The WIB

Wide Information Bus

for kbmMW v. 2.01b+

Publish/subscribe type of information transfer has become a hot subject lately, and its not without reason.

The great thing about publish/subscribe is that its a loosely coupled event oriented setup compared to the traditional request/response type of setup where clients starts a request and servers reply with a response. Those setups often do not allow for any 'out of order' server transmission to the client, or if it does, it does so in a relatively limited way requiring hard coding each situation for each client.

Each publisher or subscriber of information is connected to a virtual bus which we call the WIB (Wide Information Bus).

Why is it named like that?

'Information Bus' because it is a virtual media of information transportation, and 'Wide' because the types of data transported on it can consist of any of kbmMW's already well known data types. Thus a wide range of data types are available from simple values over objects and arrays to streams which for example could contain video, sound or images, and datasets.

All applications attached to a WIB can act both as servers and clients or rather as publishers and subscribers. Hence we generally don't distinguish between servers and clients when in a typical publish/subscribe setup, but rather about nodes.

A node is simply an application that is able to interact as an information publisher or subscriber, or both and that are attached to the WIB.

Thus any node can be generating or consuming information.

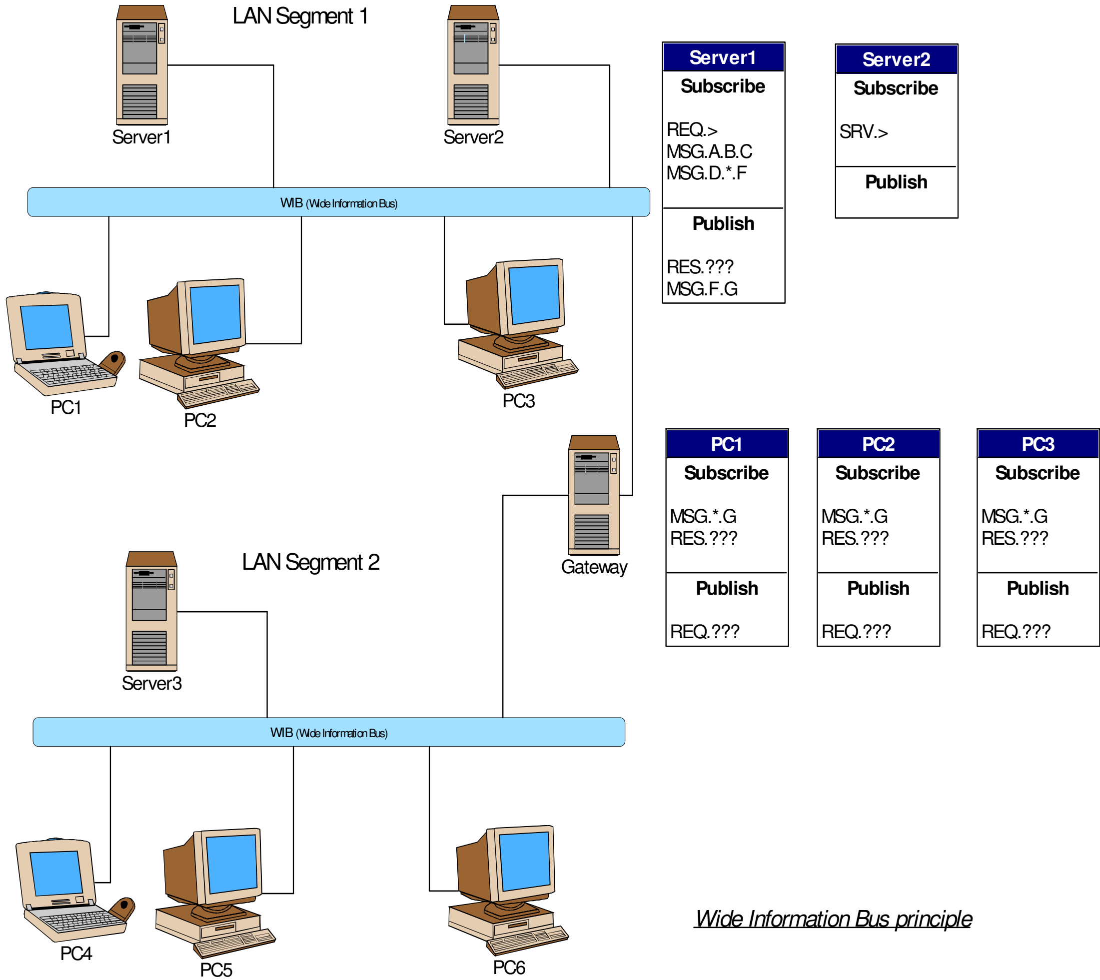
The nodes are connected to a virtual bus which we call the WIB (Wide Information Bus). Why is it named like that? IB because its a virtual media of information transportation, and wide because the types of data transported on it can consist of any of kbmMW's already well known data types. Thus a wide range of data types are available from simple numbers to streams which for example could contain video, sound or images, and datasets.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The WIB can span all from a single physical segment on a local LAN, over multiple segments on an Intranet to the world wide Internet.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

A sample publish/subscribe setup with the WIB spanning two Ethernet LAN's



Wide Information Bus principle

All of the pc's and servers on this image are all nodes in the publish/subscribe world. However some of the nodes also act as kbmMW application servers, and some of the other nodes also act as kbmMW clients.

Its easy to compare publish and subscribe communication with a news agency publishing one or more magazines (e.g. Dr Dobbs Journal (DDJ), ComputerWorld (CW), PCWorld (PCW) etc.) and having some customers who subscribe for one, some or all of the magazines.

The publishers publish each magazine under a specific name, which in the publish/subscribe (from now on called p/s) world would be called the subject. Customers would choose which subjects to subscribe on and each time a publication was ready and sent out, the customer would receive it.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The subject

In the computerized p/s world, the subject is a loosely formatted string build of one or more parts separated by '.'.

The subject is used as an advanced addressing mechanism surpassing simple IP numbers, used by nodes to know if they should receive a message or not published by another node. Since the message in theory flows all over the WIB, all nodes connected to it can choose to receive the message. I.e. one published message can be received by many subscribers.

In practice there are several built in techniques to limit the flow to only nodes who have expressed their wish to receive messages with those specific subjects.

The subject is loosely formatted because the number of parts and the contents of the parts are not defined by the p/s protocol as such except for the first part, and thus can be freely defined by the developer.

The first part generally must be one of the following texts: MSG, REQ, RES, SUB, USB, CAC, THR, SRV
We will first concentrate about MSG. Later in the whitepaper the others will be explained.

Subjects starting with MSG identifies unsolicited asynchronous messages. Using the MSG type subject we will try to build a subject which the newpaper agency can publish DDJ under.

The subject to publish messages around DDJ could be 'MSG.DDJ'. Clients would be able to subscribe for 'MSG.DDJ' and receive all messages related to DDJ.
However what if another newspaper agency have a journal which is named DDJ (Danish Deliveries Journal)? Then there would be a potential conflict if the two news agencies both publish under the same subject. Hence a refinement is needed of the subject. A news agency part should be added.

Lets say the news agency that publish Dr. Dobbs Journal is named CMP Media LLC, and the Danish Deliveries Journal is published by Berlingske, we could add a part which identifies the publisher. Eg. the subject for DrDobbs Journal would be: 'MSG.CMP.DDJ' and for Danish Deliveries would be 'MSG.BERLINGSKE.DDJ'. Notice the hierarchical layout of the subject, and that the subject is not case sensitive (all will be handled as uppercase).

Lets also say that customers doesn’t subscribe forever, but only for specific issues on a month basis. We could of course just subscribe for all DrDobbs Journal issues, and throw out the ones we don’t want after inspecting the magazine itself, but it would be better if we didn’t receive the issues we aren’t interested in. Hence yet a refinement of the subject is needed. We add an issue number: 'MSG.CMP.DDJ.yyyy.nn' where yyyy is the year the issue is published, and nn is a number from 1 to 12 identifying the issue that particular year.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Then customers can choose to subscribe for DrDobbs Journal issue 4 year 2003 by subscribing for the subject: 'MSG.CMP.DDJ.2003.04'.

What about customers who want to subscribe for all of 2003? Well then the client suddenly need to accept anything in the place of the issue part. That's where wildcards come handy.
A customer can subscribe on 'MSG.CMP.DDJ.2003.>' to receive all packages regarding DDJ from CMP in 2003 when they are published.

The **>** means, that the remaining part of the subject is not taken into consideration when kbmMW needs to figure out if a node subscribes for something or not.
If the node want to subscribe for any DDJ, regardless from which newspaper agency, but only for 2003 and issue 12, the node needs to subscribe like this: 'MSG.*.DDJ.2003.12'

The ***** means, that specific part of the subject is not to be taken into consideration. The * has to replace a complete part, and its not allowed to do like this: 'MSG.C*.DDJ'. In other words the * must be considered a placeholder for a complete part.
It's valid to have multiple * in a subscription on a subject. Its however not allowed to have more than one > (and wouldn't be very sensible to have more anyway).

A node can subscribe for multiple subjects at any given time. There is no technical limit to the number of subjects subscribed on and the length of each, but since there may arrive thousand messages/second which kbmMW needs to filter according to the subject, its good practice to keep the number of subscriptions as low as possible (less than 100/node) and to limit the number of subject parts to less than 20.

A special negate operator '!' also exists. Its purpose is for a subscriber to say that it definitely don't want to receive a specific subject.
I.e. the node could subscribe for MSG.*.DDJ.*.* which means it would receive all messages related to DDJ regardless of year or issue. But lets say that the subscriber of some reason definitely do not want anything from year 2001. In this case the negate subject can be useful.

Thus the node would have these two subscriptions:

!MSG.*.DDJ.2001.*
MSG.*.DDJ.*.*

Since the negate subscription is first, it will overrule the next more general subscription.

There is of course no point in adding negate subscriptions for subjects which are not otherwise subscribed for, as the node will never receive messages it does not actively want to receive by setting its subscriptions accordingly.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Its good practice to arrange the subject parts in such way that kbmMW is able to determine if a subject match subscriptions as soon as possible. kbmMW starts from the left when evaluating the subject parts and from the top of the subscription list iterating downwards until match or all subscriptions have been evaluated.

kbmMW use a very fast hashing based algorithm for detecting subjects which a subscription has been made for but nonetheless its good practice to follow these rules of thumbs.

The subject parts should generally only contain US/UK letters and digits. Using local charactersets can have side effects if the messages are received by computers not setup the same way.

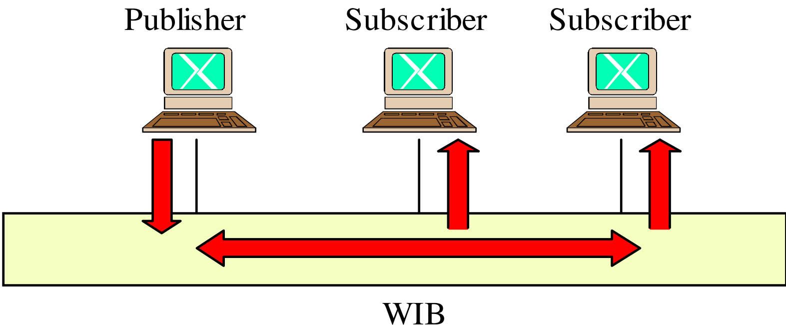
Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Messaging topologies

kbmMW bundles support for UDP broadcast based messaging, TCPIP hub/spoke and UDP/TCPIP peer to peer messaging. Other types of messaging topologies can be supported by kbmMW as well. One could for example create a messaging email transport utilizing standard emails for moving messages.

Broadcast messaging

Broadcast messages have the advantage that multiple nodes are able to subscribe for and receive the same messages from other nodes without the need to resend the message to each receiving node. Thus only one single message travels on the physical network.



This simplifies lots of situations, and generally lower bandwidth requirements.

Each node have a local list of subscriptions which it uses to choose which of the incoming messages it would like to keep and process. Its voluntary if the node would like to publish its list of subscriptions to other nodes.

Due to the typical configuration of Ethernet routers and switches, broadcasting only works for nodes attached to one single Ethernet segment.

If there are multiple segments, or switches/routers in between the segments, a gateway is needed. The gateway is an application which subscribes for messages on one segment and transport those messages via a point to point transport (typically TCP/IP) to a gateway on the other segment where the messages are republished, hence stretching the WIB over two or more segments.

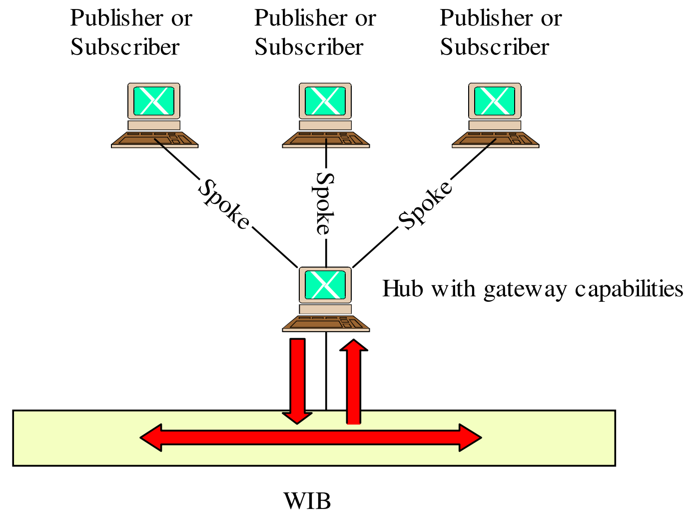
The gateway can easily be created by having a server with a registered service which a client component in the other end can send a traditional message to containing the complete messaging message. The service would then simply publish the received message on the local LAN.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

If a gateway is created its usually a good idea for each node to announce its subscriptions to let the gateway know what to listen for on the other segment. This ensures that only traffic which is really interesting for nodes on both segments will be passing the gateway.

Hub/Spoke messaging

Where the broadcast type of messaging may be impractical over multiple physical Ethernet segments (anyway using UDP as the transport layer), Hub/Spoke messaging often is the answer.



The hub/spoke setup differs from the broadcast setup in that one have to designate a specific node as a hub to which other nodes connect versus the broadcast setup where all nodes were equally attached to the WIB.

The hub itself usually is directly attached to the WIB, thus able to interact with other nodes on the WIB.

Since a hub/spoke setup requires that a message is sent multiple times to multiple subscribing nodes, a hub/spoke setup is not as bandwidth efficient as the traditional broadcast messaging setup. To lessen the bandwidth load, spokes must inform the hub about what subjects they subscribe for. On a broadcast setup, this is optional unless a filtering gateway is in place.

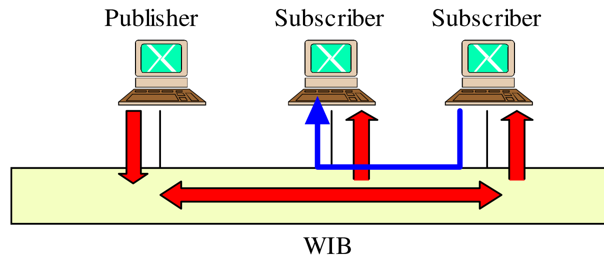
The spoke provides this information by announcing its subscriptions:

```
spoketransport.AnnounceSubscriptions;
```


Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Peer to peer messaging

One can target a message to a specific node, identified by its IP address, by specifying that IP address as the target argument in the SendMessage method.



Targeting a message allows the message to cross Ethernet segments without having a messaging gateway, provided the switch, router or hub allow UDP packages to cross.

Using the peer to peer setup, it's a true peer to peer setup where any node can communicate with any other node using the UDP messaging transport. The node targeted should of course still be subscribing for the subjects that the sender, choose to send.

As usual any messages send to the node, for which the node does not have a matching subscription for, will be ignored.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The WIB node

To create a publishing or subscribing node, all what's needed is to add a **TkbmMWxxxyyyMessagingTransport** where xxx identifies the type of transport and the yyy if its a client or server transport.

Why the distinction of a client or server transport?

There are two reasons:

- One reason is that the messaging transports also support the traditional request/response setups using kbmMW clients and servers. Hence the messaging server transport can be directly connected to a **TkbmMWServer** and operate simultaneously as a true asynchronous messaging transport and as a synchronized request/response transport.
- Another reason is in case of the hub/spoke messaging transport setup. The hub will always be a server transport, and the spoke always a client transport.

Thus if your publishing node also contains a kbmMW application server, you would usually put a server messaging transport on that node.

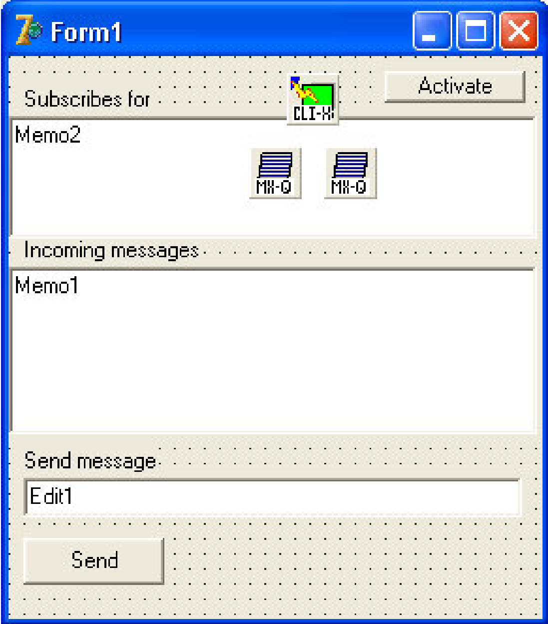
And if you have nodes which act as clients to an application server, and thus operate according to the request/response principle, you would put a client messaging transport on the node.

Its perfectly legal to create a setup of nodes who all operate 100% on the publish/subscribe bases without any requirements for an application server. In that case you don't have to add any **TkbmMWServer** components in any node, and can choose to use just client transports on all nodes, except for those who act as hub nodes.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

A simple UDP broadcast based node

We will make a small sample. Create a new application and create a form looking similarly to this:



The Client transport (CLI-X) component is in this sample a **kbmMWUDPIndyMessagingClientTransport**.

Two other components are placed on the form too. They are message queues which are responsible for buffering up inbound and outbound messages until they can be either handled (inbound) or sent (outbound).

kbmMW bundles one message queue component, the **TkbmMWMemoryMessageQueue** (which store all messages in the queue in memory), but the framework allow for extending **TkbmMWCustomMessageQueue** to for example store messages in files or databases.

The message transport component's **InboundMessageQueue** and **OutboundMessageQueue** must point to each their own message queue. Message queue types can be mixed (when available) if needed.

Further some properties needs to be setup regarding on which port, and network card(s) the transport should listen for messages, and on which port and IP address, the transport should send messages.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Set **ListenIP** to 0.0.0.0 and **ListenPort** to 4000. The Indy UDP Messaging transport allows for listening on multiple ports and IP addresses by defining them in the **ListenBindings** property.

0.0.0.0 means that the transport will listen for messages coming from any IP address.

Then setup where to send outbound messages. Set the **SendIP** to 255.255.255.255 and the **SendPort** to 4000.

255.255.255.255 means that messages are broadcast to all devices listening on the current LAN segment.

That's all the setup that's needed for now.

Now we add some code for the buttons on the form. Double click the Activate button to get to the code of its *OnClick* event.

In it type:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     kbmMWUDPIndyMessagingClientTransport1.Active:=true;
end;
```

If we don't activate the messaging transport, no messages will ever be received, and all messages we try to send will be buffered in the outbound message queue, but never sent.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The ‘*Subscribes for*’ memo will be holding the list of subjects that this client will be listening for. To ensure that changes in the memo is applied to the messaging transport component we will make a simple *OnChange* event handler for the Memo2 component.

Double click the Memo2 component and ensure the *OnChange* event looks similar to this:

```
procedure TForm1.Memo2Change(Sender: TObject);
begin
     kbmMWUDPIndyMessagingClientTransport1.Subscriptions.Assign(Memo2.Lines);
end;
```

This updates the subscription settings each time the memo is changed.

The Subscriptions property operates like a plain TStringList, but in reality much more work is going on in the background. When the subscriptions change, new optimized hashing lists are updated which makes it possible for kbmMW to filter network packets extremely fast, and thus only to let those messages in which the node subscribes for.

Each line in the subscription list is a subscription.

Then create an event handler for when messages that the client subscribes for has been received.

This is done by writing some code in the TkbmMWUDPIndyMessagingClientTransport’s *OnMessage* event. In our small demo all we want to do is to display the subject of the received message in the ‘Incoming messages’ memo:

```
procedure TForm1.kbmMWUDPIndyMessagingClientTransport1Message(
  Sender: TObject;
  const TransportStream: TkbmMWCustomMessageTransportStream;
  const Args: TkbmMWArrayVariant; UserStream: TMemoryStream);
begin
     Memo1.Lines.Add('Message received:'+TransportStream.Subject);
end;
```

We could have been looking on the received client identity too via the TransportStream properties (Username, Password, Data etc.), and we could have used any provided UserStream and arguments (Args which is a variant array) if we wanted to.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Access the arguments like this:

```
var
  i:integer;
  v:variant;
begin
  for i:=o to length(Args)-1 do
  begin
     v:=Args[i];
     //.. do something with the argument.
  end;
end;
```

Of course you can access the variant array in any way you need. kbmMW supports sending all valid variant datatypes as arguments, including single dimension arrays.
If you dont know if v is an array or not, use the VarIsArray function to test.

Eg. `if VarIsArray(v) then...`

In our small sample node, we also want to be able to send a message with a specific subject.
For that purpose we add an event handler to the *OnClick* event of the Send button:

```
procedure TForm1.Button2Click(Sender: TObject);
var
   ci:TkbmMWClientIdentity;
begin
     ci:=TkbmMWClientIdentity.Create;
     try
        ci.Username:='Kim Madsen';
        ci.Password:='somepassword';
        ci.ClientLocation:='In the office';
        kbmMWUDPIndyMessagingClientTransport1.SendMessage(Edit1.Text,
           '',ci,nil,[]);
     finally
        ci.Free;
     end;
end;
```

Notice the on the fly creation of a **TkbmMWClientIdentity** object (which require you to add **kbmMWSecurity** in the uses clause).

The contents of the client identity object is automatically transferred to the receiving end, which can then determine if the client is allowed to do what you try to do or for whatever other purpose.

Also notice the nil argument. Instead of nil, you could have provided a TStream instance containing stream data which you would like to send to all the subscribers of the subject for which this message is sent.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

If an UDP broadcast messaging transport is used then kbmMW will of course send the message under the restrictions of UDP, which means that each packet is only allowed to be maximum 32kb to 64kb depending on the underlying UDP implementation.

This is a difference compared to a TCP based transport, since TCP based transports do not limit the amount of data sent in one TCP message. The underlying network layer will for TCP automatically handle the large message by splitting it up in fragments according to the capabilities of the actual hardware, and reassemble the message on the receiving end.

However kbmMW contains advanced automatic data fragmentation and de-fragmentation for UDP based messaging transports.

What that means is that huge amounts of data can be sent using UDP, simply because kbmMW automatically chops up the data in smaller packets which the OS can handle, and reassembles the packets correctly, on the receiving end.

UDP messaging however have one disadvantage to TCP messaging because TCP based messaging automatically will guarantee packet delivery or detect delivery faults in both the sending and receiving end. UDP do not have that feature.

Since there are no guarantee on delivery of data sent using UDP, there is a risk that the receiving end will miss one packet out of the perhaps thousands send for a large message. kbmMW will on the receiving end detect the missing packet/fragment, and discard the complete message, but the sending end will currently believe it was sent ok. If you need it resent, send a message to the publisher that you want a resend and put some code in the publisher to handle that situation.

Saturating the messaging with huge packets will also decrease the number of messages possible to transfer per second. Using a message size of less than around 1000 bytes (incl. subject and client identity) will easily allow you to transfer a thousand or more messages per second depending on your hardware.

Thus it's generally a good idea to minimize the size of the messages sent unless you know the potential consequences of large messages.

Finally there are a couple of more arguments which can be given, but are not in the current sample. The argument variant array which allow you to send any type of values (including arrays) as arguments to your message, and the **Target** argument which in our sample is an empty string.

If you are using an UDP based messaging transport, setting **Target** to an IP address will direct the message directly to that specific IP address, and will not use broadcast for sending the message. The advantage is that you don't bother other clients and that the message can pass over network segments generally without being stopped by switches or routers. This is also called Peer to Peer messaging.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Add kbmMWGlobal to Unit1's interface uses clause.

Now compiling this sample, you will be able to send messages from one node to multiple other nodes on the same LAN segment.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Letting others know what you subscribe for

At times, it can be beneficial to announce what you subscribe for.

You could have nodes on the net which act as gateways between LAN segments and which would like to know what subjects the nodes on one segment listens for to be able to listen on all the relevant subjects on the other LAN segment and transport those to the first LAN segment where the node are waiting for data.

Announcing is made easy this way:

```
kbmMWUDPIndyMessagingClientTransport1.AnnounceSubscriptions;
```

If a node want to receive information about other nodes announcements, the node must be subscribing for '`SUB.>`' and '`USB.>`' If you only want to receive information about specific nodes subscriptions and unsubscriptions, its possible to add a hashed node id as the next subject part after SUB or USB.

`SUB.>` means that all subscriptions will be received.
`USB.>` means that all unsubscriptions will be received.

The subscription message format is:

```
SUB.anodeID.DELTA.asubject
```
or
```
SUB.anodeID.CLEAR.asubject
```

The subject is automatically unpacked and the parts provided as arguments to the **OnSubscription** and **OnUnsubscription** events of the messaging transport.

anodeID will be a the hashed value of the sending nodes **NodeID** (yet another property of the transport). If **NodeID** is not set by the developer, one will be generated. It's highly recommended to identify a node uniquely before sending messages.

One way to set NodeID to a unique value is to do like this:

```
messagingtransport.NodeID:=kbmMWGenerateUniqueNodeID;
```

Remember to put kbmMWGlobal in the units uses clause.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The **NodeID** is not automatically added to MSG messages, and thus it's your own responsibility to add it if you need it. **NodeID** is however automatic part of all other message types.

In the **OnSubscription** event the hashed **NodeID**, a boolean flag **Delta** and a **subject string** is provided. The Delta flag is true if the subscription is an addition to existing subscriptions in that node. If Delta flag is false, it means that all existing subscriptions should be considered invalid, and the given one is the first new subscription for that specific node.

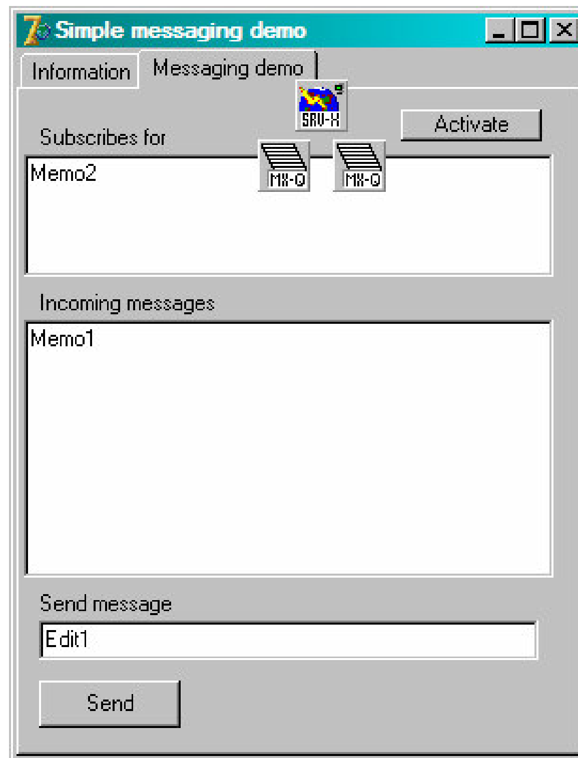
The data for the **OnUnsubscription** event is always a delta. The sending node can notify other nodes that a specific subject is no longer being subscribed for.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

A simple Hub node

This sample will show how to create a simple hub node for the Hub/Spoke messaging topology.

Create a new application with a form looking similar to the form in the 'A simple UDP broadcast node', but this time put a **TkbmMWTCPIPIndyServerMesssagingTransport** on it.



Go through the same steps as with the UDP broadcast node creation, to hook up the message queues and fill in all the events, this time referring to kbmMWTCPIPIndyMessagingServerTransport1 instead of kbmMWUDPIndyMessagingClientTransport1.

The properties of the TCP hub messaging server transport is slightly different to the UDP messaging transport in the sense that ListenIP/ListenPort and SendIP/SendPort does not exist on the hub transport. Instead there is a **Bindings** property in which its important to add at least one binding. In our case set that binding up to listen on 0.0.0.0 port 4000.

You can add more bindings if desired, to let the transport listen on multiple network cards at the same time if you have more than one installed.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Ensure that the *OnClick* event of the Activate button is looking like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     kbmMWTCPIPIndyMessagingServerTransport1.Listen;
end;
```

Since this is a server transport, we have to Listen, rather than activate.

Then we need to setup a couple of new properties on the TkbmMWTCPIPIndyServerMessagingTransport.

Set **AutoRelay** to true. **AutoRelay** controls if inbound messages should be automatically relayed/echoed to spokes who subscribes for messages with the subject of the incoming messages. If its set to false (default), spokes will never see any messages sent by other spokes, but only those messages that the developer specifically choose to send to the spokes via the server transports SendMessage method.

When messages are automatically relayed to other spokes, you can setup conditions for which types of messages should be relayed via the **RelayOptions** and **RelayTypes** properties.

RelayOptions is a set of the following values:
- **mwmroSubscribed** : If specified, then messages for which the hub itself subscribes for are relayed. (default)
- **mwmroUnsubscribed** : If specified then messages for which the hub itself do not subscribe for are relayed. (default)
- **mwmroPeerSubscribe** : If specified, then only messages for which the peer spoke is actually subscribing for is relayed to that spoke. For the server to know what messages the spoke subscribes for, the spoke will have to announce its subscriptions after connecting to the hub and every time the subscriptions are changed on the spoke node. (default)

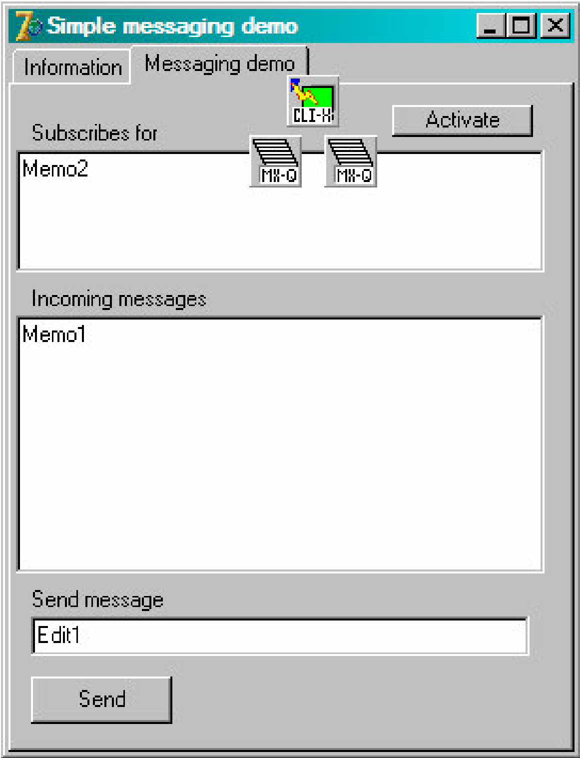
RelayTypes is a set of values specifying which message types will be relayed. Default all types are relayed.

This is all what's needed to create a messaging hub.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

A simple Spoke node

Next step is of course to create a spoke node that can connect to the hub.
Again we use the previous 'A simple UDP broadcast node' sample as a guideline, but this time place a **TkbmMWTCPIPIndyClientMessagingTransport** on it instead of the UDP transport.



Again hook up the message queues to the transport and fill in the code in the events, this time referring to **kbmMWTCPIPIndyClientMessagingTransport1** instead of the UDP transport.

Setup the client transport's **Host** and **Port** properties to point on the hub node.

Ensure that the *OnClick* event of the Activate button looks like this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     kbmMWTCPIPIndyMessagingClientTransport1.Active:=true;
     kbmMWTCPIPIndyMessagingClientTransport1.AnnounceSubscriptions;
end;
```


Copyright 2001-2004 Components4Developers
http://www.components4developers.com

and that the OnChange event of the Memo looks like this:

```
procedure TForm1.Memo2Change(Sender: TObject);
begin
     kbmMWTCPIPIndyMessagingClientTransport1.Subscriptions.Assign(Memo2.Lines);
     kbmMWTCPIPIndyMessagingClientTransport1.AnnounceSubscriptions;
end;
```

Notice the **AnnounceSubscriptions** call after activating the connection and after changing the subscriptions. This ensures that the hub knows about which messages this spoke node would like to receive.

Then compile the spoke node and run it.

If the hub node is not running, start it, and change the subscriptions memo to contain >
which means that the server subscribes for all message subjects. The click on Activate to allow spokes to connect.

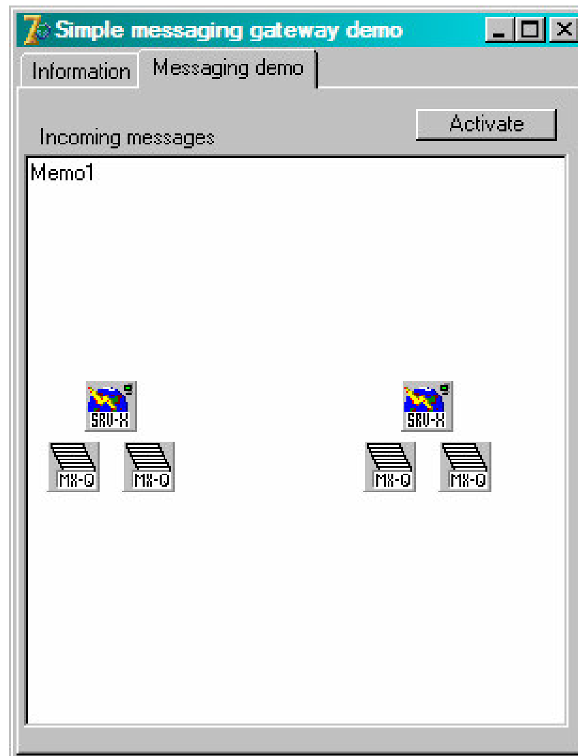
On the spoke node, also set the subscription to > and click Activate.
Each time you click the Send button, a message will be sent to the hub and relayed to this and other connected spoke nodes subscribing for the same subject.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Creating gateways and other advanced topics

The server messaging transports have a couple more interesting properties and events which are very useful if you, for example, would like to create a gateway between a hub/spoke setup and an UDP based broadcast setup.

Create a new application with a form looking similar to this:



This time we have added a **TkbmMWTCPIPIndyServerMessagingTransport** and a **TkbmMWUDPIndyServerMessagingTransport**.

Further we added 4 message queues, one inbound and one outbound for each of the transports and hooked them up.

Setup the transports bindings and ListenIP/Port and SendIP/Port's.
Notice that you must provide a unique Port value for each. In our case use port 4000 for the TCPIP transport and 4001 for the UDP transport.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Set the *OnClick* event of the Activate button to this:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
     kbmMWTCPIPIndyMessagingServerTransport1.Subscriptions.Text:='>';
     kbmMWUDPIndyMessagingServerTransport1.Subscriptions.Text:='>';
     kbmMWTCPIPIndyMessagingServerTransport1.Listen;
     kbmMWUDPIndyMessagingServerTransport1.Listen;
end;
```

Clicking on the Activate button will then make both transport start listening for and publish messages to their respective WIB segments. As this is a simple gateway we choose to listen for all subjects from each WIB segment.

For illustrational purposes add some code in the *OnMessage* events of each transport:

```
procedure TForm1.kbmMWTCPIPIndyMessagingServerTransport1Message(
  Sender: TObject;
  const TransportStream: TkbmMWCustomMessageTransportStream;
  const Args: TkbmMWArrayVariant; UserStream: TMemoryStream);
begin
     Memo1.Lines.Add('Message from spokes:'+TransportStream.Subject);
end;

procedure TForm1.kbmMWUDPIndyMessagingServerTransport1Message(
  Sender: TObject;
  const TransportStream: TkbmMWCustomMessageTransportStream;
  const Args: TkbmMWArrayVariant; UserStream: TMemoryStream);
begin
     Memo1.Lines.Add('Message from broadcast:'+TransportStream.Subject);
end;
```

Next we are going to look at the relay properties of the transports.

For the spokes to receive messages from other spokes, set the TCPIP server messaging transport's **AutoRelay** property to true (similarly to the simple Hub node example).

The new thing in this case is that we, in addition to relaying messages from spokes to other spokes. also want to relay the messages to the UDP based WIB segment.

Set the TCPIP server messaging transport's **AutoRelayAlt** property to true. Then set its **AltOutboundMessageQueue** to point to the UDP transport's outbound message queue.

For the alternative relay you have **RelayOptionsAlt** and **RelayTypesAlt** which are used similarly to the **RelayOptions** and **RelayTypes** properties, except instead controlling which messages to relay to an alternative message queue.

Because our gateway also should receive messages from the UDP segment and relay those to the Hub/Spoke WIB segment, we do the same operation on the UDP transport. Set its **AutoRelayAlt**

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

property to true, and its **AltOutboundMessageQueue** to point on the TCPIP transport's outbound message queue.

Now we have essentially created a simple gateway between the two segments.

The hub transport has a few more events which can be valuable at times.

- **OnMessageToSpoke** : This event can be used to programmatically select which spokes should receive which messages. Check the arguments provided and set the ASkip argument to true to skip relaying the message to the specific spoke, or false to let the spoke receive the message.
- **OnRelayMessage** : This even can be used to control if a specific message should be relayed to a specific message queue or not. Set ARelay to true to allow the relay, or false to reject it.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Request/Response using messsaging?

kbmMW fully supports request/response type messaging. In fact, you can replace a normal TCPIP transport with a messaging transport on a normal kbmMW client and app. server, and you will continue to be able to make requests from the client and receive responses from the app. server.

For an application server which use a server messaging transport to be able to receive requests from clients, the messaging transport must be subscribing for ‘`REQ.>`’

If you want only to react on requests from a specific client, you can add a hashed node id as an extra subject part. eg. ‘`REQ.0000339384.>`’ (see later for how to generate a hashed node id).

If you want the server only to subscribe for request messages for specific services, the subscription can be fine tuned even more: ‘`REQ.*.SOMESERVICENAME.>`’

Notice that if the actual service name contain .(dot), >(greater than) or *(asterix), those characters will have to be replaced with _ (underscore). The global function **kbmMWStripReservedSubjectChars** will do that for you.

The request subject has the following syntax:

```
REQ.anodeid.aservicename.aserviceversion.afunctionname.arequestid
```

anodeid is the hashed sender node id.
aservicename is the stripped name of the service requested.
aserviceversion is the stripped version of the service requested.
afunctionname is the stripped name of the function to call in the requested service.
arequestid is combined with the **anodeid** a unique value which identifies the specific request from the specific node. It is used when the requesting node receives the response, to match the response with the waiting request call. Remember all this happens asynchronously.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The response for the request is formatted like this:

```
RES.anodeid.arequesterid.aservicename.aserviceversion.afunctionname.arequestid
```

anodeid is the hashed sender node id (the servers hashed nodeid).
arequesterid is the hashed node id of the node initially making the request.
aservicename is the stripped name of the service requested.
aserviceversion is the stripped version of the service requested.
afunctionname is the stripped name of the function to call in the requested service.
arequestid is combined with the **arequesterid** a unique value which identifies the specific request from the specific node.

For the client to be able to receive the response, the client needs to subscribe for
`‘RES.*.aclientnodeid.>’`

where aclientnodeis is the hashed value of the clients own node id.
This can be generated like this:

```
aclientnodeid:=kbmMWGenerateNodeID(clienttransport.GetCurrentNodeID);
```

kbmMWGenerateNodeID can be found in *kbmMWGlobal.pas*.

GetCurrentNodeID return the value of the **NodeID** property if any is defined. If not, it return a unique GUID string which has been generated at program start. This string will change when the application shut down and is restarted. Thus generally better set the **NodeID** uniquely to identify the specific node before use.

Provided the correct subscriptions are setup, the client request/app.server response procedure will function as you are used to using a non messaging transport.

Its generally recommended to generate a unique node ID for each node before activating the node on the WIB.
A unique node ID can easily be generated by using the following global function:

```
function kbmMWGenerateUniqueNodeID:string;
```

The automatically generated unique node ID is based on the creation of a unique GUID.
You can choose to create your own node ID scheme, by using the kbmMWGenerateNodeID(ALocation:string):string global function.
Then provide your unique value as the ALocation argument and use the 10 digit hex decimal number as the node id.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

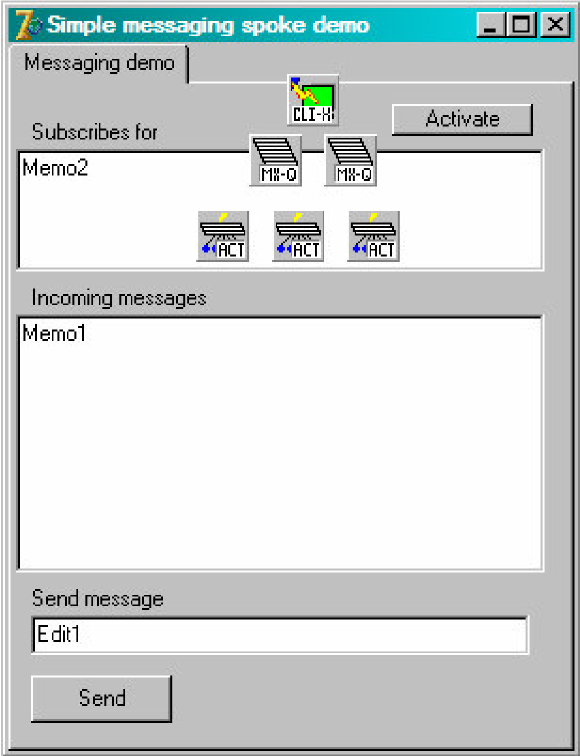
Splitting out messages

Until know we have been using the *OnMessage* event of the messaging transports. If you have a highly modularized node, you may want to have some mechanism to automatically split out messages to the places in the node where they are needed.

For this purpose the **TkbmMWMessageAction** component exists.

What it does is it allows you to specify a set of subscriptions which it then will react on. Its subscriptions must be a subset of the subscriptions on the transport, but can be specified differently. In other words, if the transport for example plainly subscribes for > (all messages), the message action component can choose to subscribe for 'MSG.X.>' and hence only for a subset of the messages the transport subscribes for.

As an example we can try to extend the spoke messaging sample.



In our case we add 3 message action components, and set their Transport property to point on the messaging transport.

Then we setup the Subscriptions on each to the subscriptions of our likings. Eg. one could have 'MSG.X.>' another 'SUB.>' and 'USB.>' etc.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Finally for the one subscribing for 'MSG.X.>' we write some code in the *OnMessage* event of that message action component, and for the one subscribing for 'SUB.>' and 'USB.>' we write some code in the *OnSubscription* and *OnUnsubscription* events.

You can have as many message action components as you like, however in a very high performance setup where thousands of messages may flow per second, one always have to think efficiency and thus not use more message action components than really needed. In the extreme cases, it may be prudent to use the events of the message transport instead as have been shown in the samples.

The message action component is also thread aware (as is the messaging system as a whole). Thus you can have multiple threads taking care of different messages.

If you want to publish messages in a threaded setup, you don't have to do anything special, but simply call **SendMessage** from the transport. It is thread safe.

You can also choose to typecast the Transport property of the message action component to have easy access to the transport's methods.

E.g.

If you know its a server transport the message action is hooked up on:

```
TkbmMWCustomMessagingServerTransport(messageaction.Transport).SendMessage(...);
```

or if its a client transport:

```
TkbmMWCustomMessagingClientTransport(messageaction.Transport).SendMessage(...);
```


Copyright 2001-2004 Components4Developers
http://www.components4developers.com

The SRV, THR and CAC subject types

SRV (Service call) is formatted the same as the REQ subject and is intended for calling a function on an application server, but without expecting any response back. It can be called from any messaging node by using this subject format:

```
SRV.anodeid.aservicename.aserviceversion.afunctionname.arequestid
```

Please see REQ for a description of the subject parts.

THR (Throttle) is a special subject type which is intended to be used by nodes to signal to other nodes that they are too busy processing incoming messages, or that they are now able to receive faster than they were before due to decreasing processing load. It's not implemented and is for future use.

CAC (Cache) is a special subject type which is intended to be used by nodes which want to get access to older messages. A cache node will then be running and will cache all messages for which it subscribes for within a configurable period of time. Newly started nodes which builds up some kind of information flow based on delta information, could then request all the relevant information until current time from the cache, and then receive the normal delta updates the usual way from the node publishing those.
The subject type is still not implemented and is for future use.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

Useful global methods

Lots of useful functions and procedures exists in kbmMWGlobal lots of useful methods exist for handling subject strings:

function kbmMWSubjectPart(const ASubject:string; const APartNo:integer):string;

Extract a part from a subject string. The APartNo is a number from 0 to n-1 where n is the number of parts in the subject. It is valid to use a large number to identify last part. The part is returned without any part seperators (.).

procedure kbmMWSubjectParts(const ASubject:string; AStringList:TStrings);

Extract all subject parts from a subject string into a stringlist. A TStrings instance have to be provided.

function kbmMWSubjectFromPart(const ASubject:string; const APartNo:integer):string;

Extract the remaining part of a subject starting from a specific part number (0 to n-1).
For example useful when listening into nodes subscription and unsubscription announcements.

function kbmMWStripReservedSubjectChars(AString:string):string;

Will replace all occurences of . (dot), > (greater than) and * (asterix) with _ (underscore). Those 3 characters are not allowed in subject parts, except for the subject which is appended in a subscription or unsubscription subject.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

function kbmMWGenerateNodeID(const ALocation:string):string;

Takes a string (ALocation) and generates a 10 digit hexdecimal node identifier. ALocation can essentially be any string, but you should generally use the result from the transports GetCurrentNodeID function as argument.

function kbmMWGenerateUniqueNodeID:string;

Generates a unique 10 digit hexdecimal node identifier. It’s the same as: kbmMWGenerateNodeID(kbmMWGenerateGUID)

function kbmMWGenerateRequestSubject(const ANodeID:string;
const AServiceName:string;
const AServiceVersion:string;
const AFunc:string;
const ARequestID:longint):string;

Generates a properly formed REQ subject string.

function kbmMWGenerateResponseSubject(const ANodeID:string;
const ARequesterNodeID:string;
const AServiceName:string;
const AServiceVersion:string;
const AFunc:string;
const ARequestID:longint):string;

Generates a properly formed RES subject string.

function kbmMWGenerateMessageSubject(const ANodeID:string;
const ASubject:string):string;

Generates a properly formed MSG subject string.

function kbmMWGenerateSubscribeSubject(const ANodeID:string;
const ADelta:boolean;
const ASubject:string):string;

Generates a property formed SUB subject string.

Copyright 2001-2004 Components4Developers
http://www.components4developers.com

**function kbmMWGenerateUnsubscribeSubject(const ANodeID:string;
const ASubject:string):string;**

Generates a properly formed USB subject string.

This concludes the whitepaper about the Wide Information Bus.

Kim Madsen
Components4Developers

Copyright 2001-2004 Components4Developers
http://www.components4developers.com