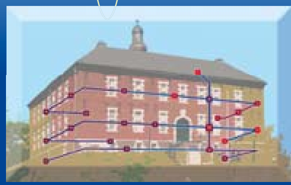
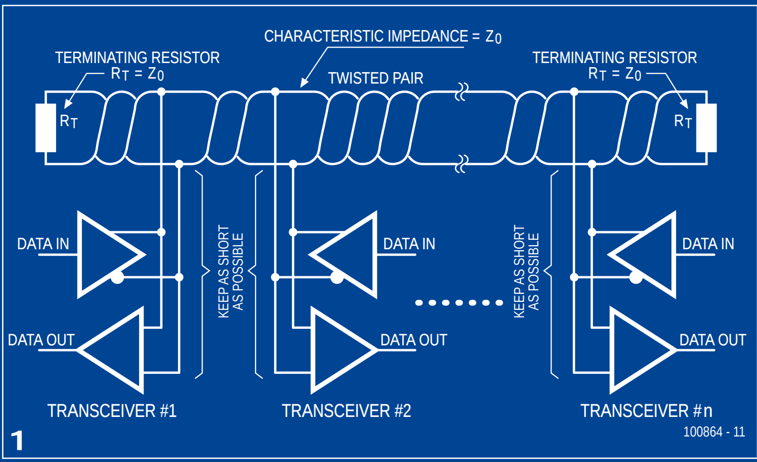
Here comes the Bus! (2)



By Jens Nickel

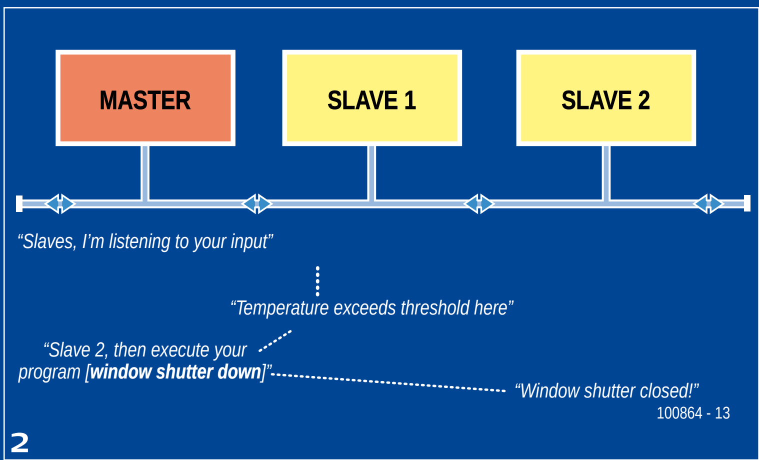
Readers whose memories stretch back to our previous issue will recall that in the first part of this series our small but highly effective team decided that electrically the ElektorBus would be based on the RS-485 standard, operating over a twisted pair. To provide reliable communications each of our bus participants needs to be able to send and receive data. The bus is wired as shown in Figure 1, which is based on a Maxim application note [1]. The screenshot on the next page shows how not to do it.

With all bus nodes connected to the same pair of wires, the obvious sixty-four thousand pound question is: how do we make sure that only one bus node is talking at any given time? Unlike the CAN bus standard, the RS-485 standard does not specify a mechanism for detecting collisions, and without such a mechanism we are in danger of losing data.



1

As you might suspect, we spent some time discussing the problem, coming up with several alternative solutions.



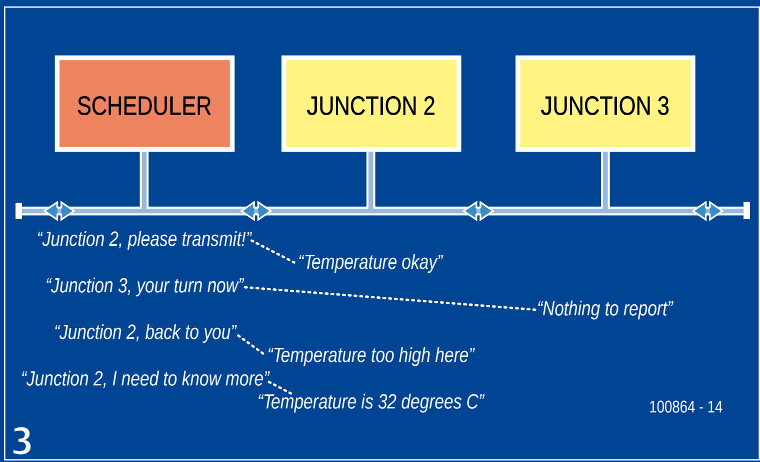
2

The simplest approach is to make one of the nodes the boss, with the underlings simply doing what they are told and speaking only when spoken to. The master-slave arrangement has the advantage that the slave nodes can be kept very simple and to a large extent standardised, which in turn relieves the developer of a considerable burden: all the nodes can use the same microcontroller, and even be running identical firmware. The master simply issues commands like ‘take port pin PB5 High’, or ‘take a reading from ADC1 and send it to me’. The software in the slave microcontrollers then simply has to parse the commands (of which there need only be a few different types) and then suit the action to the word).
However (as you might have guessed from the length of this article), there are some serious downsides to this quasi-direct access of the bus master to the slave’s I/O pins. The most significant of these is that the master must know exactly how each slave is wired. For example, if a slave includes a temperature sensor, the master must somehow know how to convert a raw A/D converter reading into a temperature value. It also makes for a lot of bus transactions. Consider, for example, the task of raising a roller blind until a limit switch is actuated. The conversation between master and slave might go something like this: “Set port pin PB5 High.” “Done that.” “Now, is port pin PC1 High?” “No.” “How about now?” “Yes.” “Okay, take port pin PB5 Low at once.”

So as you can imagine this idea was rapidly sent on its way to the shredder. After all, what we have is more of an inter-microcontroller communications protocol aimed at a certain narrow range of applications than a true bus system. In my mind’s eye I was picturing a fully-fledged home automation system, with the slaves having at least a modicum of intelligence. This means that a node should for example translate a raw A/D converter reading into a physical quantity so that different types of sensor, converter and microcontroller can be mixed on the same bus without the bus master needing to know the details. It would also be desirable to implement simple control loops running within the slave (of the form ‘set output X low until input Y goes high’), which would be enough to cover cases such as the roller blind example above.

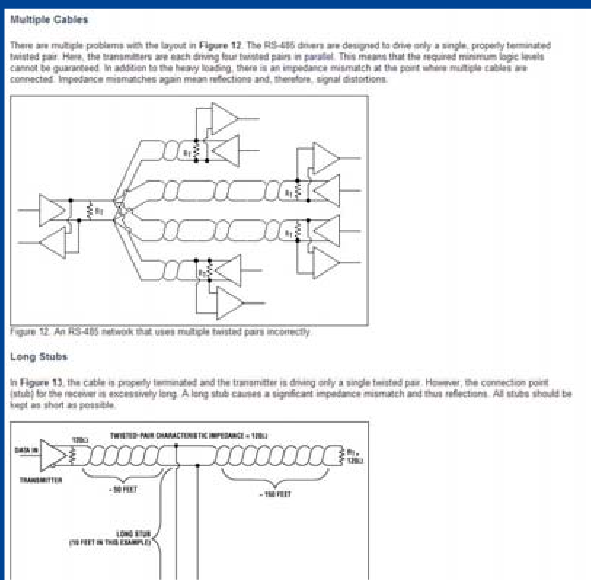
It also seemed at first sight to be a little impractical to have the slaves only send messages on request. When values need to be monitored, this means that the master must interrogate the slave on a regular basis, which, besides feeling inelegant, might result in latencies unacceptably great for applications such as alarm systems. In my vision of the bus system (Figure 2) the master can go into a ‘listen mode’, waiting for a range of events such as ‘input 2 on slave 1 has gone low’ or ‘temperature at slave #3 has gone over 100 °C’. By design these events should be relatively rare, which will help minimise the number of collisions on the bus.

“That should be enough for a modest home automation system,” said my French colleague Clemens, “but what else could we do with the bus?”
“Well,” I said, “we could provide a ‘fast transmit mode’ to allow rapid point-to-point communications.” That would allow us to send rapidly-changing data to the master, at least from one of the slaves.



3

It was obvious to us both that in this situation the bus would not be available for other activities and that we could in some circumstances miss important event notifications from other nodes. “We need some kind of prioritisation on the bus,’ said Clemens, ‘we need nodes that are workers, under-managers, over-managers, under-over-managers, over-under-managers...” So, like the CAN bus in a car, where (in the fullest sense of the word) vital data can always get through?
“And what happens,” asked Clemens, warming to his point, “when we have more than one node making decisions? In your design only the master collects data, but if all the nodes have access to all data packets, there isn’t really a single master any more.”


Multiple Cables

There are multiple problems with the layout in Figure 12. The RS-485 drivers are designed to drive only a single, properly terminated twisted pair. Here, the transmitters are each driving four twisted pairs in parallel. This means that the required minimum logic levels cannot be guaranteed. In addition to the heavy loading, there is an impedance mismatch at the point where multiple cables are connected. Impedance mismatches again mean reflections and, therefore, signal distortions.

Figure 12. An RS-485 network that uses multiple twisted pairs incorrectly.

Long Stubs

In Figure 13, the cable is properly terminated and the transmitter is driving only a single twisted pair. However, the connection point (stub) for the receiver is excessively long. A long stub causes a significant impedance mismatch and thus reflections. All stubs should be kept as short as possible.

Oh my giddy aunt, I thought to myself, things are starting to move quickly. If the nodes are allowed to talk to other nodes without going via the master, then we are getting close to designing a network that can tolerate faulty nodes. How do we stop the nodes from chattering away uncontrollably to one another on the bus? “How about allocating defined time-slices?” suggested Clemens, “though that would of course put a limit on the number of devices we could support simultaneously.”

There followed an hour of furious googling and sending one another links. We discovered, for example, a Siemens patent on a time-slice-controlled symmetric bus architecture which could be used for sending home-automation commands at the same time as multimedia data streams [2]. Also of interest was a presentation on ‘Time-triggered CAN’ [3].
The advantages of a time-slice architecture were very seductive, but the details involved in synchronisation would be fiddly. I also felt that it would be difficult to get such a system up and running with a reasonable amount of development and debugging time, both for us and for our readers.

“All right then,” said Clemens, “what about using some sort of scheduler?” This would allocate to each of the other nodes a time to speak based on the importance of what it might have to say, with the nodes being scheduled in order of decreasing priority. “Something like a pre-emptive multitasking scheduler,” he explained.

I had to concede that this approach to collision avoidance was not bad, even though it was almost exactly the opposite of what I had originally envisaged. Nevertheless, the problem still remained of how the scheduler could stop a node from talking if the bus was needed for something more important.

The solution to this came to me somewhat later: each node would only be allowed to send a fixed number of bytes before having to give way to the next node (Figure 3). If a node reports a higher-priority event, the scheduler would then give it permission to speak. All we needed to do now was to try out these fine ideas to see if they would actually work in practice...

(100864)

What do you think? Feel free to write to us with your opinions and ideas.

[1] http://www.maxim-ic.com/app-notes/index.mvp/id/763
[2] http://www.patent-de.com/20030320/DE10126339A1.htm (in German)
[3] http://www-lar.deis.unibo.it/people/crossi/files/SCD/An%20Introduction%20to%20TTCAN.pdf