Here comes the Bus! (6)
Elektor experimental PCBs and more

Our bus reaches the next stop on its route with the appearance of a couple of boards aimed at application development. These easy-to-assemble units include an experimental node with analogue and digital inputs and a compact USB to RS-485 converter. We also describe a simple system that guarantees efficient and reliable communication on the bus.

By Jens Nickel (Elektor Germany editorial)

Spurred on by our encouraging initial results, we now proceed to describe a couple of simple boards that can be used to develop bus-based applications.

The basis of our experimental node is the circuit of the 'test node' we gave in the previous instalment of this series. That design sported just a 'test LED' and a 'test button', which were enough to check whether messages could be sent successfully over the bus. All very worthwhile, but not enough to develop realistic applications.

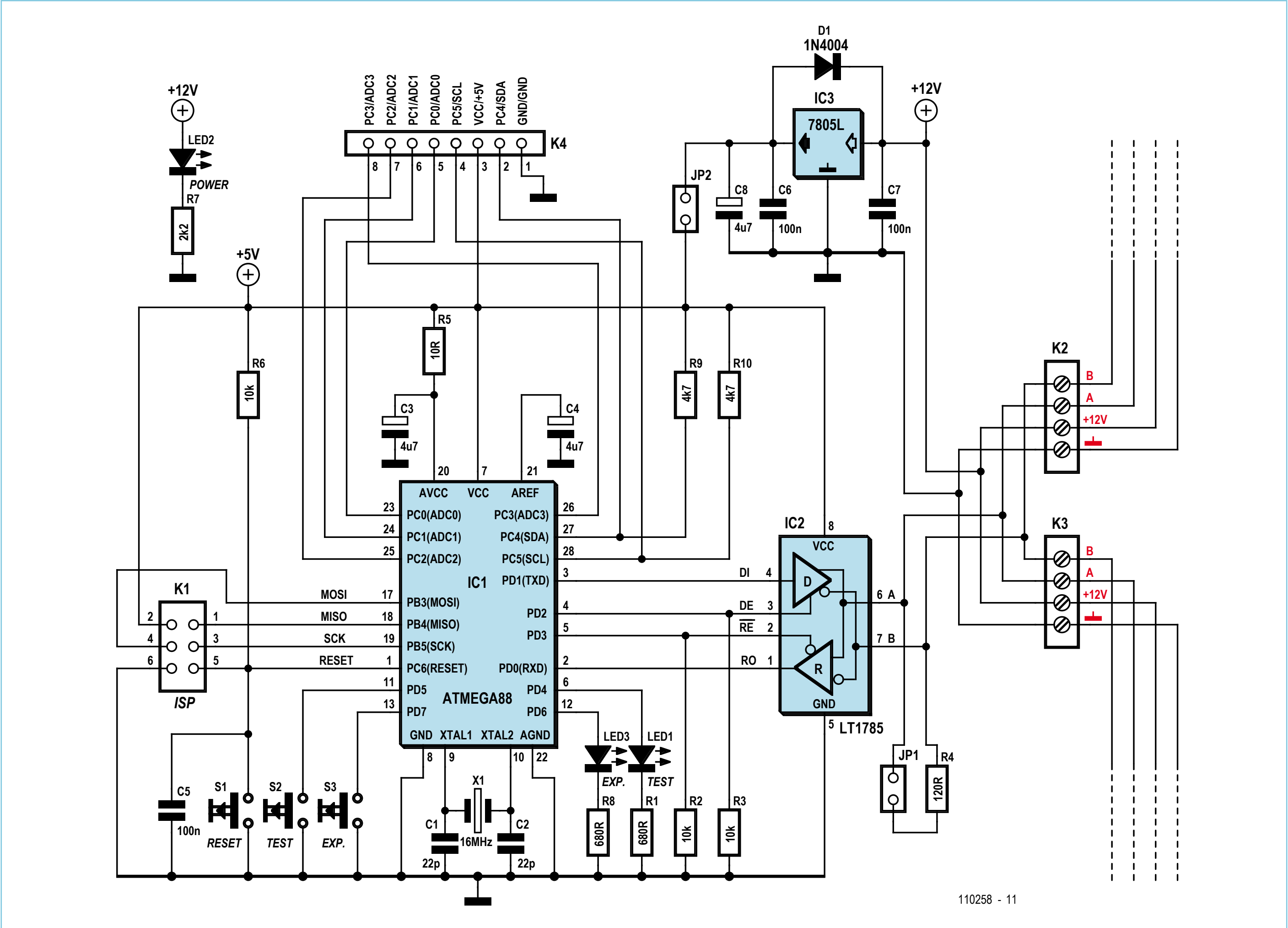
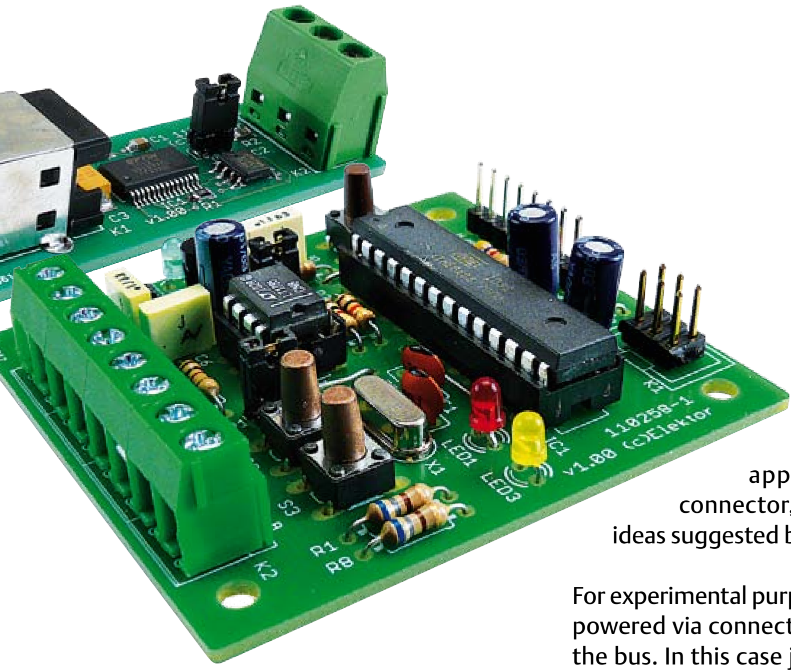


Figure 1. Circuit diagram of the experimental node.



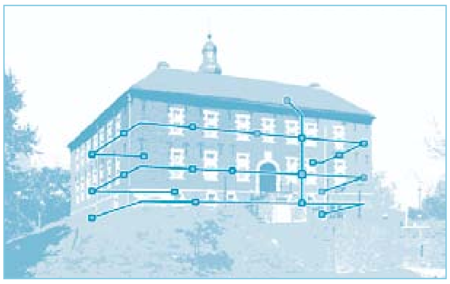
Experimental node

The circuit diagram for the new ATmega-based bus node is shown in **Figure 1**. The operation of the RS-485 bus driver was described in the previous article in this series. The doubled-up connector blocks allow the node to be connected to the bus and for the bus signals to be wired on to the next node. A small extra feature is the jumpered connection to a 120 Ω resistor, which allows the RS-485 bus to be properly terminated at each end.

We have connected an extra LED and an extra button to port pins PD6 and PD7: we will call these the 'experimental LED' and the 'experimental button'. The most significant new feature, however, is header K4, where six additional microcontroller pins and the microcontroller power supply and ground connections are brought out. The decision to use port pins PC0 to PC5 was carefully considered, as these (as is the case in many microcontrollers) have various special features. All the pins can be used as digital inputs or outputs; PC0 to PC3 can alternatively be used to measure four separate analogue voltages in the range from 0 V to 5 V with the help of the microcontroller's built-in multi-channel analogue-to-digital converter; and PC4 (SDA) and PC5 (SCL) can be used to provide access to the I²C interface in the device. In accordance with the specification of that bus we have equipped the SDA and SCL signals with a pull-up resistor each.

It is therefore possible to connect simple sensors, I²C bus slaves such as temperature monitors and other similar devices to K4. The ATmega88 datasheet [1] shows how the pins are controlled using the registers in the device. Later in this series we will develop various applications using this connector, including, we hope, ideas suggested by readers.

For experimental purposes the node can be powered via connector K4 instead of over the bus. In this case jumper JP2 should be removed. In operation it is definitely recommended to connect the ground of the supply to bus ground as well as to earth. For more on this point, see the previous article in the series [2].
For completeness we should mention that we have wired the AVCC and AREF connections for the microcontroller's internal A/D converter as suggested in its datasheet.
The single-sided printed circuit board (**Figure 2**) is easy to populate. It is simplest to start with the wire links.



USB-to-RS-485 converter

There is relatively little to say about this board: it corresponds directly to the circuit diagram of the USB-to-RS-485 converter given in the previous article in this series (**Figure 3**). As the FTDI device is only available as a surface-mount component, we decided to use SMDs throughout the circuit and offer the board as a ready-assembled unit (**Figure 4**). I believe this is the simplest and most compact USB-to-RS-485 converter we have ever published in Elektor. It is ideal for developing applications based on the bus, for example with the help of the PC software described in the previous article. A minor caveat is that the design of the ElektorBus system is not yet finalised, and so we cannot at the moment say for certain whether we will be looking at a more intelligent connection between the RS-485 bus as the PC at some point in the future (more on this below). Meanwhile the converter is still a useful unit to have, and it can of course

COMPONENT LIST

Experimental Node

Resistors
R1,R8 = 680Ω
R2,R3,R6 = 10kΩ
R4 = 120Ω
R7 = 2.2kΩ
R9,R10 = 4.7kΩ

Capacitors
C1,C2 = 22pF
C3,C4,C8 = 4.7µF
C5,C6,C7 = 100nF

Semiconductors
D1 = 1N4004
IC1 = ATmega88-20PU
IC2 = LT1785
IC3 = 78L05

Miscellaneous
X1 = 16MHz quartz crystal
S1,S2,S3 = pushbutton
JP1,JP2= jumper
LED1 = LED, 3mm, red
LED2 = LED, 3mm, green
LED3 = LED, 3mm, yellow
K1 = 6-pin (2x3) pinheader, lead pitch 2.54mm (0.1 in.)
K4 = 8-pin pinheader, lead pitch 2.54mm (0.1 in.)
K2,K3 = 4-way PCB screw terminal block, lead pitch 5.08mm (0.2 in.)
PCB # 110258-1 [3]

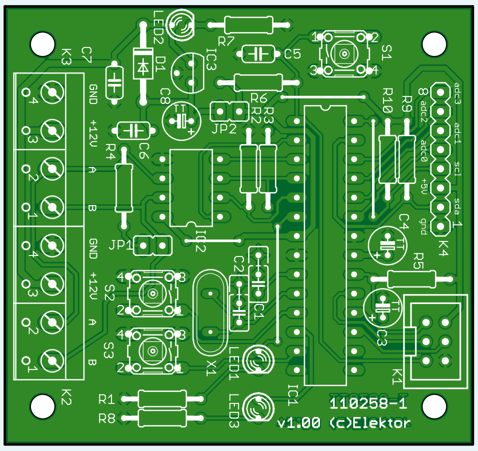


Figure 2. The circuit board is populated with conventional components throughout.

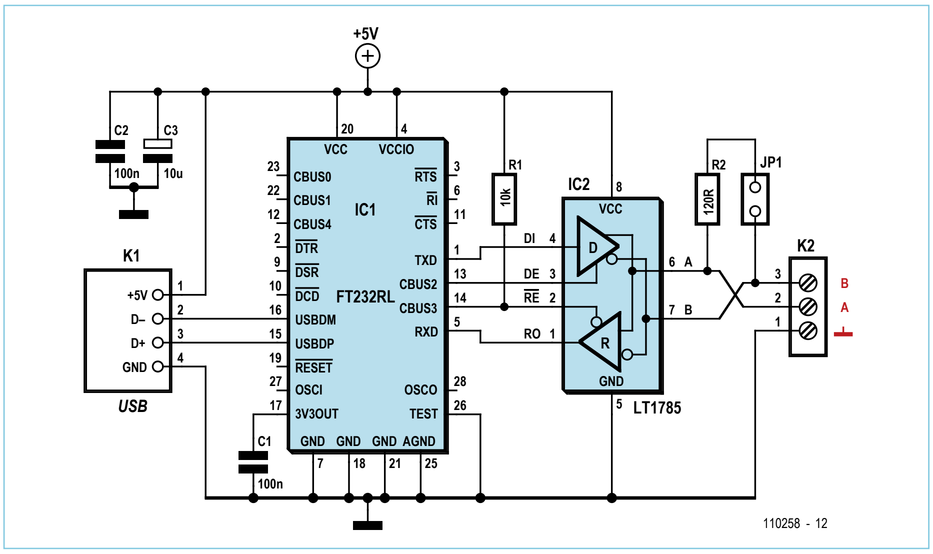


Figure 3. Circuit diagram of the USB-to-RS-485 converter.

also be used for other RS-485 applications. Since we are now set up with the boards we need for our experiments, we can turn to the question of software. The test software we described in part 5 [2] does not offer any mechanism to avoid (or even detect) collisions. Below we will look at a simple system which guarantees efficient and reliable communication on the bus. Alternative ideas have been discussed on our dedicated mailing list, and there is a brief summary of these in the text box.

Round-robin polling

In the previous articles in this series we looked at the pros and cons of cyclically polling the nodes. Each node in turn would be interrogated by a scheduler using a 16-byte message, and would reply with a message. A simple system like this would have two serious drawbacks.

1. A node that has a message that it wants to transmit must wait its turn. For example, consider what might happen when a user operates a light switch in a home automation system. In the worst case the node might have to wait while all the other nodes on the bus are interrogated. At 9600 baud with 32 nodes on the bus, it could easily take over a second before the light comes on, which is not acceptable.

2. There would be a large number of empty messages on the bus. It makes no sense, for example, to interrogate a light switch every second throughout the day if the switch is only operated perhaps ten times during that day.

The other extreme

In the case of nodes, such as switches, which only need to communicate infrequently, an alternative approach is better. The node can be arranged to transmit only when the switch is operated, without regard for the other nodes on the bus. Of course, this means that collisions can occur, resulting in random sequences of bits appearing on the bus. Instead of a neat 16-byte packet we suddenly receive a longer sequence of bytes with scrambled values, neither of the two colliding messages making it through unscathed. This means that, for important messages at least, the receiver should send an acknowledgement of safe receipt. If no acknowledgement is received by the original sender, it must repeat the transmission. This simple protocol also ensures that messages do not get lost as a result of other interference on the bus.

However, there are disadvantages. If several nodes are active simultaneously, all sending messages relatively frequently, then there will be many collisions. Also, a processing node that receives, for example, regular temperature readings from a sensor node must match each reading with an acknowledge message, which increases both the load on the bus and the risk of collisions occurring.

A happy medium

Reader Jürgen Lange and I independently hit upon the same idea: let's see if we can get the best of both worlds! We can switch periodically between a polling mode (nodes speak only when spoken to) and the other extreme (where nodes can send messages at will without regard to the activity of other nodes).

COMPONENT LIST

USB/RS485 Converter

Resistors
R1 = 10kΩ (0805)
R2 = 120Ω (0805)

Capacitors
C1,C2 = 100nF
C3 = 10µF 16V (6032)

Semiconductors
IC1 = FT232RL
IC2 = LT1785 (SO-8)

Miscellaneous
JP1 = jumper
K1 = USB socket Type A
K2 = 3-way PCB screw terminal block, lead pitch 5.08mm (0.2 in.)
PCB # 110258-2 [3]
or
ready assembled and tested PCB # 110258-71 [3]

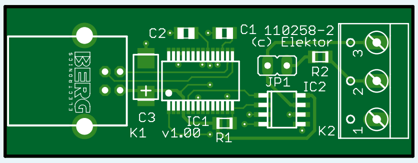


Figure 4. The USB-to-RS-485 converter is available from *Elektor* ready assembled and tested.

The alternative: a bit-level approach

When designing the experimental bus node, I thought that just developing the circuit would be enough of a challenge. However, that was nothing compared to the job of monitoring and leading the discussions on the mailing list. As alluded to in the previous article in the series, there was a number of experienced bus designers who were strongly in favour of a low-level approach to collision detection. CAN expert John Dammeyer suggested a method whereby each transmitter must wait for a 'space time' of 12 bit periods before sending a message. Since it also listens to the bus it can detect whether another node on the bus is talking. For various other approaches to collision detection that we discussed together, so-called 'bit banging' (direct manipulation of the UART port pins in software) would be required.

Other frequently-discussed topics were various low-level approaches to (re-)synchronisation of bus nodes to the start of each message. *Elektor* reader Walter Trojan was in favour of using the 'nine-bit' mode of the UART, offered by the ATmega88 and various other microcontrollers. (Atmel call this 'multi-processor communication mode': see [1]). This sends each byte as nine bits rather than eight, the extra bit being used, for example, to distinguish between address and data bytes.

We cannot go into all the advantages and disadvantages of these various methods here. Suffice it to say that some of the more sophisticated approaches have what I see as a decisive disadvantage in that they restrict the choice of processors and platforms that can be used in bus nodes. Not all microcontrollers offer a nine-bit mode, and bit-banging in PC software strikes me (even assuming it is possible) as hardly elegant. And what if we want to send our messages over a different, perhaps wireless, network? The byte is a basic unit of information that practically any system can handle, whereas using nine-bit words and individual bits makes things rather more complicated.

As you might imagine, this discussion rapidly gathered pace. I earned a certain degree of opprobrium for wanting to keep open the possibility of controlling the RS-485 bus directly from the PC. Because of the timing idiosyncrasies of the Windows operating system this is quite a tricky proposition, and there is a lot to be said for replacing the USB-to-RS-485 converter with a USB-to-RS-485 gateway, which would include its own microcontroller to handle bus communications.

In summary: I felt that it was a decisive advantage to be able to transport our 16-byte messages seamlessly across different platforms and networks. In order to make the system maximally flexible, I decided that it would be best to ensure reliable and guaranteed communication as far as possible at the message level (see text) rather than at the bit level, closer to the hardware. Again, we welcome your feedback!

First, all the nodes that need to be interrogated periodically (such as temperature sensor nodes) are probed in turn. The scheduler then releases the bus for the unprompted transmission of messages. At this point any node that only occasionally has something to say (such as a light switch) is permitted to speak. The 'free bus phase' must of course only continue for a certain period of time, so that nothing is accidentally still being transmitted when polling mode resumes.

To implement this protocol, which I dubbed 'hybrid mode', I extended the software described in the previous instalment [2]. First we will describe the implementation of the basic functions, which will subsequently be packaged up into a library; then we will look at a small application. Everything is, as usual, downloadable as source code from the *Elektor* website [3].

The scheduler

The PC takes on the role of the scheduler. Its transmitter address is defined as 0, which makes it easy for the other nodes to recognise its messages. The scheduler maintains an array, with *intPolledNodesMax* elements, containing the addresses of the nodes that are to by polled cyclically. It is also possible, of course, to arrange for a particularly loquacious node to be interrogated more often than the others.

To poll a node the scheduler sends out a special request message (*SchedulerRequest*), which includes the address of the polled node in the recipient address field. The scheduler then waits for a message with the same value in the transmitter address field (*ResponseMessage*), which can have any desired value in the recipient address field. The scheduler then turns to the next node in sequence and the process repeats. If a node fails to reply to a polling message, the process would come grinding to a halt. For this reason a timer is started when the *Sched-*

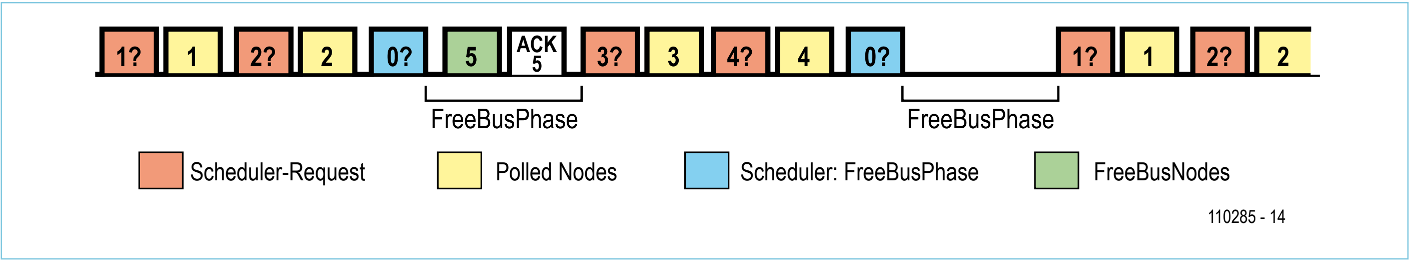


Figure 5. In h*ybrid mode* polling phases alternate with free bus phases. Collisions can occur during the *FreeBusPhase*, which means that the recipient must reply to each message with an acknowledgement.

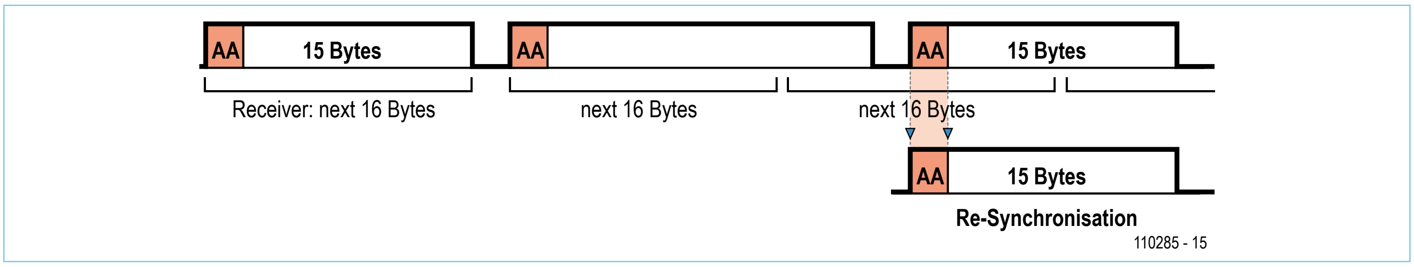


Figure 6. Our small application uses the first three bytes of the *Elektor* message protocol (the CRC is not implemented). Bit 0 of the mode byte indicates (by being set to zero rather than one) that the message originates from a polled node.

ulerRequest is sent out: if the timer expires without a reply being received, the scheduler stops waiting and moves on to the next polled node anyway.

The variable *intFreeBusCycle* has a special significance. It determines how many nodes will be polled cyclically before the scheduler switches to a *FreeBusPhase*. If the variable has the value 2, for example, this means that free transmission is permitted after every two nodes polled cyclically, as listed in the array (see **Figure 5**).

To initiate the *FreeBusPhase* the scheduler sends out a special *FreeBusMessage*, which has the recipient address field set to zero; the data payload bytes are not used. The scheduler then waits for a period *intFreeBusTime* (specified in milliseconds) before resuming polling.

Firmware

The BASCOM software from the previous article in this series was used as the basis for the code in the microcontroller nodes. Taking the advice of Günter Gerold on our bus mailing list, I modified the interrupt routine so that it returns immediately after receiving one byte. As before, the received bytes are stored in a byte array, and when the sixteenth byte is received, the routine checks wither the recipient address field matches the node's own address or is set to zero (a *FreeBusMessage*). In these cases the *ReceivedEventFlag* is set, which acts as a signal to the main loop that a message has been received and needs processing.

Now, when a node is directly interrogated by the scheduler (that is, using its own address), it is essential that it always reply immediately. The only nodes that reply to a *FreeBusMessage* are the nodes that are not polled (the *FreeBusNodes*), and even then, only when they have a message ready to send (when the *SendEventFlag* is set to true). In this case, if the message is to be sent out immediately upon receipt of the *FreeBusMessage*, then it is possible to avoid the use of a timer in the microcontroller to check for expiry of the *FreeBusTime* period.

A value (*PollingStatus*) is stored in the microcontroller's EEPROM to indicate the type of node. This allows the same firmware to be used in both node types.

When sending the message we make use of a hitherto unused bit in the mode byte (see the Elektor Message Protocol in **Figure 7**). Bit 0 being set to zero means that it is a *ResponseMessge*, and so collisions should not occur. If bit 0 is set to one then the message is part of the *FreeBusPhase* and so not safe from collisions: the receiver must reply with an acknowledge message. In no acknowledgement is received, then there has probably been a collision and the transmitter must send the message afresh, waiting at least until the next *FreeBusPhase*. To avoid repeated collisions, we need to ensure that the other message involved in the collision is not also resent in the same phase. A simple system ensures this: each transmitter waits for a different number of *FreeBusPhases* to pass before retrying. This number (the *FreeBusPriority*) is statically programmed into the EEPROM in the node, just like the node address. Further development of the software could allow the address, the *PollingStatus* and the *FreeBusPriority* to be changed dynamically.

In the course of a collision it is possible that sequences of bytes appear on the bus making packets of more than 16 bytes. This means that we need a mechanism to resynchronise to the beginning of the next message. In this version of the software we offer a partial solution to the resynchronisation problem: we simply scan the bytes of the incoming data stream for the value AA$_{hex}$, which appears at the start of each message (**Figure 6**). The result of this approach is the restriction that this byte value may not appear within the data payload, and that we cannot use the CRC for error detection in case the byte value should appear there.

A small application

Now we turn to a small application, which will let us test our first experimental nodes. To produce a continuously-variable value that can be read regularly, I wired a 100 kΩ potentiometer to K4, between the 5 V, ADC0 and GND pins. I connected the experimental node to the bus with the two test nodes from the previous article in this series, which Günter had laid out and populated.

The experimental node was programmed with the address '02' and a *PollingStatus* of '01' in its EEPROM. The two test nodes were given addresses '01' and '03', a *PollingStatus* of '00' and *FreeBusPriority* of '01' and '02'.

In the interests of simplicity I used the same firmware for all three nodes. Pressing the test button sets the *SendEventFlag*, toggles the test LED and copies the LED status to the *LEDbyte*, which forms the first byte of the data payload of the message that will subsequently be sent (Figure 7). The value of *PollingStatus* also determines whether ADC0 will be read. (Remember that the ATmega88 needs to be told to use AREF as its voltage reference: see the source code.) We divide the ten bits of the ADC conversion result into two bytes, which will form the second and third data bytes of the message. Since we are not allowed to use the value AA_{hex} in the payload, we put the seven (rather than eight) least-significant bits of the result in the *ADClow* byte and the remaining three most-significant bits in the *ADChigh* byte. We will use the PC to receive the messages, displaying the status of the three LEDs, as well as the ADC result, appropriately formatted. So as not to cause a conflict with the scheduler address (which is zero) we allocate a second address (10 in this case) to the PC. This address is used for sending acknowledge messages to nodes 1 and 3: in other words, the message has the recipient address set to the node address, the transmitter address set to 10, and the first data payload byte set to 16 plus the value of *LEDbyte*.

The secret is in the timing

After a couple of experiments it appeared that a value for *FreeBusTime* of between 50 ms and 70 ms was adequate. Since in this small application example the PC acts simultaneously as scheduler and as receiver for the microcontrollers' messages, I rather inelegantly let it send an acknowledge message (asynchronously) just after the end of the *FreeBusPhase*, only after that returning to interrogating the nodes. Normally the scheduling and the sending of acknowledge messages should be done synchronously. The message containing the reading and the acknowledge message would then both fall within a *FreeBusPhase*, as shown in Figure 5.

It took several attempts before the software was running correctly. One of the bugs caused me a particularly large amount of head-scratching before I managed finally to track it down. The symptom was that the microcontroller firmware was losing occasional incoming messages. The explanation was that in the interrupt service routine I had not checked for the correct recipient address. When during the processing in the main loop (checking buttons and reading the ADC) more than two messages were sent on the bus, the second message was overwriting the variables required to process the first message properly. The fix was simply to check the recipient address in the interrupt service routine and then accept the message for further processing. Then more time is available for the application code in the microcontroller, such as for processing readings. Here we are helped by the fact that in hybrid mode we never send two messages in succession to the same receiver.

Outlook

Since the PC simultaneously takes on the personae of scheduler, bus participant and display unit, the current version of the software is somewhat untidy. In the future we will be able to simplify things by implementing the scheduler in a microcontroller. One possibility would be to use the microcontroller to control a (yet to be developed) USB to RS-485 gateway. This option has been suggested by various people on the mailing list: see the text box.

The resynchronisation mechanism needs to be revisited so that we can allow the value AA_{hex} to appear within the data payload or CRC bytes. We already have some ideas for how to go about this.

Also, we want to look at how a node can announce its presence to the scheduler, and how to manage dynamic addressing. Last but not least we will also look in the next instalment in this series at a realistic application. And, as ever, we invite you to participate and write in with your opinions and ideas.

(110258)

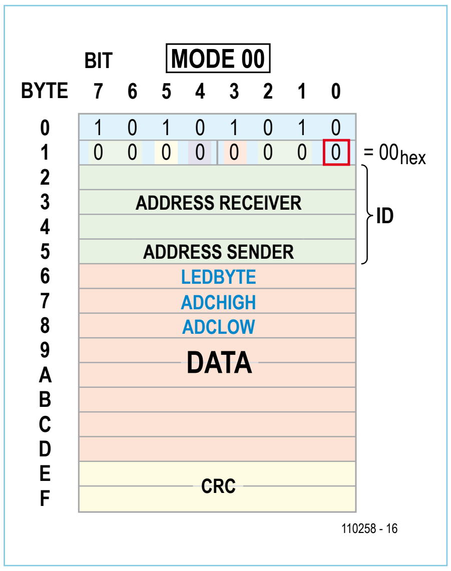


Figure 7. If a collision occurs sequences of more than 16 bytes can appear on the bus. In this case all the bus nodes need to be able to resynchronise to the start of the next message by scanning the stream for a byte with the value AA_{hex}.

Internet Links

[1] www.atmel.com/dyn/resources/prod_documents/doc2545.pdf

[2] www.elektor.com/110225

[3] www.elektor.com/110258