DANSE: Instructions for Writing User Protocols

Version 0.14 – 21th March 2012
Dave Pearce dajp1@ohm.york.ac.uk

This document contains the information required to write user-defined protocols for the DANSE simulator.

The simulator has been written so that the user protocols can be written in a language that is almost standard C – there are just a few small exceptions. However, the simulator itself is written in the more powerful language C# and user protocols can also be written in C# for those who would like to take the time to learn this language. This guide describes the API (Application Programming Interface) for both languages.

If there's a function or feature that you would like but is not currently available, please ask the author, it's quite easy to modify the simulator to add new functions.

This is a new module, and the simulator has not been exhaustively tested, so it's likely that there are some bugs left in it. You're advised to check if things are working as you expect (where possible), and report anything odd. Spotting bugs, investigating, documenting and reporting them accurately will be rewarded.

All comments and feedback, please address to the author.

Contents

1 Introduction

This document contains the information required to write user-defined protocols for the DANSE simulator.

The protocol stack used by the simulator is shown in Table 1 below:

Table 1 - Protocol Stack for DANSE

Application Layer	Sources and receives information packets
Transport Layer	End-to-end flow and error control
Network Layer	Routeing
Logical-Link Layer	Hop-by-hop flow and error control
Multiple-Access Control (MAC) Layer	Multiple access
Physical Layer	Transmission and reception of packets, collisions and interference

The application layer and physical layer are fixed and not readily accessible to users, they are part of the simulator itself. Protocols for the other four layers can be written and optimised by the users.

Chapter 2 of this document describes the five essential functions that must be provided by each protocol (receiving packets from above and below, initialisation, shutdown and callback) and describes the library functions provided for sending packets to the layers above and below.

Chapter 3 describes useful functions available at all protocol layers, including functions to call timer interrupts for some time in the future; print messages to the log files and the console; set and retrieve cross-layer information associated with packets; change the transmit and receive channels; and to report on the success (or otherwise) of transmitting packet to the layer above.

Chapter 4 describes the functions that any protocol can use to find out about the node on which it is running, including the node number, location and remaining energy.

Chapter 5 describes the functions for making packets, viewing their contents, and adding, removing and viewing headers.

Every protocol layer has a packet queue available to store packets, and chapter 6 describes the functions required for adding packets and metadata and retrieving packets from this queue. You don't have to use the built-in queue, but it might be useful, as the rest of the simulator makes use of this queue for determining whether the network capacity is being exceeded, and showing the number of waiting packets in the user interface.

The network layer has to maintain a routeing table, and there are some built-in functions that can help organise, search and store routeing tables. You don't have to use these functions, but using them allows the GUI to draw the routes on the map of the nodes. These functions are described in chapter 7.

The media access control (MAC) layer has a very close relationship with the physical layer, and there are some special functions that the MAC protocols can use to find out about the state of the channel, and also to turn off the transmitters and receivers to save power. These functions are described in chapter 8.

Chapter 9 describes the functions that can be used to store tree-structures, sometimes used at the MAC and logical-link layers.

Chapters 10 and 11 contain some information about the language used in the simulator, the differences between the version of C available to write protocols and standard C, and some hints for good programming style.

Chapter 12 provides examples of minimum-functionality protocols for each of the four layers, and chapter 13 contains some information about adding protocols to the built-in protocols.

2 Essential Functions that Protocols Must Contain

Essential functions fall into three categories: management functions for setting-up and shutting-down protocols; routines for accepting timer or event interrupts; and functions for receiving packets from the protocols at the layers above and below.

2.1 Management Functions

All layers must provide an `Initialise()` routine and a `Shutdown()` routine.

2.1.1 The Initialise Routine

The `Initialise()` routine is called when the simulator is first started. The initialise routines for user-defined protocols take either four or seven parameters, all doubles, for example:

```
void Initialise(double A, double B, double C, double D)
{
}

void Initialise(double A, double B, double C, double D,
                                       double E, double F, double G)
{
}
```

(The multiple access and routeing layers take seven parameters, the logical link and transport layers take only four.)

These parameters are taken from the sliders on the Setup Protocols tab, they can take any value from zero to ten; any attempt to set them to values lower than zero or greater than ten will result in an error. The purpose of these variables is defined by the user routines: nothing in the simulator itself uses any of these values. This allows for multiple runs in batch mode with slightly different parameters for user protocols without re-compiling the program.

2.1.2 The Shutdown Routine

The `Shutdown()` routine is called when the simulator ends a run. It takes no parameters. It is provided to allow the protocol to free up any dynamically assigned memory. If you are not using any dynamically assigned memory this function can be left empty (however it does need to be here, since the simulator will attempt to call it at the end of every simulation run).

The shutdown routine can also be used to print out the final state of the simulation, for example a routeing protocol implemented at the Network layer might include code to print out the final routeing table in the shutdown routine.

2.2 Interrupt and Timer Function

All protocols must provide a `Callback()` routine which can be used by the protocol when it wants something to happen after a defined period of time. It takes one required parameter (an integer) and one optional parameter (a packet) both of which can be set when the callback is requested.

```
void Callback(int A, cPacket packet = null)
{
}
```

The integer parameter can help distinguish between callback events when a single layer requests more than one type of callback event. For example, at the multiple-access layer timeouts might be indicated by a callback of type zero, waking up from sleep might be indicated by a callback of type one, etc.

Callbacks can be used as timeout interrupts, or to control the transmission of packets at regular intervals. You can request either relative or absolute callbacks: requesting a relative callback takes as a parameter the delay until the callback is required; requesting an absolute callback takes as a parameter the absolute local time when the callback should happen. For example, to request a callback of type three in five seconds time, a user routine would use:

```
RequestRelativeCallback(5.0, 3);
```

and to request a callback at a local time of 40.5 seconds, a user routine would use:

```
RequestAbsoluteCallback(40.5, 3);
```

It is also possible to cancel callback requests using the `CancelCallbacks()` function.

Any attempt to request a callback in the past will result in an error. It is important to note that callbacks may not happen at exactly the requested time: this is a result of the simulator using independent clocks for each node. For more details see section 3.1.

2.3 Functions Specific to Particular Layers

As well as the initialise, shutdown and callback routines which are common to all layers, there are some functions specific to particular layers, in particular those which send packets to the layers above and below.

2.3.1 The Transport Layer

The transport layer provides end-to-end flow and error control. A basic best-effort and reliable protocol come pre-defined with the simulator, anything else must be written as a user routine.

Packets are delivered to the transport layer from the application layer together with a destination address and a source and destination port number; all are numbers between zero and 254.

Packets are delivered to the transport layer from the application layer with associated information about the source and destination ports and the final destination address. Packets are delivered to the transport layer from the network layer below together with information about their original source, and a copy of the transport layer header.

Packets should be sent to the network layer below with additional information about the destination address for the packet, and to the application layer above with information about the source address, and the destination and source ports.

The functions to be completed for user routines (in addition to the usual `Initialise()`, `Callback()` and `Shutdown()` functions) are:

```
void PacketArrivesFromNetworkLayer(cPacket packet, int sourceNode)
```

```
void PacketArrivesFromApplicationLayer(cPacket packet, int destinationNode, int
destinationPort, int sourcePort)
```

For sending packets to the application and network layers, the following functions should be used:

2.3.1.1 SendPacketToApplicationLayer

Function prototype	`void SendPacketToApplicationLayer(cPacket packet, int sourceNode, int destinationPort, int sourcePort);`
Example	`SendPacketToApplicationLayer(packet, sourceNode, destPort, srcPort);`

This routine can only be called by the Transport layer. This routine sends the given packet to the application layer of the node. Note that the destination port and source port must be provided; these are usually taken from the transport layer header.

2.3.1.2 SendPacketToNetworkLayer

Function prototype	`void SendPacketToNetworkLayer(cPacket packet, int destination);`
Example	`SendPacketToNetworkLayer(packet, node);`

This routine can only be called by the transport or logical-link layers. This routine sends the given packet to the network layer of the node; the second parameter should be set to the final destination of the packet.

(`cPacket` is a class that contains information identifying a packet inside the simulator. A user protocol should never have to look inside this class, but it is often necessary to pass this parameter to another function to specify which packet to use. C programmers can treat it like a structure.)

2.3.2 The Network Layer

The network layer provides the routeing function. A basic direct routeing scheme (everything is sent directly to the end-user in one hop) and flooding scheme (all packets received directed to someone else are sent back out with the hop count reduced by one, until the hop count reaches zero) are provided; there is also a simple Bellman-Ford implementation and a simple on-Demand routeing protocol. Anything more intelligent must be written as a user routine.

Packets are delivered to the network layer from the transport layer with the final destination as an additional parameter. Packets are delivered to the network layer from the logical-link layer with the number of the node from which the packet was received.

The functions to be completed for user routines (in addition to the usual `Initialise()`, `Callback()` and `Shutdown()` functions) are:

```
void PacketArrivesFromLogicalLinkLayer(cPacket packet, int receivedFrom)
```

```
void PacketArrivesFromTransportLayer(cPacket packet, int destination)
```

and packets are sent to the higher and lower layers using the functions:

2.3.2.1 SendPacketToTransportLayer

Function prototype	`void SendPacketToTransportLayer(cPacket packet);`
Example	`SendPacketToTransportLayer(packet);`

This routine can only be called by the Network layer. This routine sends the given packet up to the transport layer of the node.

2.3.2.2 SendPacketToLogicalLinkLayer

Function prototype	`void SendPacketToLogicalLinkLayer(cPacket packet, int hopAddress);`
Example	`SendPacketToLogicalLinkLayer(packet, BROADCAST);`

This routine can only be called by the Network layer or the MAC layer. This routine sends the given packet to the logical-link layer of the node. In the case of the Network layer, the `hopAddress` parameter tells the logical-link layer where to send the packet. If the packet is to be broadcast, hopAddress should be set to the constant BROADCAST.

2.3.2.3 GetMaximumHopCount

Function prototype	`int GetMaximumHopCount();`
Example	`maxHops = GetMaximumHopCount();`

The specification of a maximum hop count is such a common requirement in routeing protocols that the slider specifying the hop count for the internal routeing protocols is not disabled for user routeing protocols, and can be accessed by the user routines using this function. It is an integer, and can be set from one to ten.

2.3.3 The Logical-Link Layer

The logical-link layer provides hop-by-hop flow and error control. A basic best-effort and reliable protocol come pre-defined with the simulator, anything else must be written as a user routine.

Packets are delivered to the logical-link layer from the network layer together with a next hop address (which can be set to BROADCAST for broadcast packets).

Packets are delivered to the logical-link layer from the network layer with associated information about the next-hop destination. Packets are delivered to the logical-link layer from the MAC layer with information about their source, and the received power.

Packets should be sent to the network layer with information about the destination address, and to the application layer with the destination and source port.

The functions to be completed for user routines (in addition to the usual `Initialise()`, `Callback()` and `Shutdown()` functions) are:

```
void PacketArrivesFromNetworkLayer(cPacket packet, int nextHop)

void PacketArrivesFromMACLayer(cPacket packet, int lastHop)
```

and packets are sent to the higher and lower layers using the functions:

2.3.3.1 SendPacketToNetworkLayer

Function prototype	`void SendPacketToNetworkLayer(cPacket packet, int node);`
Example	`SendPacketToNetworkLayer(packet, node);`

This routine can only be called by the transport or logical-link layers. This routine sends the given packet to the network layer of the node. In the case of packets being sent from the logical-link layer, the parameter should be set to the last hop: the node where the packet was received from (this information is provided by the MAC layer).

2.3.3.2 SendPacketToMACLayer

Function prototype	`void SendPacketToMACLayer(cPacket packet, int NextHop);`
Example	`SendPacketToMACLayer(packet, BROADCAST);`

This routine can only be called by the logical-link layer. This routine sends the given packet to the MAC layer of the node. The address for the MAC layer to send the packet to is given in the second parameter of this function, for broadcast packets it should be set to the constant value BROADCAST.

2.3.4 The Multiple-Access Layer

The multiple access layer decides when to send packets, and with what power. Note that the next hop destination and source are stored also within the MAC header, rather than the logical-link header, since this is how things are usually done on the Internet.

Packets are delivered to the multiple-access layer from the logical-link layer together with a next hop address (which can be set to BROADCAST for broadcast packets). Packets are delivered to the multiple-access layer from the physical layer with a reading of the received power.

There is also an option to add pre-emption. With pre-emption on, if a new packet interrupts an existing packet, but the new packet is received so strongly that the signal to interference ratio would allow the new packet to be received, then the new packet is received. If pre-emption is switched off, then an interrupting packet will never be received, no matter how loud it is.

The functions that are required to be implemented for a user protocol (in addition to the usual `Initialise()`, `Callback()` and `Shutdown()` functions) are:

```
void PacketArrivesFromLogicalLinkLayer(cPacket packet, int nextHop)
void PacketArrivesFromPhysicalLayer(cPacket packet, double rxPower)
```

and packets are sent to the higher and lower layers using the functions:

2.3.4.1 SendPacketToLogicalLinkLayer

Function prototype	`void SendPacketToLogicalLinkLayer(cPacket packet, int lastHop);`
Example	`SendPacketToLogicalLinkLayer(packet, lastHop);`

This routine can only be called by the Network layer or the MAC layer. This routine sends the given packet to the logical-link layer of the node.

In the case of the MAC layer, the second parameter should be set to the last hop: where the packet was received from (if this information is known to the MAC layer).

2.3.4.2 SendPacketToPhysicalLayer

Function prototypes	`void SendPacketToPhysicalLayer(cPacket packet);`
	
	`void SendPacketToPhysicalLayer(cPacket packet, double Power);`
	
	`void SendPacketToPhysicalLayer(cPacket packet, double Power, int bitRate);`

Example	`SendPacketToPhysicalLayer(packet, -30.0);`

This routine can only be called by the MAC layer. It sends the packet down to the physical layer for transmission. The transmit power (in dBm) can be specified for each packet; if it is not specified then the default value (the value set using the slider on the Setup Simulation tab) is used. The bit rate can also be specified for each packet, again if this is not set the default value (set using the slider) is used.

3 Constants, Variables and Miscellaneous Functions

There are a few constants provided that you can just use in your functions. These include:

```
BROADCAST           // Set destination to this to broadcast packets
NOWHERE             // Used to indicate an unknown node
CHANNELS            // The number of channels in the system
```

3.1 Some Useful Functions

Useful functions that all user routines at all layers can call:

3.1.1 RequestRelativeCallback

Function prototype	`void RequestRelativeCallback(double secondsLater, int A, cPacket packet = null);`
Example	`RequestRelativeCallback(10.0, 3);`

Registers a request for an interrupt after some specified delay. After a request, the `Callback()` function will be called the given number of seconds in the future (provided this is less than the maximum length of the simulation), and passed the value of the integer parameter `A` (and the cPacket packet if one is specified, this parameter is optional).

Note that callback events may not happen exactly at the time specified: this is a result of each node having its own clock, all of which can be running at a slightly different speed. Events are stored in the simulator in terms of the global clock to ensure that events are carried out in the right order. There can be slight rounding errors in the conversion between these clocks and this can result in callbacks occasionally happening one or two nanoseconds early or late.

3.1.2 RequestAbsoluteCallback

Function prototype	`void RequestAbsoluteCallback(double callbackTime, int A, cPacket packet = null);`
Example	`RequestAbsoluteCallback(10.0, 3);`

Registers a request for a specified time in the future. After a request, the `Callback` function will be called at the time specified, provided this time is in the future, and is less than the maximum length of the simulation. The time should be specified in seconds. When the callback function is called, it will be passed the value of the parameter `A` (and the cPacket packet, if one is specified).

Any attempt to queue an event at a negative time or before the current time will result in a warning, and the event will not be added to the queue.

Note that callback events may not happen exactly at the time specified: this is a result of each node having its own clock, all of which can be running at a slightly different speed. Events are stored in the simulator in terms of the global clock to ensure that events are carried out in the right order.

There can be slight rounding errors in the conversion between these clocks and this can result in callbacks occasionally happening one or two nanoseconds early or late.

3.1.3 CancelCallbacks

Function prototype	`void CancelCallbacks(int A = -1);`
Example	`CancelCallbacks(3);`

Cancels all requests of a specific type, or if the input parameter is omitted, cancels all callback requests for the current protocol. (In large simulations this can slow down the simulator, as the entire event queue has to be searched for any matching events. In such cases if there are not many callbacks to be cancelled, it might be faster to let the callback events happen, but do nothing when they do.)

3.1.4 printf and printfToLog

Function prototype	`void printfToLog(string format, parameters);`
	`void printf(string format, parameters);`

Examples	`printfToLog(“Hello World from node %d”, 23);`
	`printfToLog(PrintRouteingTable());`
	
	`printf(“Hello World from node %d”, 23);`

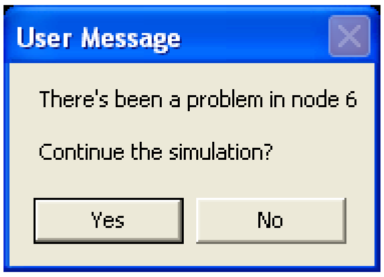
`printf()` prints to the console, and always works no matter what layers are currently being logged. The console is shown on the both the front tab and the outputs tab, and can be saved for later analysis. Warning: using a lot of calls to `printf()` in the simulation can slow the simulator down.

`printfToLog()` formats a message and adds it to the log file (if the current node and protocol layer are being logged). This has the advantage over `printf()` in that the time and node number are automatically added to the message, and it doesn’t slow the simulator down as much.

3.1.5 printfToMessageBox

Function prototype	`void printfToMessageBox(string format, parameters);`
Examples	`printfToMessageBox(“There’s been a problem in node %d”, 6);`

Formats a message and presents it to the user in a message box, pausing the simulation. The user is then asked whether to continue with the simulation or not (see the example in the figure below). Useful for spotting errors, and enabling the simulation to be stopped if a certain error condition is reached.


User Message

There's been a problem in node 6

Continue the simulation?

Figure 1 - Result from printfToMessageBox()

3.1.6 SetCrossLayer and GetCrossLayer

Function prototypes	`SetCrossLayer(cPacket packet, Object data);`
	`Object GetCrossLayer(cPacket packet);`

Example	`SetCrossLayer(packet, TxPowerFromMAC);`
	`TxPowerFromMAC = (double)GetCrossLayer(packet);`

In some cases, it might be useful for information available to one layer to be transmitted to another layer in the protocol stack.

In the simulator, this can be done by setting the cross-layer information associated with a packet in one layer, and reading it in another layer. For example, the transport layer may want to tell the MAC layer that a certain packet is a re-transmission, so that the MAC layer can give it priority.

Note that this cross-layer information does not work across a transmission: any information added to a packet at the transmitting node will be lost when the packet is transmitted, and is not available to the receiver. Also note that the cross-layer information is treated internally as an Object, which means it can be of any type (for example a double, a Boolean, an integer or even a structure or class) however the protocol layer receiving the information has to know what to expect, since it has to cast the cross-layer information into the correct type. Getting this wrong and attempting to cast it to an incompatible type will cause a run-time exception.

3.1.7 SetTransmitChannel, GetTransmitChannel, SetReceiveChannel and GetReceiveChannel

Function prototypes	`Boolean SetTransmitChannel(int newChannel);`
	`Boolean SetReceiveChannel(int newChannel);`
	`int GetTransmitChannel();`
	`int GetReceiveChannel();`

Examples	`SetTransmitChannel(2);`
	`rxChan = GetRecieveChannel();`

DANSE can model scenarios which use more than one channel (different radio frequency), though the default is that all users use the same channel zero. In the case where more than one channel is being used, the nodes have the choice of which channel to transmit and receive frames on. These functions set or get the current receive and transmit channels, and can be called by any protocol layer.

If an attempt is made to change the receive channel while a packet is being received or to change the transmit channel while a packet is being transmitted, a warning message will appear and the requested change will not be done. To avoid these messages and ensure correct behaviour, it is advised to check on the state of the receiver and/or transmitter before attempting to change the channel. This can be done using the `AmIReceiving()` and `AmITransmitting()` functions, which return a Boolean value of true if the physical layer is currently receiving or transmitting a frame respectively. For example, you could use something like:

```
if (AmIReceiving() == false) SetReceiveChannel(2);
```

Alternatively, if the multiple-access layer wants to ensure that the channel is changed as soon as possible, it can request a callback for when the channel becomes idle (using the `RequestCallbackWhenTransmitterIdle()` function) and change the channel when that callback occurs; however note that this can only be done by the multiple-access layer.

3.1.8 ReportOutcome

Function prototype	`void ReportOutcome(cPacket packet, int A)`
Example	`ReportOutcome(lostPacket, 10);`

Some protocols also require the higher layer protocol to be told if the transmission of a packet is successful or not. If called, this function will cause a callback event at the layer above, and pass as parameters into that callback the integer A, as well as the packet that has been successfully or unsuccessfully transmitted.

There are no built-in constant values for A which mean "success" or "failure"; the two layers involved have to agree these between themselves.

Note that this function is not available at the transport layer; the built-in application layer does not require or respond to report outcome events.

3.2 The Interaction Query System

At any point in the simulator, the simulation can be stopped, and the user protocols interrogated to provide information (and potentially modify the performance of the protocols in mid-simulation). This is done using the “Interact” tab on the TabControl on the front (Main) page of the simulator.

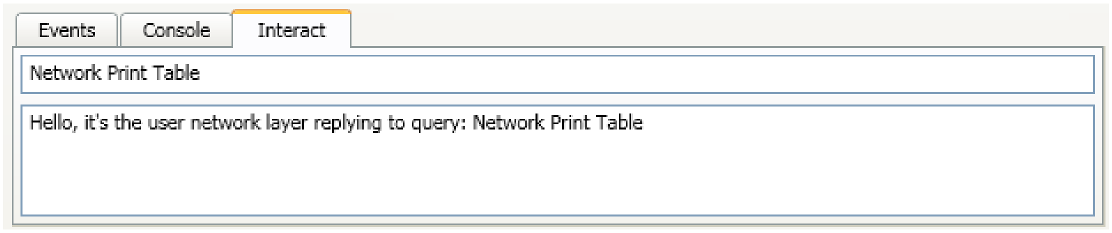
When asked, the simulator will send a query to the relevant user protocol, and print on the main page the reply. Any query starting with an “N” will be sent to the network layer protocol of the relevant node; any query starting with a “T” will be sent to the transport layer of the relevant node; any query starting with an “M” or “L” will be sent to the multiple access or logical link layer of the relevant node respectively.

The relevant node is either the currently selected node or one defined in the second field of the query; for example a query of “T” would send the query to the transport layer of the currently selected node, whereas “Network 23” would send the query to the network layer of node 23.

The handler routine in the user protocol should look something like this:

```
internal override string AskQuery(string s)
{
    // Do anything required here, including change state variables:
    return "Hello, it’s the user network layer replying to query: " + s;
}
```

The input string `s` is whatever the user types in the Query box on the main screen. The reply is printed in the response block below, for example:



Network Print Table

Hello, it's the user network layer replying to query: Network Print Table

Figure 2 – Example of Query and Response from Simulator’s Front Main Page

If no user function is provided in the user protocol the simulator will use internal default response routines. Currently these print the current routeing table for the network layer, and the contents of the transmit queue for the transport, logical link and multiple access layers.

3.2.1 Pausing the Simulator

Often, it can be useful to program the simulator to pause when a certain event occurs. This is done using the `Pause()` function.

Function prototype	`void Pause()`
Example	`if (thing > 23) Pause();`

Calling the `Pause()` function has exactly the same effect as pressing the Pause button on the main screen. Note that this cannot be done in batch file mode.

4 Finding Out About Yourself

The protocols running in the node might want to know a bit about the node where they are. There are several functions they can use to find out about their host node:

4.1 GetMyNumber

Function prototype	`int GetMyNumber();`
Example	`thisNode = GetMyNumber();`

Returns the integer number (address) of the current node, where the processing is taking place. This will be an integer between zero and the number of nodes minus one (for example if there are ten nodes in the simulation, this routine will return an integer between zero and nine).

4.2 GetMyXCoordinate / GetMyYCoordinate

Function prototype	`double GetMyXCoordinate();`
Example	`X = GetMyXCoordinate();`

Returns the X and Y coordinates of where the node happens to be at the time. This assumes the nodes have knowledge of where they are (perhaps there is some sort of GPS receiver in the nodes). Note that this is not true of many sensor and ad-hoc networks scenarios.

4.3 GetMyEnergyLeft

Function prototype	`double GetMyEnergyLeft();`
Example	`energyLeft = GetMyEnergyLeft();`

Returns the amount of charge left in the battery of this node. Currently, nodes start with a charge of 10,000. If the energy mode is set to "Add Up" then this can be negative since the nodes will not stop at a charge of zero; however if the energy mode is set to "Countdown" the returned value will countdown to zero and then stop, at which point the node will stop transmitting or receiving frames.

4.4 GetMyTime

Function prototype	`double GetMyTime();`
Example	`theTime = GetMyTime();`

Returns the time, as reported by the local clock in the node, measured in seconds. Note that when individual clocks are used, the different nodes will have clocks that run at slightly different rates, so the time may not be the same in all the nodes. Note that time is stored internally as a long measured in nanoseconds, so one nanosecond is the minimum time resolution in DANSE.

5 Headers and Packets

Packets can be generated at any layer in the protocol stack, and usually a header is added to the data at every layer the packet passes through on its way to be transmitted. This simulator assumes a strict protocol layers model, so that each layer is only able to see the layer added by peer layers (for example, the network layer can only look at and understand headers added by other network layers).

The structures representing the header contents added by each layer have suggested names:

Layer	Header
Transport	`TransportHeader`
Network	`NetworkHeader`
Logical Link	`LogicalLinkHeader`
Multiple Access	`MACHeader`

for example[1]:

```
[Serializable] struct NetworkHeader
{
    internal byte sourceNode;
    internal byte destinationNode;
}
```

You can add anything else you like into this structure, but note the structure must have a fixed length. Also note that the more information in the header, the longer the packet becomes, and the more energy it will take to transmit. There is a maximum header length, currently[2] this is set to 1024; any attempt to use headers longer than this will cause an error. (This is intended to prevent a header being longer than the maximum sensible length of a packet. If this causes a problem for your protocol then you're probably doing something a bit silly.)

You don't have to specify the header using a structure, although it is a useful way to represent them. If you want a header of a variable length (for example a packet might collect the numbers of the nodes that it passes through as it goes) then you may need to add a series of bytes to the end of the header, a single structure won't provide the flexibility you need[3].

Headers are passed to and from the simulator using a pointer to an array of bytes. As long as you set the pointer to the start of the memory where the header is stored, and tell the `AddHeader()` routine how long the header is, you can create the headers "by hand" without using a structure.

[1] Note the `[Serializable]` tag. This is necessary to allow the simulator to make copies of the packet when it is transmitted to the other nodes in the simulation.

[2] Set in Globals.cs as Globals.MaxHeaderSize

[3] Note to C# programmers: there is a better way to do this using a class as the header rather than a structure; classes can be variable sizes. However this requires careful calculation of the size of the class, and the built-in functions for adding headers to packets cannot be used in this case; you have to use the packet.AddHeader() method directly. Please ask for more details if you're finding this to be a problem.

Note that all elements in the header structures must be prefixed by the keyword `internal`. The `internal` keyword is an extension to the C syntax used in this system to define which other functions can access the parameters in the structure.

5.1 Header and Packet Functions

Some functions are provided that the user routine can call to manipulate headers:

5.1.1 AddHeader

Function prototype	`void AddHeader(cPacket packet, byte *header, int headerSize);`

Example	`HeaderType newHeader;`
	`newHeader.sourceNode = 23;`
	`newHeader.destinationNode = 24;`
	`AddHeader(packet, (byte *)&newHeader, sizeof(HeaderType));`

Adds a header to the specified packet. Used when a packet arrives from the layer above, and a header needs to be added to the packet before it is sent down to the layer below. The routine takes a pointer to the start of the header in memory, and the size of the header to be added.

C# programmers might find the alternate technique of using the `.AddHeader()` method in the `cPacket` class easier to use. This method takes as parameters a `cHeader` class, which consists of two elements: a header object and the length of that object in bytes. For example:

```
LayerHeader newHeader = new LayerHeader(... parameters for constructor );
packet.AddHeader((Object)newHeader, newHeader.GetSize());
```

In this case, the programmer can specify the length of the header, which makes adding lists of nodes much easier to do – however DANSE will not automatically check that the user routines have calculated the correct length for the header.

5.1.2 RemoveHeader

Function prototype	`byte *RemoveHeader(cPacket packet);`
Example	`myHeader = (MyHeaderType *)RemoveHeader(packet);`

Removes the outermost header in a packet, and returns a pointer to the header, expressed as a pointer to series of bytes. This should only be done when the outermost header is one that the current layer understands, for example the Network layer should only call this function when the outermost header is a `NetworkHeader`.

Again, C# programmers might prefer to use the `.RemoveHeader()` method in the `cPacket` class directly. For example:

```
myHeader = (MyHeaderType *)packet.RemoveHeader();
```

5.1.3 ViewHeader

Function prototype	`byte *ViewHeader(cPacket packet);`
Example	`myHeader = (MyHeaderType *)ViewHeader(packet);`

Returns a pointer to the outermost header, but does not remove it from the packet. Again, this should only be done when the outermost header is one that the current layer understands, for example the Network layer should only call this function when the outermost header is a `NetworkHeader`. Note that the function returns a pointer to a byte, this will usually be immediately cast to a pointer of whatever type the header is.

Once again, C# programmers might prefer to use the `.ViewHeader()` method in the `cPacket` class directly. For example:

```
myHeader = (MyHeaderType *)packet.ViewHeader();
```

5.1.4 MakePacket

Function prototype	`cPacket MakePacket(byte *content, int size);`
Example	`newPacket = MakePacket(buffer, 100);`

Generates a new packet with a payload size of `size` bytes, the content of the packet is copied starting from the memory address `content`. It returns a structure[4] `cPacket`, which contains internal information used by the simulator to represent a packet; you should never have to look inside `cPacket` yourself.

[4] Strictly speaking it's a class not a structure, but as long as you never try to look inside it they behave the same way.

C# programmers can make a packet that contains any type of content, but once again it's the programmer's responsibility to calculate and specify the size of the content. In this case, a new packet can be generated using the constructor:

```
internal cPacket(int size, int sourceNode, Object content, int ThisSystemNumber)
```

where `size` is the size (in bytes) of the new packet, `sourceNode` is the node that is generating the packet, `content` is any content, and `ThisSystemNumber` is reserved and should be set to zero. For example:

```
cPacket newPacket = new cPacket(size, node.GetNumber(), NewContent, 0);
```

5.1.5 GetPacketContents

Function prototype	`byte *GetPacketContents(cPacket packet);`

Example	`byte *Contents;`
	`Contents = GetPacketContents(packet);`

Returns a pointer to the contents of a packet, in bytes. This function should only be called by the layer that created the packet, after all headers have been removed. Any attempt to use the function when there are still headers left attached to the packet will result in an error. Typical uses of this function might include retrieving routeing information from a routeing packet sent by the network layer.

C# programmers can use the `.ViewContents()` method in the `cPacket` class, which will return an item of type object, which requires to be cast to whatever type of object the packet contains. For example:

```
myStuff stuff = (myStuff)packet.ViewContents();
```

5.1.6 GetPacketSize

Function prototype	`int GetPacketSize(cPacket packet);`
Example	`sizeOfPacket = GetPacketSize(packet);`

Returns the current size of the packet in bytes, including all the headers. It is not good practice to use this function, since lower layers might add padding to a packet but not remove it, or packets may get truncated due to errors. If a protocol layer needs to know the size of a packet, it is better to put the length of the packet in a field in the header of that layer.

C# programmers should use the slightly faster `.GetPacketSize()` method in the `cPacket` class, for example:

```
int packetSize = packet.GetPacketSize();
```

5.1.7 CopyPacket

Function prototype	`cPacket CopyPacket(cPacket packet);`
Example	`newPacket = CopyPacket(oldPacket);`

This function makes an exact copy of the given packet, including all the headers. Used when writing reliable layers, when a copy of the packet needs to be taken to be stored in case a re-transmission is required.

5.2 Tagging Packets

Some of the user interface routines print up details of packets when the cursor is placed on the packet; this happens on the main screen, and the physical layer activity screen. For application layer packets, the packet number is displayed, but for packets generated by other layers the layer itself is responsible for adding a description to the packet, otherwise nothing will be displayed.

It's not essential to use this feature, but it can help in debugging protocols.

5.2.1 SetPacketTag

Function prototype	`void SetPacketTag(cPacket packet, char *Tag);`
Example	`SetPacketTag(thisPacket, “ACK from MAC layer”);`

This routine should only be called by the layer that creates the packet. (It can be called by any layer, and the current tag of the packet would then be overwritten; but this is not recommended practice.)

6 Packet Stores

Every protocol layer has a packet store that it can use to store packets. This is especially useful for reliable layers, but is also useful for MAC layers that have to store packets before they get the chance to transmit them.

The functions to use the protocol store are detailed in this section. (You can write your own packet store functions, but using the built-in ones allows the simulator to display the number of packets waiting at each protocol layer in the information box on the main screen, and to check that the number of packets in the queues is not steadily increasing, indicating network saturation.)

It's likely that at some layers some additional information will require to be stored in the queue alongside the packets, and the routines provide a means to attach a structure to every packet in the queue; this is called the metadata. This structure can be defined by the user, and can contain anything you want. After a packet has been added to the queue, the contents of the metadata can be modified by using the `ReplaceMetadataInQueue()` function.

6.1.1 GetQueueLength

Function prototype	`int GetQueueLength();`
Example	`QueueLength = GetQueueLength();`

Returns the number of packets currently held in the queue.

6.1.2 PutPacketInQueue

Function prototypes	`void PutPacketInQueue(cPacket packet);`
	`void PutPacketInQueue(cPacket packet, Object metadata)`

Examples	`PutPacketInQueue(packet);`

	`struct queuemetadata`
	`{`
	`        internal double time_added;`
	`        internal int attempts;`
	`}`
	
	`queuemetadata meta;`
	`meta.time_added = GetMyTime();`
	`meta.attempts = 0;`
	`PutPacketInQueue(packet, meta);`

Adds the supplied packet to the queue. Packets are automatically time-stamped when they enter the queue, so the oldest packet can be identified by the `GetOldestPacket()` routine described below, however the example above adds a timestamp to the metadata for the use of the user protocol.

(Metadata is additional data about the packet that is stored in the queue with the packet. It could be a single number, or a structure / class.)

Note to C-programmers: the Object in the function declaration allows the routine to accept any type of structure[5], or just a single value, perhaps a single time or an integer number of attempts. However this means that the `GetMetadataInQueue()` function returns an object, which must be cast into the expected type before being used, see the `GetMetadataInQueue()` function for more information.

6.1.3 FindPacketInQueue

Function prototype	`int FindPacketInQueue(cPacket packet);`
Example	`int indexInQueue = FindPacketInQueue(thisPacket);`

Returns the location of a specified packet in the queue, or returns -1 if the packet cannot be found in the queue.

6.1.4 GetMetadataInQueue

Function prototype	`Object GetMetadataInQueue(int index);`

Example	`queuemetadata thismeta`
	`       = (queuemetadata)GetMetadataInQueue(0);`

Gets the metadata from a specified position in the queue. Note that the routine returns an object, so it must be cast to the correct type before use.

If the input parameter `Index` is outside the current range of the queue, it will return `null`, which can be a bit of a problem, since structures are not nullable types (in other words they cannot be given the null value). This is likely to cause a NullReferenceUnhandled run-time error.

It is recommended that any attempt to use this function is immediately preceded by a `FindPacketInQueue()` function call to return the index number, or a check that the index number is within the current range using `GetQueueLength()`.

6.1.5 ReplaceMetadataInQueue

Function prototype	`void ReplaceMetadataInQueue(int index, Object newStuff);`
Example	`ReplaceMetadataInQueue(0, newMeta);`

[5] Note to C# programmers: The metadata Object can be a class as well as a structure, but if you're new to C# you might forget that classes are reference types while structures are value types, so getting the metadata using GetMetadataInQueue() for a class returns a pointer if metadata is a class, but a copy of the structure if metadata is a structure.

Replaces the metadata associated with a packet in the queue. The new metadata does not have to be the same type as the original metadata (although it might be rather confusing to have two different types of structure as metadata in the same queue). This can be used when you want to leave a packet in the queue, but update the metadata structure[6], for example when attempting to transmit the packets for a second time at a reliable layer.

6.1.6 GetPacketInQueue

Function prototype	`cPacket GetPacketInQueue(int index);`
Example	`cPacket firstInQueue = GetPacketInQueue(0);`

Gets the packet at a specified position in the queue. If the input parameter `Index` is outside the range of the queue, it will return `null`. Note that this does not remove the packet from the queue; if you want to remove the packet, use `RemovePacketFromQueue()`.

6.1.7 ClearQueue

Function prototype	`void ClearQueue();`
Example	`ClearQueue();`

Empties the queue.

6.1.8 RemovePacketFromQueue

Function prototype	`cPacket RemovePacketFromQueue(int index);`
Example	`FirstPacket = RemovePacketFromQueue(0);`

Removes the packet at the specified position from the queue. The function returns the packet removed, or `null` if no packet is at that index.

6.1.9 GetIndexOfOldestPacket

Function prototype	`int GetIndexOfOldestPacket();`
Example	`int whereIsOldest = GetIndexOfOldestPacket();`

Returns the index (the position in the queue) of the packet that has been in the queue for the longest time. This packet is probably at position zero in the queue.

[6] Another reminder for C# programmers: if the metadata in the queue is a class then this function is not required, since you can just do a GetMetadataInQueue() function call, get a pointer to the class, and directly change the metadata elements in the queue.

6.1.10 GetOldestPacket

Function prototype	`cPacket GetOldestPacket();`
Example	`cPacket nextToSend = GetOldestPacket();`

This function returns the oldest packet in the queue. Note that this returns packet that was put into the queue the longest time ago, not necessarily the one that should be transmitted (or re-transmitted) first. It does not remove the packet from the queue.

7 Routeing Tables at the Network Layer

There is some built-in support for storing routeing tables. You don't have to use the internal routeing table, you can write your own, but unless you want to do something rather unusual it might be easier to use the one provided.

The other advantage to using the routeing table provided is that the front-panel checkbox "Show Routes" can then be used to visualise the routes easily: it gets its information from the internal routeing tables.

7.1 Routeing Table Elements

The built-in routeing tables always stores four elements for each entry in the tables:

- Destination: an integer specifying the final destination
- Cost: a double specifying the cost of the route
- ValidTime: a long with the time when the entry was put in the routeing table
- Parameter: an object with no defined meaning, the user can interpret this in any way.

In addition, the routeing table will store one of the following:

- NextHop: an integer specifying where packets to this destination should be sent next
- Route: a list of integers specifying the complete route to the destination, starting with the source and ending with the destination.

NextHop is used for protocols that store only the next hop to the destination, Route is used for protocols that store the entire route to the destination.

7.2 Routeing Table User-Accessible Functions

Routines are provided to add or update and delete entries in the routeing table, and query the table for the NextHop or Route to a specified node. The functions are described below:

7.2.1 AddRouteingTableEntry

Function prototypes	`void AddRouteingTableEntry(int destination, int nextHop, double cost)`
	`void AddRouteingTableEntry(int destination, int nextHop, double cost)`
	`void AddRouteingTableEntry(int destination, byte *route, int routeLength, double cost)`
	`void AddRouteingTableEntry(int destination, byte *route, int routeLength, double cost, Object parameter)`

Example	`AddRouteingTableEntry(dest, hop, 1.0, 0.0);`

These routines add a new entry into the routeing table, setting the entry for packets addressed to "destination" to be sent via node "nextHop" or via route "route", and assigning a cost of "cost" to the route. If the final parameter "parameter" is present, this is added to the routeing table entry as well.

If a route is specified, the function also requires to know how long the route is.

If an entry to the defined destination already exists, it is always overwritten, whether or not the cost is greater or less than the cost of the existing route.

The routeing table element parameter "validTime" is set to the current time when the entry is made/updated.

C# programmers can use the alternate form:

```
void AddRouteingTableEntry(int destination, List<int> route, double cost);
```
which is easier than using byte arrays.

7.2.2 DeleteRouteingTableEntry

Function prototype	`Boolean DeleteRouteingTableEntry(int destination)`
Example	`DeleteRouteingTableEntry(12);`

Removes an entry from the routeing table if one is found to the specified destination. Returns true if a routeing entry was found, or false if no matching routeing entry was found.

7.2.3 GetNumberOfRoutes

Function prototype	`int GetNumberOfRoutes ()`
Example	`CurrentSize = GetNumberOfRoutes();`

Returns the current number of routeing elements in the routeing table. Useful if you're trying to produce a packet containing the table to sent to other nodes (e.g. in the Bellman-Ford algorithm).

7.2.4 LookupCost, LookupNextHop, LookupValidTime and LookupParameter

Function prototypes	`double LookupCost(int destination)`
	`int LookupNextHop(int destination)`
	`long LookupValidTime(int destination)`
	`Object LookupParameter(int destination)`

Example	`CostToA = LookupCost(A);`

These routeing interrogate the routeing table. They look for an entry to the specified destination, and return the cost, nextHop, validTime and Parameter of that entry (if one is found).

If no entry to the specified destination is found, lookupCost returns INFINITY, lookupNextHop returns NOWHERE, lookupValidTime returns zero and lookupParameter returns null.

Note that the parameter is returned as an Object, so will need to be cast to whatever variable type it was stored as. This can be a class, a structure, or any variable type such as an integer or double; however if you're storing the parameter as a non-nullable type (such as a structure, integer or double) and then attempt to lookup a parameter that does not exist, you will get a run-time error[7].

7.2.5 LookupRoute

Function prototypes	`byte *LookupRoute(int destination, int *Hops)`

Example	`int hopsInRoute = 0;`
	`byte *routeToA = LookupCost(A, &hopsInRoute);`

This function takes the destination and the address of an array of integers, which the function fills in with the number of hops in the route. The function returns a pointer to a list of bytes containing the stored route. Note that the array must be long enough to store the maximum length route.

C# programmers can use the alternate form:

```
List<int> LookUpRoute(int destination)
```

which is safer, and returns a list of integers containing the node numbers on the route.

7.2.6 GetDestinationByIndex, GetCostByIndex, GetNextHopByIndex, GetRouteByIndex, GetValidTimeByIndex and GetParameterByIndex

Function prototypes	`int GetDestinationByIndex(int index)`
	`double GetCostByIndex (int index)`
	`int GetNextHopByIndex (int index)`
	`List<int> GetRouteByIndex (int index)`
	`long GetValidTimeByIndex (int index)`
	`Object GetParameterByIndex (int index)`

Example	`ThisDestination = GetDestinationByIndex (loop);`

These routines return the parameters of the routeing table entry at a certain place in the routeing table. Useful for building up a copy of all or part of the routeing table in order to put the data in a packet and send it to someone, you could for example use a for loop from zero to `GetNumberOfRoutes() – 1` to create a packet with information about all the routes.

`GetRouteByIndex()` returns a list of integers in the route. This function is for C# programmers

[7] If in doubt, look up the cost first, and only look up the parameter if that returns a value which is not INFINITY.

only; there is no equivalent for C programmers.

Note that `GetParameterByIndex()` returns an Object, which will need to be cast to whatever variable type it was stored as. This can be a class, a structure, or any variable type such as an integer or double; however if it is a non-nullable type (for example a structure, integer or double) and this function is called with a value of index for which no entry exists in the table, then this function will return null, and any attempt to cast null to a non-nullable type will throw a run-time error.

7.2.7 ClearRoutingTable

Function prototype	`void ClearRouteingTable()`
Example	`ClearRouteingTable();`

Clears all entries in the routeing table, except the entry to the node itself. This is added in by default with a cost of zero and a parameter which is a double of value zero.

7.2.8 PrintRouteingTable

Function prototypes	`string PrintRouteingTable ()`
Example	`A = PrintRouteingTable ();`

Formats the routeing table of the current node in a neat style, and returns a string with the description of the routeing table in it. Useful for `OutToLog()` events.

8 MAC Layer Specific Functions

Quite often, a MAC layer needs to know some details about the physical layer: the most common case is when the MAC is trying to sense the received signal to know whether the receiver is detecting another transmission or not.

There are several built-in functions that can help with this.

8.1.1 IsTransmitterBusy

Function prototype	`Boolean IsTransmitterBusy()`
Example	`if (IsTransmitterBusy() == false) {...`

This function returns true if the transmitter is currently transmitting a frame, and false otherwise. (It should not be necessary, the MAC layer should already know whether the transmitter is busy or not, but sometimes it might be useful to check.)

8.1.2 IsChannelQuiet

Function prototype	`Boolean IsChannelQuiet()`
Example	`if (IsChannelQuiet()) { ...`

This function returns true if the receiver is detecting that the channel is quiet. In this context, "quiet" means that the ratio of the received signal power to noise at the receiver is less than the current value of the detection threshold, and the receiver is awake and currently listening.

If the receiver is currently asleep, the function will always return false, no matter what is going on.

8.1.3 HowEmptyIsChannel

Function prototype	`double HowEmptyIsChannel(double period)`
Example	`used = HowEmptyIsChannel(1.0)`

This function returns the proportion of the recent past during which the channel has been sensed and found to be quiet; the total time considered is given by the input parameter period, so that in the example the function returns the proportion of time that the channel has been free in the last second. The returned value should be between 0.0 (always busy) and 1.0 (always quiet).

Note that the node must be active and listening for the channel to count as quiet; if during the last period the node was asleep then this time would not count as quiet time.

8.1.4 GetDetectionSNIR / SetDetectionSNIR

Function prototype	`double GetDetectionSNIR ()`
	`void SetDetectionSNIR(double newSNIR);`

Example	`double thisDetect = GetDetectionSNIR();`
	 `SetDetectionSNIR(10.0);`

These functions set and return the current value of the ratio of the received signal power to noise that the nodes use to detect a transmission. At initialisation of the protocol, this value is set by the value of the Setup Simulation tab, but protocols can modify this value during the simulation itself.

It's not necessary to use these functions at all; you could just use the next function and determine whether the received signal level is likely to cause problems yourself.

8.1.5 GetCurrentReceivePower

Function prototype	`double GetCurrentReceivePower()`
Example	`double incoming = GetCurrentReceivePower();`

This function returns the total amount of signal power that the receiver is detecting, including the noise.

8.1.6 GetCurrentTransmitPower

Function prototype	`double GetCurrentTransmitPower()`
Example	`double outgoing = GetCurrentTransmitPower();`

This function returns the total amount of signal power that the transmitting is currently transmitting. If the node is not currently transmitting, then this function will return zero.

8.1.7 RequestCallbackWhenChannelClear

Function prototypes	`void RequestCallbackWhenChannelClear (int x)`
	 `void RequestCallbackWhenChannelClear (int x, double cleartime)`

Example	`RequestCallbackWhenChannelClear(2);`
	 `RequestCallbackWhenChannelClear(2, 0.01);`

This function will register a request from the MAC layer that a callback be generated when the channel is next sensed as clear. The second (optional) parameter is the minimum time for which the channel must be clear before the callback is generated in seconds.

If the parameter `cleartime` is set to zero (or not set) and the channel is already clear, this will generate a callback immediately; if the node is currently asleep, this will be when the node wakes up (provided the channel is clear at that time). If the parameter `cleartime` is not set to zero, then the node will wait for at least this time from the time when the function is called before making the callback.

The parameter x is the parameter returned as an input parameter when the callback function in the MAC protocol occurs.

For example, the second example requests a callback (with a parameter of two) at least 10 ms in the future (if the channel is clear for all that time) or the next time that the channel remains clear for 10 ms.

8.1.8 RequestCallbackWhenTransmitterIdle

Function prototypes	`void RequestCallbackWhenTransmitterIdle (int x)`
	`void RequestCallbackWhenTransmitterIdle (int x, double cleartime)`

Examples	`RequestCallbackWhenTransmitterIdle(3);`
	`RequestCallbackWhenTransmitterIdle(3, 0.01);`

If the parameter `cleartime` is set to zero (or not set) and the transmitter is not currently transmitting a frame, this will generate a callback immediately. If the parameter `cleartime` is not set to zero, then the node will wait for at least this time from the time when the function is called before making the callback. For example, the second example requests a callback (with a parameter of two) at least 10 ms in the future (if the transmitter is off for all that time) or 10 ms after the transmitter finishes transmitting the current frame (provided it doesn't start transmitting another one within the 10 ms period).

The parameter x is the parameter returned as an input parameter when the callback function in the MAC protocol occurs.

Note that this function is not strictly necessary, the MAC layer should know when the transmitter will stop sending the current frame, since the MAC layer knows how long the frames are, when the transmission started, and the current bit rate.

Also please note that there’s a common mistake made when using this function. Writing lines of the form:

```
SendPacketToPhysicalLayer(toGo);
RequestCallbackWhenTransmitterIdle(3);
```

will not produce a callback after packet toGo has been sent. It will produce a callback immediately, as sending the packet to the physical layer does not immediately result in the physical layer starting to transmit the packet: that happens a short time later. At the time the `RequestCallbackWhenTransmitterIdle()` call is made, the transmitter is still idle.

If you want to send a packet and get a callback when the packet has finished, use something like this:

```
SendPacketToPhysicalLayer(toGo);
RequestCallbackWhenTransmitterIdle(3, 10e-9);
```

(I often define a very short time like 10e-9 as a constant called something like `oneJiffy` and then use this whenever I want a very short period of time.)

8.1.9 GoToSleep and WakeUp

Function prototypes	`GoToSleep ()`
	`WakeUp()`

Example	`GoToSleep();`
	`WakeUp();`

These functions can be used to save power.

Calling `GoToSleep()` puts the node in a "sleep" mode, in which the transmitter and receiver are turned off. If any packet arrives at the physical layer when the node is asleep, it is not received, and the `PacketArrivesFromPhysicalLayer()` function is never called. Similarly, if any packet is passed down to the physical layer using `SendPacketToPhysicalLayer()` while the node is asleep, the physical layer will not transmit the packet.

Whenever a call to `GoToSleep()` is made, it is vitally important that a function call such as `RequestAbsoluteCallback()` or `RequestRelativeCallback()` is also made, so that the node is able to wake up at some point in the future. Otherwise, the node would be turned off for ever. (It's not like the `sleep()` function provided in some operating systems, the node doesn't automatically wake up some time in the future.)

9 Clusters and Trees at the MAC and Logical-Link Layers

Some logical-link and MAC layers operate by grouping nodes into clusters with a local controller, or into trees, which are not necessarily the same structures used by the routeing protocols. For these protocols, it can be helpful if the structure at the MAC and logical-link layers is visible.

DANSE provides a way to do this, with the "Show Parents" checkbox on the main screen. Check this checkbox, and dark cyan lines will appear between each node and its designated parent for the MAC layer structure, and/or dark khaki lines appear to show the logical-link layer structure.

To set or find the parent of any node, the following functions can be called by any MAC or logical-link layer protocol:

9.1 SetParent

Function prototype	`void SetParent (int parent)`
Example	`SetParent(3); void`

The default value of the parent for each node is NOWHERE. To reset the parent of a node, it can be set to be equal to the global constant NOWHERE using the same routine.

9.2 GetParent

Function prototype	`int GetParent()`
Example	`int myParent = GetParent();`

Returns the current value of the parent node for any node.

Note that both of these functions can only be called from the MAC or LLC layers, and the MAC parent is not the same as the LLC parent, they are stored separately.

10 C and C#

The simulator is written in C#, which shares a lot of syntax with C, but not quite all of it. There are a few differences that C-programmers should be aware of before trying to write protocols:

- Don't use include files: they don't work in C#. It would be better to put all your functions in the relevant source files (cTransport.cs, cNetwork.cs, cLogicalLink.cs and cMAC.cs). If you really want to put some of your functions in a different source file it is possible, please ask for details.

  If you find the file sizes become so large that it's difficult to find things, try enclosing parts of the code using regions, for example:

  ```
  #region The Callback Routine
  void Callback (int A, cPacket packet = null)
  {
      ... a lot of lines of code ...
  }
  #endregion
  ```

  This isn't really part of the language, it's part of the text editor. It just means you can hide those parts of the code you're not working on at the moment.

- Don't modify anything at the top or bottom of the source file. It's there for the C# compiler to tie the functions in the user source files into the rest of the simulator correctly.

- Structures: in both C and C#, a structure is defined by lines of the form:

  ```
  struct XYZ {}
  ```

  Then in C, if you want to create an instance of this structure, you do it like this:

  ```
  struct XYZ newStruct;
  ```

  whereas in C#, it is done like this:

  ```
  XYZ newStruct;
  ```

  In other words, you don't need to add the keyword `struct` when you want to create a new structure in C#, the compiler knows that XYZ is a structure. This makes `typedef` rather redundant.

- You can't use `typedef` in C#.

- If you're declaring a structure that will represent a header to be attached to a packet, it needs to add the `[Serializable]` tag to the start of the declaration[8], for example:

[8] This allows the simulator to make copies of the packet more easily, which the physical layer has to do: the packets that arrive at different receivers are all different copies of the transmitted packet.

```
[Serializable] struct LogicalLinkHeader
```

- You can't use `#define` in C#. If you want to assign a number to a constant that never changes, do it using the `const` keyword like this:

```
const int THINGY = 23;
```

- When declaring a structure, you need to insert the keyword `internal` before all variables that are part of the structure, for example:

```
struct NetworkHeader
{
      internal byte sourceNode;
      internal byte destinationNode;
}
```

- Don't use fixed size arrays (there should be no need to; the protocols should be written to cope with any size of network). If you want to use an array, reserve the memory with a call to `malloc()`.

- An awkward one: the variable types used for 8-bit numbers have different names in the two languages:

C	**C#**
char	sbyte
uchar	byte

(There is a variable type in C# called `char` used to store characters, but it's a 16-bit number. Watch out for this one, it might catch you out.)

Apart from that, you should be able to write pretty much standard C. In particular standard library functions like `malloc()`, `free()`, `rand()`, `srand()`, `floor()`, `pow()`, `log()`, `printf()` and so on should all just work. If you find something that doesn't work, please let me know.

10.1 Why C#?

You might be wondering why the simulator was written in C#, and not, for example, C++ or JAVA.

The main reason not to write in JAVA was that JAVA does not support pointers or structures, so it would be impossible to write code for the protocols that looked like standard C and have it compile within a JAVA environment. It would also prevent the use of the .NET framework and Windows Presentation Foundation (WPF) graphics libraries, which are the standard graphics libraries for the Windows operating system.

C is a much more useful language for communications engineers than JAVA, and using C# provides a better opportunity to gain more familiarity with C.

C++ would have been a possible choice, and it would have had the advantage that the user protocols could have been written with fewer exceptions from standard C, however C++ does not provide many of the most useful and friendly features of JAVA or C# (for example it doesn't have a garbage collector helping to simplify the memory management, standard and easy-to-use XML and serialisation libraries or the same level of support for functional programming). I didn't want to write in C++ as it's easier to make mistakes and harder to debug, the program would have taken longer to write, and there would most likely have been more bugs left in the code.

11 A Note about Style

Code is much easier to read if it is formatted consistently and commented carefully. There are numerous style guides for how to write good code, the important thing is not so much which convention you choose, but how carefully and consistently you stick to it.

For example, the code in this simulator is written using the following conventions (at least I try to follow these):

- Don't use tabs, they are dangerous: different text editors and printers interpret them in different ways and you can find a printed version of your code looks terrible. Instead of tabs, use a series of spaces (four is recommended).
- Try and avoid the use of the `/* */` comment structure. It's far too useful for commenting out large sections of code when debugging, and you can't do this if there is a `*/` somewhere in the block of code being commented out.
- Variables are written in `camelCase`, functions and structure types are written in `TitleCase`, and constants in `CAPITALS`. Hungarian notation (prefixing the name of the variable with an indication of what sort of variable it is, for example using `intVariable` or `iVariable` to indicate an integer variable) is avoided (although I do still use this for objects on the GUI, as it makes them easier to find using the search functions and auto-complete functions in Microsoft Visual Studio).
- Add lots of comments, so it's easy for other people, (including yourself in a few years time) to work out what the code is trying to do, and how the code does it.
- Keep lines short (a maximum of 80 characters per line is suggested), it makes the code much more readable when it is printed out. This means that comments are usually put above the lines they refer to, not as in-line comments after the line (except for very short comments).
- Don't use "magic numbers" hidden in the code. If you need to use a constant number, give it a sensible name, and define it as a constant at the top of the file where it's easy to find.

12 Examples of Minimum Implementations of User Functions

This section contains a bare minimum implementation of user layer protocols at each of the layer.

12.1 Minimum Implementation of Transport Layer

The only things a transport layer has to do is put the destination and source port numbers into a header, so that the receiving node knows which application layer protocol to pass the data in the arriving packet to. When a packet arrives, the transport layer header is removed, and the source and destination ports are passed up to the application layer.

The following minimum implementation does only this.

```
[Serializable] struct TransportHeader
{
    // The only necessary things are the destination and source
    // port numbers.
    internal byte destinationPort;
    internal byte sourcePort;
}

void PacketArrivesFromNetworkLayer(cPacket packet, int sourceNode)
{
    // Get a pointer to the TransportHeader and cast to the type:
    TransportHeader* thisHeader = (TransportHeader*)RemoveHeader(packet);

    // Then send the packet up to the application layer:
    SendPacketToApplicationLayer(packet, sourceNode,
       thisHeader->destinationPort, thisHeader->sourcePort);
}

void PacketArrivesFromApplicationLayer(cPacket packet, int destinationNode, int
destinationPort, int sourcePort)
{
    // Generate a new TransportHeader:
    TransportHeader MyHeader;
    MyHeader.destinationPort = (byte)destinationPort;
    MyHeader.sourcePort = (byte)sourcePort;
    // Add the header to the packet,
    AddHeader(packet, (byte*)&MyHeader, sizeof(TransportHeader));
    // Send the packet down to the network layer:
    SendPacketToNetworkLayer(packet, destinationNode);
}

void Callback(int A, cPacket packet = null)
{
    // No callbacks ever requested, so nothing to do.
}

void Initialise(double A, double B, double C)
{
    // Nothing to initialise.
}

void Shutdown()
{
    // No dynamically assigned memory to free up.
}
```

12.2 Minimum Implementation of Network Layer

This minimum implementation of a network layer does no routeing; it assumes that the transmission is always done with enough power so that the final destination can hear the transmission from the original source. In this case packets are never forwarded by a node to another node, they always travel directly from the source to the destination.

It is possible, however, that a node will send a packet to itself, so the network layer should spot this, and send the packet straight back up to the transport layer, instead of passing it down to the logical-link layer. The following minimum implementation does this.

The only things a minimum implementation of the network layer header has to contain are the destination and source addresses.

```
[Serializable] struct NetworkHeader
{
    internal byte sourceNode;
    internal byte destinationNode;
}

void PacketArrivesFromLogicalLinkLayer(cPacket packet, int receivedFrom)
{
    // Get a pointer to the NetworkHeader and cast to the structure type:
    NetworkHeader* thisHeader = (NetworkHeader*)RemoveHeader(packet);

    // If the packet is addressed to me, get the source node and send
    // the packet up to the transport layer:
    if (thisHeader->destinationNode == GetMyNumber())
    {
        int sourceNode = thisHeader->sourceNode;
        SendPacketToTransportLayer(packet, sourceNode);
    }
    else
    {
        // Otherwise ignore the packet.  Don't do anything.
    }
}

void PacketArrivesFromTransportLayer(cPacket packet, int destination)
{
    if (destination == GetMyNumber())
    {
       // If packet is addressed to me, send it back up to the transport layer:
        SendPacketToTransportLayer(packet, GetMyNumber());
    }
    else
    {
        // Otherwise it must be for someone else, so generate a new networkHeader,
        NetworkHeader MyHeader;
        MyHeader.destinationNode = (byte)destination;
        MyHeader.sourceNode = (byte)GetMyNumber();
        // Add the header to the packet,
        AddHeader(packet, (byte*)&MyHeader, sizeof(NetworkHeader));
        // Set the next hop to the final destination, and send the packet
        // down to the logical-link layer:
        int nextHop = destination;
        SendPacketToLogicalLinkLayer(packet, nextHop);
    }
}

void Callback(int A, cPacket packet = null)
{
    // No callbacks ever requested, so nothing to do.
}

void Initialise(double A, double B, double C, double D, double E)
{
    // Nothing to initialise.
}

void Shutdown()
{
    // No dynamically assigned memory to free up.
}
```

12.3 Minimum Implementation of Logical-Link Layer

The minimum possible logical-link layer does nothing, just passes packet from the network layer straight down to the multiple-access (MAC) layer, and passes packets from the MAC layer straight up to the network layer.

```
void PacketArrivesFromMACLayer(cPacket packet, int lastHop)
{
    SendPacketToNetworkLayer(packet, lastHop);
}

void PacketArrivesFromNetworkLayer(cPacket packet, int nextHop)
{
    SendPacketToMACLayer(packet, nextHop);
}

void Callback(int A, cPacket packet = null)
{
    // No callbacks ever requested, so nothing to do.
}

void Initialise(double A, double B, double C)
{
    // Nothing to initialise.
}

void Shutdown()
{
    // No dynamically assigned memory to free up.
}
```

12.4 Minimum Implementation of Multiple-Access Layer

A minimum implementation of the multiple-access layer protocol implements the ALOHA access protocol: as soon as a packet arrives from the logical-link layer, give it to the physical layer to start transmitting. The header added here does have to add the transmitting node and the next hop destination, in case the logical-link layer above is reliable and wants to send an acknowledgement.

```
[Serializable] struct MACHeader
```

```
{
    // Just contains the destination (next-hop)
    // and source (current node) addresses:
    internal byte destinationNode;
    internal byte sourceNode;
}

void PacketArrivesFromPhysicalLayer(cPacket packet, double rxPower)
{
    // Get the header from the packet:
    MACHeader *header = (MACHeader *)RemoveHeader(packet);
    // Find out where it came from:
    int sourceNode = header->sourceNode;
    // Pass straight up to the logical-link layer:
    SendPacketToLogicalLinkLayer(packet, sourceNode);
}

void PacketArrivesFromLogicalLinkLayer(cPacket packet, int nextHop)
{
    // Generate a MAC-layer header for this packet:
    MACHeader newHeader;
    newHeader.destinationNode = (byte)nextHop;
    newHeader.sourceNode = (byte)GetMyNumber();
    // Add this header to the packet:
    AddHeader(packet, (byte *)&newHeader, sizeof(MACHeader));
    // Send packet down to the physical layer:
    SendPacketToPhysicalLayer(packet);
}

void Callback(int A, cPacket packet)
{
    // No callbacks ever requested.
}

void Initialise(double A, double B, double C, double D, double E)
{
    // Nothing to do.
}

void Shutdown()
{
    // No dynamically assigned memory to free up.
}
```

13 Adding Protocols to the Simulator

DANSE comes with a few “built-in” protocols. It is possible to add new protocols to these built-in protocols. This section provides instructions on how to do this.

There are four main steps:

- Converting the existing protocol from C to C# (only necessary if the protocol was written in C, and even then not entirely necessary, but usually a good idea);
- Setting up the controls for the protocol (user protocols are restricted to three or five linear sliders called A, B, C, D and E that vary from 0 to 10);
- Writing the constructor with the delegate functions;
- Registering the protocol with the parent protocol class, the Globals class and the cNode class.

13.1 Converting Protocols from C to C#

There are two main advantages to doing this: speed of execution (many of the wrapper functions can be bypassed), and better error checking (particularly with headers).

For example, adding a header using:

```
packet.AddHeader((Object)header, sizeof(HeaderType));
```

rather than using the `AddHeader()` routine allows the simulator to check the type of the header at run-time, so that when a packet is received from the layer below some code like this can be run to alert the user that something has gone wrong with a lower-layer protocol:

```
// Some idiot checking:
if (!(packet.ViewHeader() is HeaderType))
{
    ShowMessageBox("Packet arrived without an appropiate header. \n"
        + "Packet will be deleted.", "User Protocol Error?");
    return;
}
```

It’s also a good idea to change any maths routines to use the C# Math class library directly, this will speed them up slightly.

13.2 Setting up the Protocol Controls

User controls have only linear sliders from zero to ten to provide inputs to the protocol. Built-in protocols can rename these sliders and make them integer, logarithmic or square-root, or replace them with checkboxes. These should be put in a partial Window class at the start of the file, for example:

```
public partial class Window1 : Window
{
    private void NetworkSetupBellmanFordControls()
    {
        Globals.stsNetworkHopCount.Enable(true);
        Globals.stsNetworkParameterOne.NameOnSetup.Content = "Mean Interval";
        Globals.stsNetworkParameterOne.SetNewRange(1, 1000, 10, false, true,
false, false, false);
        Globals.stsNetworkParameterOne.Enable(true);

        Globals.stsNetworkParameterTwo.NameOnSetup.Content = "Stop After";
        Globals.stsNetworkParameterTwo.SetNewRange(1, 1e6, 1e3, true, true, false,
false, false);
        Globals.stsNetworkParameterTwo.Enable(true);
    }
}
```

(This example from the built-in BellmanFord protocol.) This uses the function:

```
internal void SetNewRange(double newMinimum, double newMaximum, double newDefault,
Boolean Ints, Boolean Logs, Boolean Just125, Boolean SqrRoot, Boolean
LogsWithZero)
```

which is provided in the cGUIStuff.cs file; see the comments in that file for more details.

13.3 Writing the Constructor

When a node is instantiated, a set of protocols are setup for it, and an initialising function is required that sets up some delegate functions. The easiest thing to do is copy and rename an existing class at the same layer, and then change the name of the class and the initialisation function, for example at the Network layer, a constructor might look like this:

```
class cDirectRouteing : cNetwork
{
    //////////////////////////////////////////////////////
    // Direct routeing: always send direct to destination.
    //

    internal cDirectRouteing(cNode Here, ref cNode.NetworkDelegates Delegates):
base(Here)
    {
        Delegates.CallWhenCallback =
             new cNode.NetworkCallbackDelegate(this.Callback);
        Delegates.CallWhenInitialise =
             new cNode.NetworkInitialiseDelegate(this.Initialise);
        Delegates.CallWhenPacketArrivesFromLogicalLinkLayer =
             new cNode.NetworkPacketFromBelowDelegate(
             this.PacketArrivesFromLogicalLinkLayer);
        Delegates.CallWhenPacketArrivesFromTransportLayer =
             new cNode.NetworkPacketFromAboveDelegate(
             this.PacketArrivesFromTransportLayer);
        Delegates.CallWhenShutdown =
             new cNode.NetworkShutdownDelegate(this.Shutdown);
    }
```

The names of the delegate functions vary according to the protocol layer concerned, so it's important to copy this from another protocol at the same layer.

13.4 Registering the New Protocol

There are three places in the code where a new 'built-in' protocol has to be registered.

Firstly, in `Globals.cs`, under (for example), the Network Control Box, in the enumerated type for the protocol layer (in this example, the NetworkType). The entry in here will appear in the drop-down box on the set-up screen so it can be selected by the user; it need not be the same name as the class. (Note that any underscores in the enumerated name will be converted to spaces in the combo-box and wherever else the name is displayed.)

Secondly, in `cNode.cs`, in the `Initialise()` function, another entry must be added to the switch statement selecting a class to instantiate for that protocol layer. This ensures that the new protocol can be added to a node's software stack when a node is initialised. Copy the others already there for style.

Thirdly, in `cProtocols.cs`, in the partial `Window1` class for the relevant layer controls, in (for example), the `NetworkTypeCombo_SelectionChanged()` function. Add to the switch statement here a call to setup the slider controls. This ensures that the sliders on the set-up screen are changed when the new protocol is selected using the combo-box.

That's it.