBuffer.IsTransmitting) and
       (InputBuffer.ToAddress = Address)) then
    Step := sp_Confirm;
  {-----------------------}
  If we are not in the process of passing the token (first time in routine)
    Reset the token passing counter to zero
    If we are changing addresses
      Set the status flags so node is no longer changing addresses,
      and is out of the ring.
    If we are soliciting a successor for initialization
      Reset the Next Solicit counter to zero
    EndIf
    Set node status flags so the node is passing the token, is not
    sending Who Follows frames, is not soliciting a successor/
    for initialization, is not resolving Solic. Succ. contention,/    and is not currently soliciting a successor.
    Call PutToken to transmit a token frame and update status
  if not IsPassingToken then
    begin
      PassingTokenCount := 0;
      if IsChangingAddress then
        begin
          IsChangingAddress := false;
IsAddNode := true;
end {if}
if IsSolicitSucc then
NextSolicitCount := 0;
IsPassingToken := true;
IsWhoFollows := false;
IsInitSolicit := false;
IsSolicitContention := false;
IsSolicitSucc := false;
PutToken;
end {if}

---

*** This is not first time into this routine since token passing
*** started.  We are in the process of passing token.  If the
*** counter is two, then we have sent token once and are now going
*** to check for a valid frame on the bus from the new token owner.
*** If the counter is 4, we sent token once, did not hear a valid
*** frame and resent the token.  We will again check for a valid
*** frame from new token owner and will send a Who-Follows frame
*** if we do not hear one.

---

Else  (we are currently passing token)
Case the token passing counter of
  counter = 1:
    Set the Status to Active
    If there was contention when we sent the token
      (this would happen if we have the same address as
       another node and we both received and sent the
       token at the same time.)
    Call DropToken to update node status
  counter = 2 or 4:
    If there was activity on bus and there was no contention
      Set flag so node is no longer passing token
      If this node is currently the token owner
        (this node can be the token owner /
         if it just received the token back/
         from the other node in the ring.)
    Set flag so the node is not currently deleting from ring
Else  (not token owner)
  If node is not receiving a transmission
    Set Status, Action, and Step to an Active Listener
EndIf
EndIf (token owner)
---

else  (We are in the process of passing token)
case PassingTokenCount of
  1):
    begin
      Status := st_Active;
      if InputBuffer.Contention then


DropToken
end; {counter = 1}

2, 4:
begin
if inputBuffer.IsTransmitting and not
InputBuffer.Contention then
begin
IsPassingToken := false;
if IsTokenOwner then
IsDeleting := false
else
begin
if not ((InputBuffer.IsTransmitting) and
(InputBuffer.ToAddress = Address))
then
begin
Status := st_Active;
Action := ac_Listen;
Step := sp_Listen;
end; {if}
end; {else not token owner}
end {if IsTransmitting.}

{-----------------------------------------------|
Else  {no valid frame heard}
If the token passing counter is 2 then
Call PutToken to retransmit token and update status
Else
Set status flag that we are sending Who-Follows frames
Transmit a Who-Follows frame
EndIf  {activity on the bus}
{-----------------------------------------------|

else  {no successful transmission heard}
if PassingTokenCount = 2 then
PutToken
else
begin
IsWhoFollows := true;
Xmit(Address, 0, ft_WhoFollows, NextAddress, 0);
end
end; {counter = 2 or 4}

{-----------------------------------------------|
*** At this point, Who-Follows frame has been sent with no response
{-----------------------------------------------|
Case counter = 6:
Set the status flag that we are sending Who-Follows frames
Transmit a Who-Follows frame
{-----------------------------------------------|
6:
begin
IsWhoFollows := true;
Xmit(Address, 0, ft_WhoFollows, NextAddress, 0);
end; {counter = 6}

**At this point, two Who-Follows frames have been sent with no response. Assume the network ring has failed. Reinitialize the ring by soliciting a successor (open to all nodes).**

Case counter = 8
Call GetToken to make this node the official token owner
Set flags so node is not sending Who-Follows frames,/
is not deleting from ring, and is not in process/
of soliciting a successor at this time.
Set the previous and next addresses to zero
Set next solicit successor counter to its max + 1 so we will/
send a solicit successor frame.

8:
begin
IsWhoFollows := false;
IsDeleting := false;
GetToken;
PrevAddress := Address;
NextAddress := Address;
IsSolicitSucc := false;
NextSolicitCount := C_NextSolicitMax + 1;
end; {counter = 8}

**At this point, we solicited a successor and got no valid reply**

Case counter = 11:
If we are not still soliciting a successor
Call GetToken to make this node the official token owner
Set status flags so this node is soliciting a successor/
or initialization, is not sending Who-Follows frames,/
is not deleting from the ring, is not in the process /
of soliciting a successor at this time.
EndIf
EndCase

11:
if not IsSolicitSucc then
begin
IsWhoFollows := false;
IsDeleting := false;
GetToken;
PrevAddress := Address;
NextAddress := Address;
IsSolicitSucc := false;
IsInitSolicit := true;
end; {if}
end; {case PassingTokenCount}
end; {PassToken}

{===============================================
Procedure SendData
Handles all data sending and confirming duties for a token holding node. This routine is initially called to send the data to another node. After that is it called to confirm proper transfer of the data took place. In several cases, this routine relies on the fact that the CheckInputBuffer routine will see a valid response on bus and set sending data status flag to false, thereby confirming that the data was transmitted successfully.
===============================================}
procedure SendData;
{===============================================
Procedure PutData
Places data on the bus and updates status.
===============================================}
procedure PutData;
begin
    Xmit(Address, SendToAddress, ft_Data, 0, 0);
    inc(NumberSent);
    SendingCount := 0;
    Action := ac_Send;
    Step := sp_Send;
end; {PutData}

{-----------------------------------------------}

begin {SendData}
{
-----------------------------------------------

*** Beginning of the main SendData procedure
-----------------------------------------------

if we have not exceeded the max number of transmissions to try
    If we are not confirming a transmission
        Set the status flags so the node is currently sending data, /
            and is not ready to send any new data
        Set the number already sent to zero
        Call PutData to transmit the data and update status
-----------------------------------------------

if NumberSent < C_MaxNumberToSend then
    if not IsSending then
        begin
            AlreadyAutoSent := true;
            IsSending := true;

-----------------------------------------------

}
IsReadyToSend := false;
NumberSent := 0;
PutData;
end {if}

---

*** At this point we have already sent data and need to confirm

Else  (We are confirming a previous transmission)
  Set the current Step to Confirm
  If we have timed out or received a Negative Acknowledgement
    Call PutData to re-send data and update status
  EndIf

else  [We are in the process of sending/confirming]
  begin
    Step := sp_Confirm;
    if (SendingCount >= C_SendTimeOut) or
       (((InputBuffer.ToAddress = Address) and
          (InputBuffer.FrameType = ft_Nak)) then
        PutData
    end {else}

---

*** We have already sent the *maximum* number of
*** re-transmissions allowed. If we cannot confirm the last one
*** we will stop trying.

Else  (we have already sent the *maximum* number allowed)
  If we have timed out or received a Negative Acknowledgement
    Set the status flag that node is no longer sending data
    Reset the number already sent to zero
  Else
    Set the current Step to Confirm
  EndIf
EndIf

else  [We have reached the max number of re-sends]
  if (SendingCount >= C_SendTimeOut) or
     (((InputBuffer.ToAddress = Address) and
       (InputBuffer.FrameType = ft_Nak)) then
    begin
      IsSending := false;
      NumberSent := 0;
    end {else if}
  else
    Step := sp_Confirm;
end; {SendData}
Procedure ResolveContention
Resolves contention state during Initialization or Adding to ring. This procedure is called from the CheckAddNode and CheckInitialize procedures when they are having contention problems. Contention is resolved checking the first pair of bits in the nodes address byte and waiting either 0, 2, 4, or 6 clock cycles to resend depending on whether the two bits are 00, 01, 10, or 11. If there is still contention, the second pair of bits is used. If there is still contention, the third and then the fourth pair are used. After four rounds of trying to resolve the contention, this routine should not be called. If it is it waits a random number (2, 4, or 6) before sending instead of using address bits.

procedure ResolveContention;
var
ConFrameType: T_FrameType;

begin
{

------------------------
If node is in the process of adding into the ring
  Set the type of frame to Set Successor
EndIf
If node is in the process of initializing the ring
  Set the type of frame to Claim Token
EndIf
If node just sent a frame or is not already resolving contention
  If this node is in the process of resolving a contention
    Set the current Step to Resolve
    Increment pair of address bits to examine to resolve contention
    Reset the contention counter to zero
    Call Next2Bits function to find out when node transmits next
    If current contention step is > than 4 and there is still/ contention on the bus
      Assume there is a node with a duplicate address and reset / the node's address to a random number and update / the state and status of node.

EndIf

------------------------
if IsAdding then
  ConFrameType := ft_SetSucc;
if IsInitializing then
  ConFrameType := ft_ClaimToken;

if (DidSend = ConFrameType) or (not IsResolvingContention) then
  if IsResolvingContention then
    begin
      Step := sp.Resolve;
      inc(AddressBits);
      ContentionCount := 0;
    end


WhenToSend := getNext2Bits(Address, AddressBits);
If (AddressBits > 4) and (InputBuffer.Contention) then
begin
  IsInitializing := false;
  IsAdding := false;
  IsResolvingContention := false;
  Action := ac_Add;
  Step := sp_Listen;
  Address := trunc(Random * 99) + 1;
  AddressBits := 0;
  WhenToSend := 0;
end {if}
end {else if}

{--------------------------------------------------------------------------}

Else (this is the first contention)
  Set the status flag that node is currently resolving contention
  Set the current Step to Resolve
  Reset the pair of address bits to examine to the first pair
  Call getNext2Bits to find out when this node transmits next
EndIf
{--------------------------------------------------------------------------}

else {first contention}
begin
  IsResolvingContention := true;
  Step := sp_Resolve;
  ContentionCount := 0;
  AddressBits := 1;
  WhenToSend := getNext2Bits(Address, AddressBits);
end {else}

{--------------------------------------------------------------------------}

*** At this point we are just waiting for our turn to send the next
*** frame while resolving the contention. If we hear another frame
*** on the bus before we send, we will back off.

Else (we did not just send a frame)
  Set the current Step to Resolve
  If there was activity on bus and it was not a resolve cntrl frame
      Set the current Status, Action, and Step for a new Active node
  Reset next and previous addresses, pair of address bits, and/
    the number of time slots until next transmission to zero
  Set status flags so node is trying to add to the ring, not in/
    the process of adding, is not initializing the ring, is not/
    resolving contention, and is not soliciting a successor.
EndIf
EndIf
{--------------------------------------------------------------------------}

else {Did not just send a frame}
begin
  Step := sp_Resolve;
if (InputBuffer.IsTransmitting) and (not (InputBuffer.FrameType = ft.Resolve)) then begin
  Status := st_Active;
  Action := ac_Add;
  Step := sp_Listen;
  PrevAddress := 0;
  NextAddress := 0;
  AddressBits := 0;
  WhenToSend := 0;
  IsAddNode := true;
  IsAdding := false;
  IsInitializing := false;
  IsResolvingContention := false;
  IsInitSolicit := false;
  IsSolicitSucc := false;
end {if}
end; {else}

{---------------------------------------------------------------------
If contention counter is equal to the number of time slots to wait
  If node is initializing the ring
    Set current Step to Claim Token
    Transmit a claim token frame
  EndIf
  If node is currently adding to the ring
    Set the current Step to Set Successor
    Transmit a set successor frame
  EndIf
EndIf
{---------------------------------------------------------------------}
If ContentionCount = WhenToSend then begin
  if IsInitializing then begin
    begin
      Step := sp_ClaimToken;
      Xmit(Address, 0, ft_ClaimToken, 0, 0);
    end; {if}
  if IsAdding then begin
    begin
      Step := sp_SetSucc;
      Xmit(Address, PrevAddress, ft_SetSucc, Address, 0);
    end; {if}
  end; {if}
end; {ResolveContention}

{=====================================================================
Procedure CheckInitialize
  Performs ring initialization if it is needed.
}
This routine checks to see if bus has been dead (no activity) longer than a constant number of clock cycles. If it has, the node issues a claim token frame. If there was no contention, the node claims itself the token holder and moves on to solicit a successor in CheckSolicSuucc routine within UpdateTokenState. If it finds contention on bus when it sends its claim token, this routine calls ResolveContention to resolve the contention for it.

```pascal
procedure CheckInitialize;
var
  InContention: boolean;  {node was in contention on last transmission}
begin
  '{
  If the bus has been dead (no activity) for longer than max time /
    and we are not currently initializing the ring
  If this node owns the token
    Set the status flags so node is currently initializing ring, /
      is not token owner, is not passing the token, is not /
        soliciting successor for initialization, is not resolving /
          contention, is not trying to or currently adding into ring.
    Set the next and previous addresses to zero
    Set the Status, Action, and Step for an initializing node
    Transmit a claim token frame
  EndIf
  if (DeadBusCount > C_DeadBusMax) and (not IsInitializing) then
  begin
    IsInitializing := true;
    IsTokenOwner := false;
    IsPassingToken := false;
    IsInitSolicit := false;
    IsResolvingContention := false;
    IsAddNode := false;
    IsAdding := false;
    NextAddress := 0;
    PrevAddress := 0;
    Status := st_Active;
    Action := ac_Initialize;
    Step := sp_ClaimToken;
    XMit(Address, 0, ft_ClaimToken, 0, 0);
  end; {if}
  '{
  If this node is currently initializing the ring
    Set the contention on last try flag to false
    Set the current action to initialize
  '{
  if IsInitializing then
```
begin
  InContention := false;
  Action := ac_Initialize;

{                        |}

| If this node just sent a claim token frame last time |
| If there was no contention on the last transmission |
| Set the previous and next addresses to the node's address |
| Call GetToken to make node the current token owner |
| Set status flag so node needs to solicit a successor for /|
| initialization |
| Else (We sent a claim token but ran into contention) |
| Set the contention on last try flag to true |
| EndIf (contention last time) |
| EndIf (just sent claim token) |

{                        |}

| if DidSend = ft_ClaimToken then |
| if not InputBuffer.Contention then |
| begin |
| PrevAddress := Address; |
| NextAddress := Address; |
| GetToken; |
| IsInitSolicit := true; |
| end {if} |
| else {This node Did Send last time but there was |
| contention} |
| InContention := true; |

{                        |}

| If we just got contention or are already resolving a contention |
| Call ResolveContention to resolve the contention problem |
| EndIf |
| EndIf (We are currently Initializing) |

{                        |}

| if InContention or IsResolvingContention then |
| ResolveContention; |
| end; {if IsInitializing} |
| end; {CheckInitialize} |

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

Procedure UpdateTokenState
| Performs state update for an Token Owning network node. |
| Performs soliciting of successors, deleting from the ring, |
| sending data to other nodes (includes AutoSend function), and |
| token passing. |

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

procedure UpdateTokenState;
Procedure CheckSolicitSucc
    Performs any Solicit Successor processing needed for
    the token holding node. This routine solicits a successor
    after initializing the ring or after a constant number of
    token possessions have occurred.

procedure CheckSolicitSucc;
begin

    {------------------------------------------------------------------------}
    If we need to solicit successor for initialization or the next
    solicit successor count is greater than the max
    If we are not already soliciting a successor
    Transmit a solicit successor frame
    Set status flags so node is soliciting a successor and is not
    in resolving a solicit successor contention
    Reset the solicit successor counter to zero
    Set the current Step to Solicit Successor
    {------------------------------------------------------------------------}
    if (IsInitSolicit) or (NextSolicitCount > C_NextSolicitMax) then
    if not IsSolicitSucc then
        begin
            XMit(Address, 0, ft_SolSucc, Address, NextAddress);
            IsSolicitSucc := true;
            IsSolicitContention := false;
            SolicitSuccCount := 0;
            NumSolicitContentions := 0;
            Action := ac_Poll;
            Step := sp_SolSucc;
        end [if]

    {------------------------------------------------------------------------}
    *** We have already sent a solicit-successor frame and are looking
    ** to see what the response was.

    Else (We are already soliciting a successor)
    If the counter is greater than 1
    If there was no contention last transmission
    If we received a set-successor frame
        Call PassToken to send the token to our new successor
    Else  (no set-successor frame sent)
        If a node other than this one transmitted something and /  
          we are not changing addresses
            Call DropToken to drop the token and go to a listener
        state
    {------------------------------------------------------------------------}
    else  [We are currently soliciting a successor]
        if SolicitSuccCount > 1 then
            if not InputBuffer.Contention then
                if (InputBuffer.FrameType = ft_SetSucc) and
(InputBuffer.ToAddress = Address) then
begin
  IsPassingToken := false;
  PassToken;
end {if}
else {There was no Set-Successor frame transmitted}
begin
  if (InputBuffer.IsTransmitting) and
      (InputBuffer.FromAddress <> Address) and
      (not IsAddressNode) then
    DropToken

*** There was no response to solicit successor frame we sent. It
*** appears as though nobody wants to add into ring. If this was
*** a routine solicit then go on with normal business and stop
*** soliciting. If this was for initialization then drop the token.

Else (there was no response at all)
  If we are soliciting a successor for intialization
  Call DropToken to drop token and go to listen state
  EndIf
  Reset the next solicit successor counter to zero
  Set status flags so node is not soliciting a successor/
  in any way
  EndIf (there was a transmission)
EndIf (we received a set-successor frame)

else {No frame of any kind was sent}
  if not IsSolicitContention then
  begin
    If IsInitSolicit then
      DropToken;
    NextSolicitCount := 0;
    IsSolicitSucc := false;
    IsSolicitContention := false;
    IsInitSolicit := false;
    end {if}
  end {else}

*** We have already sent the solicit successor frame but there was
*** more than one responding node. We must now resolve this
contention.

Else (the nodes responding to the solicit successor contended)
  If we aren't currently resolving solicit successor contention
  Set status flag so node is currently resolving a solicit/
  successor contention
  Reset the solicit successor counter to zero
  Transmit a resolve-contention frame
Set the current Step to Resolve
else {We are soliciting a successor but got contention}
    if not IsSolicitContention then
        begin
            inc(NumSolicitContentions);
            IsSolicitContention := true;
            SolicitSuccCount := 0;
            Xmit(Address, 0, ft_Resolve, 0, 0);
            Step := sp_Resolve;
        end {if}
    }

*** We are already resolving previous contention. The maximum time
*** it should take to resolve a contention is 4 contention rounds.
*** So after 4 contentions, if we are still having problems,
*** something's wrong. It's time to retransmit solicit successor
*** frame and try all over. If that does not work (after 4 more
*** contentions), give up and get rid of the token.

Else {we are already resolving a previous contention}
    If there have been more than 4 contentions
        Transmit another solicit-successor frame
        Set the current Step to Solicit Successor
    EndIf
    If there have been more than 9 contentions total
        If we are soliciting a successor for initialization
            Call DropToken to drop the token and update status
        Else
            Call PassToken to pass the token and update status
        EndIf
    EndIf (contentions =10)
EndIf (already resolving contention)
else {We are already resolving a previous contention}
    begin
        inc(NumSolicitContentions);
        if NumSolicitContentions = 5 then
            begin
                IsSolicitContention := false;
                SolicitSuccCount := 0;
                Xmit(Address, 0, ft_SolSucc, Address,
NextAddress);
                Step := sp_SolSucc;
            end {if}
        if NumSolicitContentions = 10 then
            begin
                if IsInitSolicit then
                    DropToken
                else
                    end {if}
EndIf (already resolving contention)
begin
  IsPassingToken := false;
  PassToken;
end;
end; {if
end; {else
end; {CheckSolicitSucc

procedure CheckDeleteNode;
begin

{------------------------------------------------------------------

Procedure CheckDeleteNode

Handles node deletion tasks for a token
node that wants to be deleted from the ring.
Deletion is handled in two steps. First, the
status is set to currently deleting and a set-
successor frame is passed to the previous node
pointing to the next one. On the next cycle,
the currently deleting flag is set to false so
the token passing routine will pass the token.

{------------------------------------------------------------------

procedure CheckDeleteNode;
begin

{------------------------------------------------------------------

If node is in the process of deleting from the ring
  Set the current Action to Delete
  Set status flags so node is done deleting and is not a member /
    of the ring (trying to add).
Else  (node is not currently deleting)
  If this node is still the token owner,
    (wants to delete or is changing addresses), and is not /
    currently soliciting a successor
    Set the current Action to Delete and Step to Set Successor
    Set the status flag so node is currently deleting from ring/
      and is no longer changing addresses
    Transmit set successor frame to prev address with set value/
      of next address
  EndIf
EndIf (node is deleting)

if IsDeleting then
begin
  Action := ac_Delete;
  IsDeleting := false;
  IsAddNode := true;
end {if
else
  if (IsTokenOwner) and (IsDeleteNode or IsAddressNode) and
    (not IsSolicitSucc) then
begin
  Action := ac_Delete;
end
```
Step := sp_SetSucc;
IsDeleting := true;
IsAddressNode := false;
Xmit(Address, PrevAddress, ft_SetSucc, NextAddress, 0);
end; {if}
end; {CheckDeleteNode}

function GetSendAddress: integer;
var
  TempPointer: PNode;
  CheckedSet: set of 0..7;
  RandInt: 0..7;

begin

{------------------------------------------------------------------}

  *** Get random address for nodes 1 through 8 until the address does
  *** not equal this node's address. A set of nodes checked so far
  *** is kept in case all of the nodes have same address. In this
  *** case it's possible to search for a different address forever if
  *** we don't track which nodes in the collection we've checked.

  Initialize the set of nodes checked so far to null
  Repeat
    Get random number between 0 and 7(node indices in collection)
    Get the pointer to node with the random index from collection
    Add the index to the set of nodes checked so far
    Until we found an address not equal to this node's address or/
    we have checked every node in the collection (all had /
    the same address)
  Return the address found or zero if it is this node's address

  CheckedSet := [];

repeat
  RandInt := trunc(8 * Random);
  TempPointer := PNodeCollection^.At(RandInt);
  CheckedSet := CheckedSet + [RandInt];
  until (TempPointer^.Address <> Address) or
     (CheckedSet = [0,1,2,3,4,5,6,7]);

GetSendAddress := TempPointer^.Address;
If TempPointer^.Address = Address then
  GetSendAddress := 0;
end; {GetAutoSendAddress}

{-------------------------------------------------------------------}

begin {UpdateTokenState}  
{-------------------------------------------------------------------}

*** Start of main routine for subroutine UpdateTokenState

Call CheckDeleteNode to see if the node needs to delete from ring
If node does not want to delete from ring
   Call CheckSolicitSucc to solicit a successor if we need to
EndIf
If node's not ready to send, has not already sent data this cycle,/
   is not currently sending data, is not trying to delete /
   from the ring, is the token owner, and AutoSend is active
   Set flags so node has already sent data this cycle and is /
   ready to send data now.
   Set the send to address to the result of function GetSendAddress
EndIf
{-------------------------------------------------------------------}

CheckDeleteNode;

if (IsSolicitSucc) or ((not IsDeleteNode) and (not IsDeleting))
then
   CheckSolicitSucc;
   if (not IsReadyToSend) and (IsTokenOwner) and (IsAutoSend) and
   (not IsSending) and (not AlreadyAutoSent) and (not
IsDeleteNode) then
   begin
   AlreadyAutoSent := true;
   IsReadyToSend := true;
   SendToAddress := GetSendAddress;
   end;
{-------------------------------------------------------------------}

If node is token owner, ready to send data, is not soliciting a /
   successor, and is not deleting from the ring
   Set flag so node is not currently sending data
   Call SendData to handle data sending responsibilities
EndIf
{-------------------------------------------------------------------}

if (IsTokenOwner) and (IsReadyToSend) and (not IsSolicitSucc) and
   (not IsDeleting) and (not IsAddressNode) then
begin
   isSending := false;
   SendData;
end; {if}
If node is token owner, not ready to send data, is not soliciting a successor, and not deleting from the ring, not currently sending data, and is not currently passing the token or node is token owner and bus has been dead for more than a constant time
Set flag so node is not currently passing the token
Call PassToken to handle token passing responsibilities
EndIf

if ((not IsReadyToSend) and (not IsSolicitSucc) and (not IsSending) and (IsTokenOwner) and (not IsDeleting) and (not IsPassingToken) and (not IsAddressNode)) or ((IsTokenOwner) and (DeadBusCount > C_DeadBusTokenMax)) then
begin
  IsPassingToken := false;
  PassToken;
end; {if}
end; {UpdateTokenState}

Procedure CheckAddNode
Performs processing for a node trying to add into the logical ring. This routine is called from UpdateActiveState when the node is trying to add to the logical ring (is not currently a member). The routine checks to see if solicit successor frame was sent that this node's eligible to respond to. If there is one, a set-successor frame is sent to the originating node. If the token is returned, the CheckInputBuffer routine updates the adding status and calls GetToken. If there was contention for the solicit, then this routine calls ResolveContention until the contention is resolved.

procedure CheckAddNode;
var
  ValidAddress,
  InContention: boolean;

begin
{ Set the contention on last transmit flag to false
if node is not currently adding to the ring
  if there was a solicit-successor frame sent
    Determine if this node's eligible to respond to solicit successor

  InContention := false;
if not IsAdding then
  if (InputBuffer.FrameType = ft_SolSucc) then
    begin
      with InputBuffer do
        begin
          ValidAddress := false;
          if FrameArg1 = FrameArg2 then
            ValidAddress := true;
          if (FrameArg1 < FrameArg2) and (Address > FrameArg1)
            and (Address < FrameArg2)
            then
              ValidAddress := true;
          if (FrameArg1 > FrameArg2) and ((Address > FrameArg1)
            or (Address < FrameArg2)) then
            ValidAddress := true;
        end; {with}
      end; {if}

    {-----------------------------}
    *** This node can respond to solicit successor frame that was sent.
    *** The node sets its next and prev addr and sends a set-successor
    *** frame in response. If it ends up losing contention battle later,
    *** its next and prev addresses will be reset to zero later.
    {-----------------------------}
    If this node is eligible to respond
      Set status flags so node is currently adding to ring and
      is not resolving contention
      Reset the adding counter to zero
      Transmit a set-successor frame to sender of the solicit
      Set previous and next addresses to those of solicit frame
      Set the current Step to Set-Successor
    {-----------------------------}
    if ValidAddress then
      begin
        IsAdding := true;
        IsResolvingContention := false;
        AddingCount := 0;
        Xmit(Address, InputBuffer.FromAddress, ft_SetSucc,
          Address, 0);
        PrevAddress := InputBuffer.FrameArg1;
        NextAddress := InputBuffer.FrameArg2;
        Step := sp_SetSucc;
      end; {if}
  else
    {-----------------------------}
    *** We are currently trying to add to the ring. We will either get
    *** a contention message or the token. If the token is sent, the
    *** CheckInputBuffer routine will handle the status changes
    {-----------------------------}
Else (we are in the process of adding a node to the ring)
    If the adding counter > 1
        If a resolve-contention frame was sent
            Set the contention on last transmission flag to true
        Else
            If there was no initial reply at all (when counter is 2)
                Set the status flags so node is not currently adding
        EndIf
    EndIf (a resolve frame sent)
EndIf (counter > 1)
EndIf (we are already adding)

else {We are in the process of inserting node into ring}
    if AddingCount > 1 then
        if (InputBuffer.FrameType = ft.Resolve) then
            InContention := true
        else
            if (AddingCount = 2) and
                not ((InputBuffer.IsTransmitting) and
                     (InputBuffer.ToAddress = Address)) then
                begin
                    IsAdding := false;
                    PrevAddress := 0;
                    NextAddress := 0;
                end; {if}
            {else This should be the token if it wasn't a resolve frame}
            {The status should have been updated by GetToken called from}
            {CheckInputBuffer when the token came in}
        {else} {CheckAddNode}
    {else} {CheckAddNode}

Procedure CheckInputBuffer
Handles generic message receiving duties for a node.
The situation specific routines such as PassToken, CheckSolicitSucc,
CheckAddNode, and SendData handle inputs for their special cases.
This routine, however, is counted on by these others to handle some
generic cases and update status of the node accordingly, depending
on the nodes current status and type of input.
If the node is the token owner then it checks to see if other nodes
have sent any frames that only token owners are allowed to. If it
sees any of these frames, it assumes that another node also owns the
token and drops the token.

procedure CheckInputBuffer;
begin
  -----------------------------;
  If there was activity on the bus from another node
  Case frame type of:
  Case frame type = Token:
    If node is the token owner
      Call DropToken to drop token and update status
      EndIf
    If this node's Status is Active and token was sent to this/
      address
      Call GetToken to get the token and update status
      EndIf
  with InputBuffer do
    if (IsTransmitting) and (FromAddress <> Address) then
      case FrameType of
        ft_Token: begin
          if IsTokenOwner and not IsChangingAddress
            then
              DropToken;
              if Status = st_Active then
                if ToAddress = Address then
                  GetToken;
        end; {ft_Token:}
  Case frame type = Data:
    If this node is the token owner
      Call DropToken to drop the token and update status
      EndIf
    If the Status of this node is Active or Passive
      If the data was sent to this address
        Set the current Action to Receive
      If there was no error in the transmitted data
        Transmit an acknowledgement to the sending node
        Set the current Step to Ack
      Else (there was error in the transmission)
        Transmit a negative acknowledgement to sending node
        Set the current Step to Nak
        EndIf
    EndIf (data to this address)
    EndIf (status is active or passive)
  -----------------------------;
ft_Data: begin
  if IsTokenOwner and not IsChangingAddress then
    DropToken;
    if (Status = st_Active) or (Status = st_Passive) then
      if ToAddress = Address then
        begin
          Action := ac_Receive;
          if not Error then
            begin
              XMit(Address, FromAddress,
              ft_Ack, 0, 0);
              Step := sp_Ack;
            end {if}
        else
          begin
            XMit(Address, FromAddress,
            ft_Nak, 0, 0);
            Step := sp_Nak;
          end; {else}
      end;
    end; {ft_Data:}

{---------------------------------------------------------------------;
|
| Case frame type = Set Successor:
| If the current Status is Active or Token
|  if the frame was sent to this node
|    Set next address to the next address from the frame
|    If we are currently passing the token
|      Set status flag so node is no longer passing token
|      If we are sending WhoFolows frames(this frame's a/ reply to a who-follows frame)
|      Call PassToken to pass the token to replying node
| EndIf
| EndIf (passing token)
| EndIf (sent to this node)
| EndIf (status is active or token)
{---------------------------------------------------------------------;

ft_SetSucc: if (Status = st_Active) or (Status = st_Token) then
  if ToAddress = Address then
    begin
      NextAddress := FrameArg1;
      if IsPassingToken then
        begin
          IsPassingToken := false;
        if IsWhoFolows then
          PassToken;
        end; {if}
    end; {ft_SetSucc: if}
Case frame type = Solicit Successor:
   If the current Status is Active or Token
      If node is the token owner
         Call DropToken to drop token and update status
      EndIf
   EndIf
   If the from and to addresses of solicit are the same /
      (this is open for all nodes to respond)
      and this node is already a member of the ring
      Set flags so node is not a member of the ring and/
      is not in the process of adding to the ring
      Call CheckAddNode to respond to solicit-successor frame
   EndIf
   EndIf (status is active or token)

ft_SolSucc: if (Status = st_Active) or (Status = st_Token)
then
   begin
      if IsTokenOwner and not IsChangingAddress
      then
         DropToken;
         if (FrameArg1 = FrameArg2) and
            (not IsAddNode) then
            begin
               IsAdding := false;
               IsAddNode := true;
               CheckAddNode;
            end;
      end; {ft_SolSucc:}

Case frame type = Who Follows:
   If this node is the token owner
      Call DropToken to drop the token and update status
   EndIf
   If current status is Active
      If previous address equals the who follows value
         Transmit a set-successor frame to who-follows sender
         Set the current Step to Set Successor
      EndIf
   EndIf (status is active)

ft_WhoFollows: begin
   if IsTokenOwner and not IsChangingAddress then
      DropToken;
      if Status = st_Active then
         if PrevAddress = FrameArg1 then
            begin

ft_SetSucc, Xmit(Address, FromAddress, Address, 0);
    Step := sp_SetSucc;
end;
end; {ft_WhoFollows: }

{--------------------------------------------------------}
    Case frame type = Acknowledge:
      If current status is Active or Token
        If frame was sent to this node and node is currently/
           sending data
          Set status flag so this node's not currently sending/
             data
        Reset the number of data frames sent to zero
      EndIf
    EndIf (status is active or token)
{--------------------------------------------------------}
    ft_Ack: if (Status = st_Token) or (Status = st_Active)
      then
        if (ToAddress = Address) and (IsSending) then
          begin
            IsSending := false;
            NumberSent := 0;
          end; {ft_ACK: if}
      end; {case}
end; {CheckInputBuffer}

{===============================================
  Procedure UpdateActiveState
  Performs state update for an Active network node.
===============================================
}
procedure UpdateActiveState;
begin
{
{-----------------------------------------------}
  If node is in the process of passing the token
  Call PassToken to handle token passing/confirmation
EndIf
{-----------------------------------------------}
  If node is in the process of sending data
  Call SendData to handle data sending/confirmation
EndIf
Call CheckInitialize to handle ring initialization if it is needed
{-----------------------------------------------}
  If node is the token owner
  Call UpdateTokenState to perform node deletion, token handling/
     new data sending, or solicit successor activities needed
EndIf
If node is trying to add into the ring
   Call CheckAddNode to perform ring addition activities for node
EndIf

if IsPassingToken then
   PassToken;
if IsSending then
   SendData;
   CheckInitialize;
if IsTokenOwner then
   UpdateTokenState;
if IsAddNode then
   CheckAddNode;
end; {UpdateActiveState}

begin {UpdateNodeState}
{|----------------------------------------------|
   Call UpdateNodeStatus to adjust status of node for one clock pulse
   Call UpdateCounters to update all of the counters
   If this node is active (either token holding or just active)
      Call UpdateActiveState to perform receiving, transmitting, state /
      changes, token handling, etc. for one clock pulse
   EndIf
   If this node is not transmitting this time
      Update the last frame sent variable to Null (idle)
   Else
      Update the last frame sent variable to the frame being sent
   EndIf
{|----------------------------------------------|
   UpdateNodeStatus;
   UpdateCounters;
   CheckInputBuffer;
   if (Status = st_Token) or (Status = st_Active) then
      UpdateActiveState;
   end; {TNode.UpdateNodeState}

{|----------------------------------------------|

{***************************************************************************
* Procedure TNode.TimerTick
*   Responds to a tick of the Windows system timer clock.  
***************************************************************************

procedure TNode.TimerTick(DC: HDC);
begin

   {------------------------------------------
   Call UpdateNodeState to do status and state processing for one
   clock pulse
   Call UpdateNodeDisplay to display any status changes for the node
   ------------------------------------------}
UpdateNodeState(DC);
UpdateNodeDisplay(DC);
end; {TNode.TimerTick}

{*******************************************************************************
* Procedure TNode.KillToken
* Removes the token from the node if it has the token.
*******************************************************************************
}
procedure TNode.KillToken;
begin

   {------------------------------------------
   Set the status flags so node is not token owner or passing token
   If there is a token in the input buffer
   Clear the token out of the buffer
   EndIf
   If the node's current Status = Token
   Set the node Status, Action, Step for an Active Listening node
   EndIf
   ------------------------------------------}
IsTokenOwner := false;
IsPassingToken := false;
if InputBuffer.FrameType = ft_Token then
   begin
      InputBuffer.FrameType := ft_Null;
      InputBuffer.IsTransmitting := false;
   end; {if}
if Status = st_Token then
   begin
      Status := st_Active;
      Action := ac_Listen;
      Step := sp_Listen;
   end; {if}
end; {TNode.KillToken}
* Procedure TNode.SetCommand
  * Sets the node's current command to the command given.

procedure TNode.SetCommand(pCommand: T_NodeCommand); begin
  Command := pCommand;
end; {TNode.SetCommand}

procedure TNode.MakeAutoSend(pIsChecked: boolean); begin
  lsAutoSend := pIsChecked;
end; {TNode.MakeAutoSend}

procedure TNode.MakeSendTo(pToAddress: integer); begin
  isReadyToSend := true;
  SendToAddress := pToAddress;
end; {TNode.MakeSendTo}
```pascal
***************
* Procedure TNode.ChangeAddress  *
* Responds to an 'Address' menu selection for this node. *
***************

procedure TNode.ChangeAddress(pNewAddress: integer);
begin
  |
  -------------------------------
  If the input parameter address is different from the node's address
  Set the node's address to the input parameter
  If this node is the token owner
    Set the flag so node will know to delete from ring gracefully
  Else
    If the node Status is Active or Token
      Set status flags so node is out of ring
    EndIf
  EndIf
EndIf
-------------------------------
if pNewAddress <> Address then
begin
  Address := pNewAddress;
  if IsTokenOwner then
  begin
    IsAddressNode := true;
    IsChangingAddress := true;
  end; {if}
else
  if (Status = st_Active) or (Status = st_Token) then
    IsAddrNode := true;
end; {if}
end; {TNode.ChangeAddress}

end. {GNodeObj}
```
unit GGlobal;
{
 ******************************
Unit Global
Contains all of the global constant, type, and variable declarations
for the Token Bus Simulation program.
******************************
}

interface
uses WObjects, {ObjectWindows Objects}
GSimObj, {TSimObject definition for the TNodeCollection declaration}
Strings; {Turbo string handlers}

const
{
----------------------------------------
-----------------
UnTyped Constants
-----------------
----------------------------------------

MaxTimerDelay controls the 'top speed' of the simulation.
The higher the number, the slower the simulation runs.
Is equal to clock pulse delay in milliseconds
----------------------------------------
C_MaxTimerDelay = 22000;

----------------------------------------
Algorithm maximums/time-out values
----------------------------------------
C_DeadBusMax = 8; {Number of clock counts without any activity on the bus before node decides to initialize a new ring}
C_DeadBusTokenMax = 6; {Number of clock counts without any activity on bus before token holding node passes the token}
C_NextSolicitMax = 5; {Number token possessions between solicit successor frames}
C_SendTimeOut = 4; {Number clock counts to wait for an ACK/NAK}
C_MaxNumberOfSend = 3; {Number frames to send/resend before giving up after receiving a NAK or timing out}

----------------------------------------
Miscellaneous color definitions that may or may not be used by sim.
Numbers are in hex in format $00BBGRR where B=Blue,G=Green,R=Red.
----------------------------------------
GrapeCol = $00804080;
LightGrapeCol = $00AA5AAA;
BurntOrange = $000B33BF;
MediumOrange = $002060FF;
RedOrangeCol = $000040FF;
BlueGreenCol = $00808040;
LightGrayCol = $00D0D0D0;
WhiteCol = $00FFFFFF;
MediumGrayCol = $00808080;

{----------------------------------------}
{Definitions for node header and border colors,
 depending on status of node}
{----------------------------------------}
C_TokenColor = RedOrangeCol;  {Token header color}
C_HeaderColor = BlueGreenCol;  {Active header color}
C_PassiveColor = MediumGrayCol;  {Passive header color}
C_NormalColor = LightGrayCol;  {Normal border color}
C_DownColor = WhiteCol;  {Down border color}

{----------------------------------------}
{Button and Scroll Bar X-coordinate positions on the screen}
{----------------------------------------}
C_ButtonPos = 7;
C_ScrollPos = C_ButtonPos + 300;
C_ErrScrollPos = C_ScrollPos + 206;

{----------------------------------------}
{Bus placement and size}
{----------------------------------------}
BusX = 10;  {Upper right corner X coordinate}
BusY = 215;  {Upper right corner Y coordinate}
BusWidth = 620;  {Width of bus in pixels}
BusHeight = 40;  {Height of bus in pixels}

{----------------------------------------}
{Node placement and sizing}
{----------------------------------------}
NodeWidth = 130;  {Width of node in pixels}
NodeHeight = 155;  {Height of node in pixels}
NodeMargin = 15;  {Left X coordinate of left side nodes}
TopNodeMargin = 45;  {Upper Y coordinate of top nodes}
HSpacing = 160;  {Horizontals space between nodes in pixels}
VSpacing = 225;  {Vertical space between nodes in pixels}

{----------------------------------------}
{Bus/Node Link dimensions}
{----------------------------------------}
LinkWidth = 20;
LinkHeight = BusY - (TopNodeMargin + NodeHeight);
LinkMargin = NodeMargin + ((NodeWidth-LinkWidth) div 2);
(Individual node coordinates, X and Y)

YPos_1_4 = TopNodeMargin;
YPos_5_8 = YPos_1_4 + VSpacing;
XPos_1_5 = NodeMargin;
XPos_2_6 = XPos_1_5 + HSpacing;
XPos_3_7 = XPos_2_6 + HSpacing;
XPos_4_8 = XPos_3_7 + HSpacing;
N1_X = XPos_1_5; N1_Y = YPos_1_4;
N2_X = XPos_2_6; N2_Y = YPos_1_4;
N3_X = XPos_3_7; N3_Y = YPos_1_4;
N4_X = XPos_4_8; N4_Y = YPos_1_4;
N5_X = XPos_1_5; N5_Y = YPos_5_8;
N6_X = XPos_2_6; N6_Y = YPos_5_8;
N7_X = XPos_3_7; N7_Y = YPos_5_8;
N8_X = XPos_4_8; N8_Y = YPos_5_8;

(Initial Node Addresses)

N1_Address = 1; N2_Address = 2;
N3_Address = 3; N4_Address = 4;
N5_Address = 5; N6_Address = 6;
N7_Address = 7; N8_Address = 8;

(Windows resource identifiers for menus and controls)

cm_N1_Active = 101;
cm_N1_InActive = 102;
cm_N1_Passive = 103;
cm_N1_Crash = 104;
cm_N1_AutoSend = 105;
cm_N1_SendTo = 106;
cm_N1_Address = 107;
cm_N2_Active = 201;
cm_N2_InActive = 202;
cm_N2_Passive = 203;
cm_N2_Crash = 204;
cm_N2_AutoSend = 205;
cm_N2_SendTo = 206;
cm_N2_Address = 207;
cm_N3_Active = 301;
cm_N3_InActive = 302;
cm_N3_Passive = 303;
cm_N3_Crash = 304;
cm_N3_AutoSend = 305;
cm_N3_SendTo = 306;
cm_N3_Address = 307;
cm_N4_Active = 401;
cm_N4_InActive = 402;
cm_N4_Passive = 403;
cm_N4_Crash = 404;
cm_N4_AutoSend = 405;
cm_N4_SendTo = 406;
cm_N4_Address = 407;
cm_N5_Active = 501;
cm_N5_InActive = 502;
cm_N5_Passive = 503;
cm_N5_Crash = 504;
cm_N5_AutoSend = 505;
cm_N5_SendTo = 506;
cm_N5_Address = 507;
cm_N6_Active = 601;
cm_N6_InActive = 602;
cm_N6_Passive = 603;
cm_N6_Crash = 604;
cm_N6_AutoSend = 605;
cm_N6_SendTo = 606;
cm_N6_Address = 607;
cm_N7_Active = 701;
cm_N7_InActive = 702;
cm_N7_Passive = 703;
cm_N7_Crash = 704;
cm_N7_AutoSend = 705;
cm_N7_SendTo = 706;
cm_N7_Address = 707;
cm_N8_Active = 801;
cm_N8_InActive = 802;
cm_N8_Passive = 803;
cm_N8_Crash = 804;
cm_N8_AutoSend = 805;
cm_N8_SendTo = 806;
cm_N8_Address = 807;
cm_NoToken = 1000;

id_Menu = 100;
id-TokenIcon = 10000;
id_Pause = 2101;
id_Step = 2102;
id_Go = 2103;
id_SpeedScroll = 2201;
id_ErrorScroll = 2202;
id_SpeedStatic = 2301;
id_ErrorStatic = 2302;
id_SepStatic = 2303;
type

[-----------------------------]
[-----------------------------]
[-----------------------------]
[-----------------------------]

{-----------------------------------}
{Link direction depending on node activity}
{-----------------------------------}
T_Direction = (Transmit, Receive, Idle);

{-----------------------------------}
{Link location relative to bus}
{-----------------------------------}
T_Side = (AboveBus, BelowBus);

{-----------------------------------}
{Command types from the menu}
{-----------------------------------}
T_NodeCommand = (nc_Active, nc_Inactive, nc_Passive, nc_Crash,
    nc_AutoSend, nc_SendTo, nc_Address);
T_NodeCommandArray = array[1..8,T_NodeCommand] of integer;

{-----------------------------------}
{Node status values}
{-----------------------------------}
T_Status = (st_Active, st_Token, st_Inactive, st_Passive, st_Down);

{-----------------------------------}
{Node Action values}
{-----------------------------------}
T_Action = (ac_Listen, ac_Token, ac_Poll, ac_Send, ac_Receive,
    ac_Initialize, ac_Add, ac_Delete, ac_Crash, ac_Wait);

{-----------------------------------}
{Node step values}
{-----------------------------------}
T_Step = (sp_Listen, sp_GetToken, sp_PutToken, sp_SolSucc, sp_SetSucc,
    sp.Resolve, sp_Send, sp_Receive, sp_ClaimToken,
    sp_Confirm, sp_Wait, sp_Ack, sp_Nak);
{Frame types allowed on the bus}
{-------------------------------------------------------------------------}
T_FrameType = (ft_Null, ft_Token, ft_Data, ft_contention, ft_SolSucc,
               ft_SetSucc, ft_RequestToken, ft_ClaimToken, ft.Resolve,
               ft_WhoFollows, ft_Ack, ft_Nak);

{-------------------------------------------------------------------------}
{Definition of a bus buffer for transmitting data.}
{Nodes have an input and output buffer. The bus has its own buffer.}
{-------------------------------------------------------------------------}
T_Buffer = record
    FromAddress: integer;         {Address of originating node}
    ToAddress : integer;         {Address of destination node}
    FrameType : T_FrameType;     {Type of frame being sent}
    FrameArg1, etc.
    FrameArg2 : integer;
    IsTransmitting: boolean;     {true if transmitting on bus}
    Contention : boolean;       {true if there is contention on bus}
    Error : boolean;            {true if there was an error in
                                transmission}
  end; {TBuffer}

{-------------------------------------------------------------------------}
{Object definition for a sorted collection.}
{Used for node and link collections}
{-------------------------------------------------------------------------}
PSortedAddressCollection = ^TSortedAddressCollection;
TSortedAddressCollection = object(TSortedCollection)
  function KeyOf(Item: pointer): pointer; virtual;
  function Compare(Key1, Key2 : pointer): integer; virtual;
end; {TSortedAddressCollection}

{typed} const

{------------------------------------------------------------- Typed Constants -------------------}
{-------------------------------------------------------------

{-------------------------------------------------------------------------}
{NodeCommand is an array of all of the menu resource i.d.'s.}
{It is used to figure out which node is being selected from the menu and}
{which command to issue to the node.}
{-------------------------------------------------------------------------}
NodeCommand: T_NodeCommandArray =
  (cm_N1_Active, cm_N1_Inactive, cm_N1_Passive, cm_N1_Crash,
   cm_N1_AutoSend, cm_N1_SendTo, cm_n1_Address),
(cm_N2_Active, cm_N2_Inactive, cm_N2 Passive, cm_N2_Crash,
cm_N2_AutoSend, cm_N2_SendTo, cm_N2_Address),
(cm_N3_Active, cm_N3_Inactive, cm_N3 Passive, cm_N3_Crash,
cm_N3_AutoSend, cm_N3_SendTo, cm_N3_Address),
(cm_N4_Active, cm_N4_Inactive, cm_N4 Passive, cm_N4_Crash,
cm_N4_AutoSend, cm_N4_SendTo, cm_N4_Address),
(cm_N5_Active, cm_N5_Inactive, cm_N5 Passive, cm_N5_Crash,
cm_N5_AutoSend, cm_N5_SendTo, cm_N5_Address),
(cm_N6_Active, cm_N6_Inactive, cm_N6 Passive, cm_N6_Crash,
cm_N6_AutoSend, cm_N6_SendTo, cm_N6_Address),
(cm_N7_Active, cm_N7_Inactive, cm_N7 Passive, cm_N7_Crash,
cm_N7_AutoSend, cm_N7_SendTo, cm_N7_Address),
(cm_N8_Active, cm_N8_Inactive, cm_N8 Passive, cm_N8_Crash,
cm_N8_AutoSend, cm_N8_SendTo, cm_N8_Address)
);

{------------------------------------------------------------------
(NodeAddress is an array containing the initial address of each node. 
It is used to verify valid addresses input by the user and to 
randomly generate addresses for the AutoSend function)
{------------------------------------------------------------------

NodeAddress: array[1..8] of integer =
(N1_Address, N2_Address, N3_Address, N4_Address, N5_Address,
 N6_Address, N7_Address, N8_Address);

{------------------------------------------------------------------
(BusMsg is an array containing the message to display on the bus 
for each frame type.)
{------------------------------------------------------------------

BusMsg: array[T_FrameType] of array[0..29] of char =
(  ' . . . . Idle . . . . ',
   ' >>>>>>>> Token <<<<<<<<<<<',
   ' --Data--Data--Data--Data--Data',
   ' C#0%$%$%TS%E%$%T#(1%0%N%',
   ' ***** Solicit Successor *****',
   ' ! ! ! ! Set Successor ! ! ! ! !',
   ' ~~~~~ Request Token ~~~~~',
   ' ~~~~~ Claim Token ~~~~~',
   ' xxxx Resolve Contention xxxx',
   ' ~~~~~~~~~ ACK ~~~~~~~~~',
   ' \ \ \ \ \ NAK \ \ \ \ \');

{------------------------------------------------------------------
(NodeStatusMsg is an array containing the status to display for each 
status value.)
{------------------------------------------------------------------

NodeStatusMsg: array[T_Status] of array[0..9] of char =
('Active',
'Token',
'Inactive',
'Passive',
'Down');

{---------------------------------------------------------------}
{NodeActionMsg is an array containing the action message to display
for each action value.}
{---------------------------------------------------------------}
NodeActionMsg: array[T_Action] of array[0..9] of char =
('Listen',
'Pass Token',
'Poll',
'Send Data',
'Receive',
'Initialize',
'Add',
'Delete',
'Crash',
'Wait');

{---------------------------------------------------------------}
{NodeStepMsg is an array containing step message to display for each
step value.}
{---------------------------------------------------------------}
NodeStepMsg: array[T_Step] of array[0..11] of char =
('Listen',
'Get Token',
'Put Token',
'Solicit Succ',
'Set Successr',
'Resolve Cont',
'Send Data',
'Receive',
'Claim Token',
'Confirm',
'Wait',
'ACK',
'NAK');

var
{---------------------------------------------------------------}
{--------------- Global Variables ---------------}
{---------------------------------------------------------------}
implementation

{******************************************************************************************
 xxxxxxxx  TSercterCollection's Method Implementations  xxxxxxxx
 xxxxxxxx  **********************************************************************************} }

function TSortedAddressCollection.KeyOf(Item: pointer): pointer;
begin
  KeyOf := @PTSimObject(Item)^.Address;
end; {TSortedAddressCollection}

{******************************************************************************************
 xxxxxxxx  The compare function for the collection sort function.  xxxxxxxx
 xxxxxxxx  **********************************************************************************} }

function TSortedAddressCollection.Compare(Key1, Key2: pointer): integer;
begin
  Compare := StrIComp(PChar(Key1), PChar(Key2));
end; {TSortedAddressCollection.Compare}

var
  index: integer;

end. {unit TokenGlobals}