REIHE INFORMATIK
TR-2006-004

The Impact of Resync on Wireless Sensor Network Performance

Marcel Busse, Thomas Haenselmann,
and Wolfgang Effelsberg

Universität Mannheim
Praktische Informatik IV
Seminargebäude A5
D-68159 Mannheim, Germany

The Impact of Resync on Wireless Sensor Network Performance

Marcel Busse, Thomas Haenselmann, and Wolfgang Effelsberg

Computer Science IV, University of Mannheim,Seminargebäude A5
D-68159 Mannheim, Germany
{busse, haenselmann, effelsberg}@informatik.uni-mannheim.de

Abstract. Many of todays sensor nodes exhibit special transmission errors. These errors are due to some common hardware components being used, particularly the so-called UART (serial communication) circuits that interconnects radio transceivers and the CPUs. UARTs generate start-, data- and stop-bits. As long as the state machine at the sender and the receiver is synchronized, even single bit errors can often be corrected by Forward Error Correction (FEC). However, once one or several bits are missed, the state machine at the receiver side will get out of sync so that data bits are misinterpreted as start- or stop-bits and vice versa, rendering the entire remaining communication useless. In this paper, we will devise a periodic resync scheme that enables the receiver to catch up on a data stream even in case of skipped bits. In noisy environments as well as for weak senders, we can improve the overall data throughput significantly.

Key words: Measurements, Resync, UART, Wireless Sensor Networks

1 Introduction

In early evaluations of our *Embedded Sensor Board* (ESB) platform, we encountered a poor delivery rate for weak radio signals and/or large inter-node distances, as it is shown in [1]. So we analyzed the probability of erroneous bits within a single packet in more detail. Surprisingly, the probability of a bit being inverted was not distributed evenly over all bits within a packet but was an almost monotonously rising function. As our analysis will proof, this is due to the fact that the radio transceiver may miss or overhear single bits. As a consequence, all following bits are shifted to the left with regard to the true bit-stream being sent. Intuitively, shifting them a few bits to the right would align them with the true stream again. But getting the stream re-synchronized is quite a challenge and tackled in this paper.

2 Communication Characteristics

2.1 The Embedded Sensor Board

The ESB platform was developed by the Free University of Berlin [2, 3] and is equipped with the 8 MHz MSP430 chip from Texas Instruments [4, 5]. Most of its 64 kB memory are implemented as flash memory which contains the software and all constant data. The RAM occupies 2 kB. For the radio communication, the RFM TR1001 radio

transceiver [6] is used, which sends up to 1 mW and can be powered down to zero in 100 steps. The radio range depends heavily on the environment and can range from several 100 meters outdoors to less than 10 meters. Using a *Universal Asynchronous Receiver-Transmitter* (UART), the radio transceiver is connected to the processor. We will now describe the UART in more detail.

2.2 UARTS

A UART is basically an 8-bit shift register to serialize bytes into a sequence of bits at the sending side [5]. At the receiver side, the ESB uses the UART to combine a series of bits into bytes that are further processed by the CPU at once. In order to use the UART for radio communication, the input pin of the radio transceiver is fed with a sequence of bits by the output pin of the UART.

To support the asynchronous communication, a byte is framed by a preceding start bit (logical 0) and a succeeding stop bit (logical 1). The start bit is represented as a low signal level, the stop bit as a high level. At the receiver, a low level is interpreted as a start bit whenever the UART state machine is idle. Thus in the worst case, the receiver might skip one or more ones until it encounters a falling edge, and it might misinterpret any next zero bit as the start of a byte frame. In this case, the receiver will be out of sync for the entire remaining bit stream.

To avoid an erroneous detection of the start bit, three samples are taken around the middle of a bit. If the start bit is detected successfully, the receiver samples 8 bits and waits for the stop bit. If no stop bit is detected, a frame error occurred. Nevertheless, the received byte is transferred to the UART buffer and reported to the CPU for processing. Afterwards, the UART becomes idle again and waits for the next $1 \rightarrow 0$ transition.

The sender works in a complementary way. Whenever the UART is idle, the next byte waiting for transmission is transferred from the UART buffer to a shift register. Before the byte is serialized, the UART generates the start bit. The start bit, all data bits, and finally the stop bit are then transferred to the radio transceiver that starts the radio transmission.

As we have described above, at the sending side a logical 0 is mapped to a low voltage level and a logical 1 to a high one. At the receiver side, the received baseband signal is converted to a voltage that is linear to the strength of the signal. A voltage being higher than a predefined threshold is interpreted as a logical 1. A voltage being lower than the threshold as a logical 0. According to the current baseband voltage, the radio receiver will tune the threshold, i.e., the threshold might be increased or decreased. Thus it would be best if the threshold is tuned to the middle of the highest logical 1 and the lowest logical 0 level. It is therefore recommended that each byte roughly contains the same amount of ones and zeros. That property is also referred to as *DC-balanced* [7].

To synchronize the receiver side to the beginning of a packet transmission, a *preamble* is sent first. The preamble consists of five 0xAA bytes (which is a sequence of alternating bits) and is followed by a synchronization byte 0xFF (idle line) to synchronize the receiver to the start bit of the next byte. In order to allow the receiver to recognize the start of the packet, two start bytes (0x01 and 0x7F) are used[1]. The idle condition of

[1] A start byte should be distinguishable from other bytes. It therefore consists of two bytes containing a run of zeros followed by a run of ones to keep the transceiver DC-balanced.

the last start byte allows the receiver to simply recognize the next start bit and with that hopefully most of the following bytes.

2.3 Error Characteristics

During preliminary experiments we have observed that the frequency of an error almost monotonously increases with the number of transmitted bytes. Since radio disturbances should actually be independent of the byte's position, we assume that most of these errors are caused by the hardware itself.

Figure 1 shows a sample of received bytes of an erroneous packet that was captured during the experiments. Due to the used Manchester coding, a data byte is encoded into two successive bytes. All bytes that are incorrect are labeled with a gray box. There are two things that should be noted: (1) The wrong byte 0x51 is due to a single bit error within the data bits. The correct byte was 0x59. (2) With the erroneous byte 0xAA, all following bytes until the end of the packet are misinterpreted. Thus we assume that the receiver runs out of synchronization near the end of the packet. If we take a look at the disturbed bytes, we can recover the originally byte stream. However, the byte stream (indicated by a white text color in Figure 1) is shifted to the left by a number of bytes which could not be recovered unless the content of the packet is known a priori.

We can explain these errors as follows: Single bit errors are actually rare and are not limited to data bits only. It is also possible that the start bit, which is used by the UART for the byte synchronization, will be disturbed. However, the loss of a start bit has serious consequences on the following bytes. In the first place, the receiver gets out of sync and the following bits are shifted to the left. Secondly, at the end of a byte, some bits of the next byte might be dropped until the UART receives the next start bit. Hence, it is possible that a single-bit radio disturbance can corrupt an entire packet.

3 Resync Mechanism

As we have seen, most of the occurring errors are due to an unsynchronized UART which lost one or more bits and thus misinterprets data bits as start or stop bits or vice versa. This leads to an unrecoverable misinterpretation of all following data. Thus, the disturbance of start bits is most critical. Since the receiver does not know the transmitted data a priori, it will not be able to re-synchronize to the original data stream by itself. Instead, the sending node should provide a mechanism for an automatic resync.

As we have discussed in Section 2.2, the byte synchronization is performed by means of start and stop bits. Since a start bit will be detected by a falling edge, the best way for a resync is to restart the transmissions with an idle line, i.e., a consecutive run of ones.

Our solution becomes more intuitive if we consider both the sender and the receiver as state machines. As long as both sides are synchronized, a start bit at the sender side is also interpreted as such on the receiver side. The same is true for data- and stop-bits. But once the receiver has missed a bit somewhere, its state machine falls behind. As a consequence, it will read the next start-bit as data-bit and it will take the first falling edge in the data-section of the transmission as a start bit. To resolve this situation, the sender sends eight consecutive high values followed by a falling edge, so that the state machine of the receiver will be in the state "waiting for a falling edge", no matter how

many bits the receiver fell behind before. From that time on, both state machines will be in the state “start-bit detected”.

We can conclude that the UART’s state machine is not very adaptive. It usually changes from a preceding state to a single succeeding state. Only when waiting for a start bit, “no falling edge” means “stay in your current state” while “falling edge” means “start reading bit one”. This is the only opportunity at which we can make the receiver wait until both state machines are synchronous again.

In order the make the receiver’s state machine wait once in a while, we stuff resync bytes (0xFF) into the data stream. 0xFF maps to a long run of high signal values on the channel which can be interpreted as an anchor if the UART becomes out of sync. This kind of anchor has to occur in the data stream periodically because the sender cannot know when the receiver might get out of synch. That way, the receiver will be able to recognize the start of the following byte. No matter how many bits were skipped coincidentally, the UART will detect a falling edge after the sequence of 1-bits (the 0xFF) in any case, no matter how many bits were overheard before.

In order to achieve a resync at the byte level, the following approach is used: Let n be the number of bytes after a resync byte is stuffed into the stream and m be the number of yet received (encoded) data bytes by the receiver. The stuffing starts with the first data byte (after bytes 0x01 and 0x7F). Thus, the receiver expects a resync byte each n bytes, i.e., if

$$m \mod n = 0. \tag{1}$$

Then for each received byte, the receiver checks if it is a resync byte (0xFF). To be more robust against different kinds of radio disturbances, 1-bit errors are introduced intentionally[2]. If a resync byte is detected (or even one with a erroneously inverted bit), the receiver verifies if the byte stream is still synchronous. Otherwise, Equation 1 is false and so is the receiver’s byte position. Since it is likely that the receiver is some bytes behind, it increases its byte position until Equation 1 is satisfied again.

If no resync byte could be recognized but m becomes a multiple of n, the receiver might have missed the resync byte. However, it is likely that the resync byte was considered as a data byte and therefore missed. Hence, the receiver skips the current byte and waits for the next resync byte.

Figure 2 shows a data packet that was transmitted using resync bytes every n-th encoded byte ($n = 4$). Except for two cases, all resync bytes are detected correctly (indicated with a ‘:’). The total amount of byte errors is much lower since the receiver could sooner or later be re-synchronized to the actual data stream in almost every case.

For example consider the sixth row of Figure 2. The resync byte is interpreted as a data byte since (i) the receiver was behind the data stream and (ii) the resync byte contains more than one single-bit error. Since the next byte (0xA5) is skipped due to Equation 1, the three following bytes are shifted to the left by one byte. As a resync byte is recognized after byte 0x56, the receiver re-synchronizes and increases its byte position by one (byte 0x00). Thus, the following bytes are at the right place again.

[2] Due to the encoding scheme presented before, it is still distinguishable from other bytes.

```
01 7F AA 55 AA 55 59 A6 55 AA 96 A6 A9 5A 55 AA
59 6A 5A 5A 96 99 5A AA A6 5A 56 69 66 6A 56 6A
51 96 66 6A AA 69 A5 99 AA 66 69 59 56 6A 56 96
95 69 69 95 95 A9 AA 56 5A 55 A5 A9 99 56 96 95
95 5A 99 55 A9 96 69 55 9A 66 56 99 9A 69 95 AA
A9 AA 66 59 9A 9A 55 AA AA 59 4B 33 33 66 99 65
A9 99 99 59 9A 99 56 A9 56 96 64 9A 99 2B 2B 95
95 66 69 56 5A 69 65 AA 56 56 99 6A 9A A6 61 56
5A 66 66 5A A9 69 95 59 65 65 65 9A A6 45 69 69
65 4B 66 6A 59 99 AA A6 55 99 A5 99 AA 6A A5 A5
69 56 9A 99 A5 66 99 6A 16 59 95 6A 69 89 95 6A
```

Fig. 1. Part of an Erroneously Received Packet

```
01 7F AA 55 AA 55:59 A6 55 AA:96 A6 A9 5A:55 AA
59 6A:5A 5A 96 99:5A AA A6 5A:56 69 36 6A:56 6A
59 96:66 6A AA 69:A5 99 AA 66:69 59 56 6A:56 96
95 69:69 95 95 A9:AA 56 59 5F A9 99 56:00 96 95
95 5A:99 55 A9 96:69 55 9A 66:56 99 9A 69:95 AA
A9 AA:66 59 9A 9A:55 AA A6 69 96 65 9A AA:A9 A6
96 66:99 65 A1 8A:99 59 9A 99:56 A9 56 96:64 9A
99 65:65 AA 95 95:66 66 A5 59:99 A6 A5 A9:69 69
65 AA:56 56 9A A6:61 56 5A 66:66 5A A9 69:95 59
65 65:65 9A A6 45:69 69 59 4B:96 6A 59 99:AA A6
55 99:A5 99 AA 6A:A5 A5 69 56:9A 99 A5 66:99 6A
```

Fig. 2. Part of an Erroneously Received Packet using Resync

4 Evaluation

For an evaluation of our resync approach we performed a comprehensive set of measurements. We placed 16 ESB nodes (including one *source* node) in line on our office floor, each at a distance of 1 m. The source node was powered by a transformer while all other nodes used rechargeable batteries. Thus, we hope to get similar signal strengths for outgoing packets during each evaluation run.

In order to evaluate different transmission powers, the transmission power was varied from 100 down to 0 in step sizes of four. For each transmission power, the source node broadcasts 100 data packets containing 255 bytes[3]. The data rate was set to 2 packets/sec. Nodes receiving a data packet do not send an acknowledgment but quietly update their statistics. Due to implementation issues, the content of each packet was predefined and known to all nodes a priori. In this way, the receiving nodes could simply keep book of the number of erroneously received packets as well as the number of byte errors per packet.

At the end of each evaluation run, the source node collects all statistics stored at the receiving nodes. A notebook connected to the source nodes served as a database in order to support a convenient evaluation. We developed several *code modules* that operates as firmware plug-ins, e.g., for the generating of data packets, the recording of packet statistics, simple statistical calculations and for the collection process of stored statistics. Altogether, we have performed evaluation runs for different resync frequencies n, varying from 0 (which means no resync is employed at all) to 32.

Before we investigate the evaluation results of our resync approach, Figure 3 shows the average number of packets detected and received as well as the number of correctly received packets for different transmission strengths with no resync being performed. As expected, the number of packets increases with an increasing transmission power. However, for a transmission power lower than 16, packets are not received at all or are erroneous all the time. While the difference between detected and received packets is very low, the difference between received and faultless packets is significant. Although the reception rate of faultless packets increases with an increasing transmission power, the amount of erroneous packets is quite high. This is not surprising if we keep in mind that all nodes are placed at different distances.

The influence of the node's distance to the sending node is depicted in Figure 4, averaged over all transmission powers. Although the number of packets tends to decrease with increasing distance, there exists a high variation between adjacent nodes.

[3] Note that no packet forwarding or routing are employed.

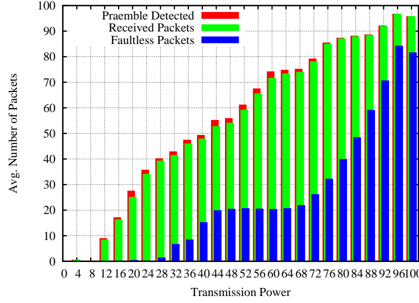

Fig. 3. Average Packet Reception for Different Transmission Powers without Resync ($n = 0$)

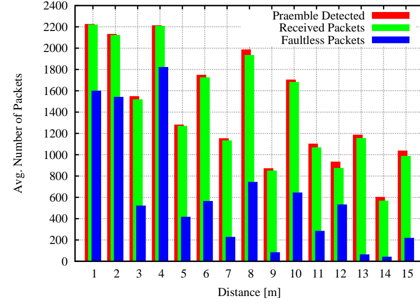

Fig. 4. Average Packet Reception for Different Distances without Resync ($n = 0$)

This might have several causes like signal interference, dispersion, or multi-path fading due to our indoor placement or transceiver calibration errors that are due to the low-cost, low-energy hardware. As other authors have reported [8–10], we believe that the latter is more likely since we have encountered such variations of different nodes very often during preceding tests.

In the following, we will now take a closer look at the improvements resync is able to achieve. Figure 5 depicts the relative frequency with which an error occurred at a specific byte position of a packet. Due to the fact that some bits get lost if the UART misses a start bit (as it was discussed in Section 2.3), it is likely that bytes near the end of a packet get disturbed. This assumption is proven by Figure 5 where the error frequency heavily increases with the number of transmitted bytes per packet if no resync is used.

Interestingly, there are some peaks where the error frequency breaks down. For example, at the byte position 81 and 83 the error frequency is lower in a very small interval. How can the probability of a single byte for being false be lower in a stream of many bytes? If it was caused by an implicit resync, the following bytes would experience fewer errors, too. We concluded that these peaks are due to the packet content itself. We performed a similar evaluation run with 50 zero bytes added to the beginning of the data stream. As assumed, the peaks moved to the right by exactly 50 bytes.

A deeper analyzes shows the following circumstance: Beginning with byte number 81, the data stream contains the following (not encoded) bytes: 0xF4, 0xC6, 0xF4, 0x2B, 0x2F, 0x56 etc. As we see, 0xF4 recurs two bytes later. So if the stream shifts to the left by two bytes, byte number 81 would be correct coincidentally. For the second 0xF4 byte we also find a similar bit pattern, the end of byte 0x2F and the start of 0x56 (except for the last bit). Even if the receiver is out of sync and hence start and stop bits are considered as data bits, the pattern remains the same. Thus, the effect we observed is simply due to the packet content itself, which we have chosen at random.

As Figure 5 proofs, significantly better results are achievable using resync as proposed in Section 3. However, particularly in terms of efficiency, it is essential to analyze how often resync bytes should be used for optimality. The best results are achieved for $n = 16$, followed by $n = 32$, $n = 8$, $n = 4$, and $n = 2$. Thus we can see that resync improves the error characteristics in all cases. But the improvement is not monotonic with a decreasing n. In fact, the results for $n = 64$ were worse than those for $n = 32$.

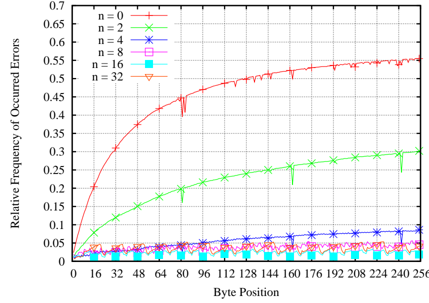

Fig. 5. Number of Errors Occurred at a Specific Byte Position

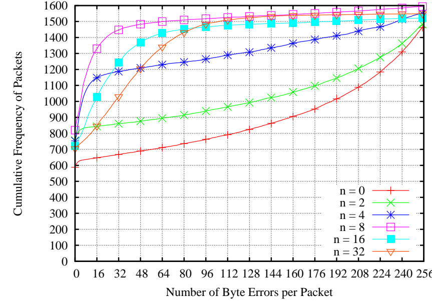

Fig. 6. Cumulative Distribution of Byte Errors per Packet

That is due to the following trade-off: For $n \to \infty$, resync bytes are used sparsely so that they might be effectless. On the other hand, for $n \to 1$, resync bytes are used too often so that the byte stream will be DC unbalanced. As discussed in Section 2.2, the threshold used by the UART to distinguish between a high and a low signal is adapted to the ongoing transmission. That is, if too many ones are transmitted due to too many resync bytes, the threshold increases and succeeding ones might be misinterpreted as zeros. Consequently, resync bytes might no longer be detectable and thus be ignored.

In order to give an expression on how many byte errors occurred per packet, Figure 6 depicts the cumulative error frequencies vs. the number of byte errors per packet. In all cases, using resync increases the number of faultlessly received packets. The best result is achieved for $n = 8$ with about 30% more correct packets than in the case of $n = 0$. For some nodes, even ten times more packets are received without an error.

Concerning the number of errors per packet, for $n = 0$ most of the errors occur at the end of a packet. Thus, almost the entire packet is corrupted. The same applies to $n = 2$. However, for $n > 2$ we can observe how the error characteristics changes. While for $n = 0$ about 56% of all received packets have more than 16 errors, there are only 25% with more errors for $n = 4$. For $n = 8$, it is even better. Only 17% of all received packets have more than 16 errors. However, for $n > 8$ the benefit of a resync decreases since the number of skipped bytes becomes too high if a resync occurs. Remember, the resync mechanism might skip some bytes due to Equation 1 if the receiving UART is fallen behind[4]. Thus, for $n = 16$ and $n = 32$, too many errors occurs due to the (even though successful) resync.

So far we did not take the communication efficiency or the packet throughput defined as the ratio between used bandwidth and costs into account. The used bandwidth relates to the number of bytes at a certain error rate as shown in Figure 6. The costs are influences by the frequency a resync is performed because every resync byte is lost for carrying valuable information[5]. According to Figure 6, Figure 7 shows the cumulative throughput for an increasing number of byte errors. Resync with $n = 2$ performs worst with respect to the packet throughput since the number of resync bytes is quite high. It thus performs even worse than the case of using no resync. For up to 58 errors per

[4] In the worst case, there are $\lceil \frac{n-1}{2} \rceil$ skipped bytes.

[5] In general, the efficiency will be $1/n$ for a single packet.

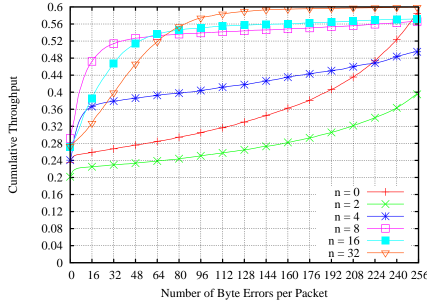

Fig. 7. Cumulative Throughput vs. Number of Byte Errors

n	Received Packets and (Throughput) per Node with up to e Errors					
	$e = 0$		$e \leq 16$		$e \leq 32$	
0	588	(0.24)	648	(0.26)	668	(0.26)
2	756	(0.20)	844	(0.22)	861	(0.22)
4	754	(0.24)	1148	(0.36)	1188	(0.38)
8	821	(0.30)	1330	(0.48)	1448	(0.52)
16	722	(0.28)	1026	(0.38)	1243	(0.46)
32	706	(0.28)	846	(0.32)	1030	(0.40)

Table 1. Error Characteristics for Different Resync Schemes

packet, $n = 8$ achieves the best results. It is then outperformed by $n = 16$ and later by $n = 32$ since both schemes need fewer resync bytes and are therefore more efficient. However, we can conclude that up to a reasonable number of errors, which might be corrected by appropriate Forward Error Correction (FEC) codes [11, 12], $n = 8$ trades off the bandwidth usage and required resync costs best. The main results are once again summarized in Table 1.

5 Conclusion

Due to the loss of one or several bits, the state machine of the receiver might get out of sync such that following bits are misinterpreted. We have proposed a resync mechanism that is able to resync both the sender and the receiver state machine, and in this way reduces the number of byte errors per packet significantly. In addition, the resync mechanism enables the usage of Forward Error Correction codes which would further increase the delivery rate. We leave this issue for future work.

References

1. Willig, A., Mitschke, R.: Results of Bit Error Measurements with Sensor Nodes and Casuistic Consequences for Design of Energy-Efficient Error Control Schemes. In: Proceedings of EWSN, Zurich, Switzerland (2006)
2. Schiller, J., Liers, A., Ritter, H., Winter, R., Voigt, T.: ScatterWeb - Low Power Sensor Nodes and Energy Aware Routing. In: Proceedings of HICSS, Hawaii, USA (2005)
3. The ScatterWeb Project: (http://www.scatterweb.com/)
4. Texas Instruments: (MSP430x1xx Family User's Guide)
5. Bierl, L.: Das große MSP430 Praxisbuch (in German). Franzis Verlag GmbH (2004)
6. RFM: (868.35 MHz Hybrid Transceiver TR1001)
7. RFM: (ASH Transceiver Designer's Guide)
8. Zhao, J., Govindan, R.: Understanding Packet Delivery Performance in Dense Wireless Sensor Networks. In: Proceedings of SenSys, Los Angeles, CA, USA (2003)
9. Zuniga, M., Krishnamachari, B.: Analyzing the Transitional Region in Low Power Wireless Links. In: Proceedings of Secon, Santa Clara, CA, USA (2004)
10. Zhou, G., He, T., Krishnamurthy, S., Stankovic, J.A.: Impact of Radio Irregularity on Wireless Sensor Networks. In: Proceedings of MobiSys, Boston, MA, USA (2004)
11. Lin, S., Costello, D.J.: Error Control Coding: Fundamentals and Applications. Prentice Hall (1983)
12. Peterson, W.W., Weldon, E.J.: Error-Correcting Codes. MIT Press (1972)