A communication protocol

Előd Csirmaz László Csirmaz*

The most frequently used device for exchanging data between computers is the floppy disc. This is especially true for portable machines and laptops. Keeping the data on the portable device and on the base desktop machine up to date – to synchronize them – is tiresome, to say the least. You insert the floppy into the laptop, copy the data, floppy out, floppy into the desktop, copy, floppy out and into the laptop again, endlessly. Of course, this works only if there is no data error, and the files are not too large. Using a cable this can be done much easier. There are hundreds of excellent utilities which use a serial cable to perform the data exchange. Unfortunately, the new machines have no old-fashioned serial ports anymore, thus these programs cannot be used either. Nevertheless, they still have the parallel-port connection which usually used for printing. Can this facility be used for communication between two machines? If yes, will it be reliable? What the communication speed will be? We have made a trial connection cable, and wrote the communication programs. Designing the communication protocol was the most interesting part of the whole project. We hope that this short account will be useful for those who are interested in how protocols are designed.

1 Connection cable

The cable connects the parallel ports of two machines. Originally the port was designed for one-way communication: by assumption the computer sends data to a printer through this port. During data transmission eight bits are sent simultaneously; this feature gave the name, as data is transmitted parallelly, as opposed to serial transmission, when bits are sent one by one. The receiving device provides a feedback through a few pins. The connectors have 25 pins. Out of them 12 pins hold output signals: eight for data, and four for control. Input, or feedback signals use 5 pins, the rest of the pins is grounded. For the connection we used a cable of 5 wires. One out of the five was used as ground, two for communication in one direction, and two wires for communication in the other direction. The cable is symmetric, we put 25-pin connectors at both end, and had a small box full of resistors at the middle of the cable. The general layout is sketched on figure 1. Resistors R_A are there for safety reasons only; the electronics inside the machines should be safe even in case of a short-cut. Resistors R_B are there to make sure that the pins are not “floating” even if there is no connection.

*Central European University, Budapest

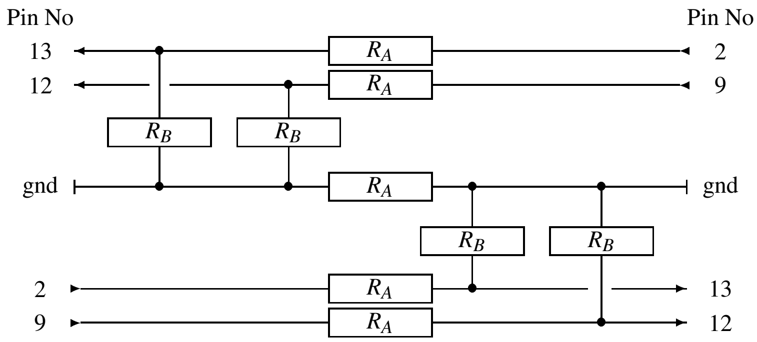

Figure 1: The cable

Parallel ports are handled by writing and reading data from three consecutive port addresses. These addresses will be denoted by $b+0$, $b+1$ and $b+2$, respectively. The standard value for b is 378h in hex, or 888 in decimal; sometimes the value 4BDh (or 956 in decimal) is used as well. Bits written to the port $b+0$ appear on pins 2–9. Values of pins 13 and 12 can be recovered by reading from port $b+1$; the fifth and sixth bit (corresponding to 16 and 32) gives them. Thus our cable is used as follows: sending out `00`, `01`, `10`, or `11`, one must write 0, 1, 128, or 129, respectively to port address $b+0$. Reading from port $b+1$ and masking the value by $(16+32)$, these values appear as 0, 16, 32, and 48. Consequently the cable can transmit two bits in both directions simultaneously.

Thus we had the cable, and wanted to use it to transfer data. We have faced two problems:

- How the machines will recognize that they are connected and can start transmitting data, even if we don't know which machine will start earlier; and
- How to send the data? How the sender and receiver should work together? How to make the transfer fast enough, but still allow for error detection?

Our attempts to solve these problems are described in the subsequent sections.

2 Establishing connection

Before starting data transmission, both machines must check that the corresponding communication program is also running on the other side. This, however, is not enough: the programs must agree that the communication can be started, and have a consensus on who starts the dialog. We designed our protocol in a way that it should work under the following conditions:

- the communication program can be started first on any of the machines;
- when starting the program, the input bits on the cable can be anything;
- do not make any assumption on the speed of the machines.

2.1 First attempt

Our first protocol is symmetric, which means that both parties do the same thing. The protocol has the drawback that the program must know the initial state of the two output bits it has command of. This information might be recovered by reading from the port $b+0$, but this not always works. At the description of the protocol the participants are *Alice* and *Bob*, as usual.

1. read in both the input and the output bits;
2. change the output bits as follows: if it was `01` then change it to `10`, otherwise change it to `01`;
3. wait until the input bits change from the ones read at step 1;
4. write out `00`;
5. wait until the input bits become `00`.

Protocol 1: Symmetric handshaking

Suppose both machines are running protocol 1. After starting, both of them are waiting in step 3 for the other to change the bits on the input pins. As the protocol is symmetric, we may assume that Alice starts sooner. She sets her output either to `01` or to `10`, and in waits in step 3 for her input to change. Meanwhile Bob starts the protocol, too. He also reads his input, where he finds either the value what Alice set in step 2, or the value which was there before that; after that he changes his output. Alice notices this change and she writes out `00` in step 4. Bob waits for his input to change. Now it changes, so Bob also advances in the protocol, and writes out `00`. Alice is waiting for `00` in step 5, it arrives, thus Alice finishes the protocol. Bob in step 5 checks his input, which is also `00`, and Bob finishes the protocol, too. At this point both participants know that the connections has been established, and the communication may start.

The protocol works fine even if Alice and Bob starts exactly at the same moment, or if one of them is much slower than the other. Both of them advances from step 3 only when the other party executed either the second or the fourth step – thus know that the other works as well. Alice and Bob completes the protocol almost simultaneously: the difference is the time necessary to read the input and check it.

2.2 How to improve it?

During our first protocol 1 both participants were waiting for the other to change his (or her) output. This can be done only if they know what their output was. We would like to improve the protocol so that they should not know this information. The idea is that Alice relates the input what she sees and the output what she controls in a certain way, and Bob does the same. We choose two relations so that they cannot hold simultaneously, thus one of them should change the output, which will be recognized by the other. The improved protocol 2 is symmetric, too. Both parties use the value `00` to signal that the connection has been established.

Both parties set their output to a value different from `00` in step 2, and it is set to `00` (in step 4 or 6) only after the input changed. That is, when they are convinced that they

1. read the input; let us denote it by xx;
2. depending on xx determine the output bits as follows:

xx:	00	01	10	11
output:	01	10	11	01

3. wait until the input changes compared to xx
4. let the new input be yy. If yy=00, then write out 00, and continue the protocol at step 7. If yy differs from this, then, depending on yy and on the the previous output, determine the next output as follows:

Previous output:		01	10	11
yy:	01	10	01	01
	10	10	11	10
	11	11	11	01

5. wait again until the input changes compared to the one read in step 3;
6. write out 00;
7. wait until the input becomes 00.

Protocol 2: Improved handshaking

can communicate with the other party. Thus if the protocol stops, both participant is prepared, and they know this fact about their partner, too. Thus we need to check only that the protocol will eventually stop.

It cannot happen that both Alice and Bob are waiting at step 3. Indeed, if Alice is waiting here, then she set her output bits at step two, thus those must be 01, 10, or 11. Similarly, Bob also set his output sometimes earlier to one of 01, 10, or 11. Consequently Alice read one of these values in step 1, as her input bits did not change since then. What Alice sees determines what she set in step two, this is put in the left hand side of this table.

Alice sees:	01	10	11	Bob sees:	10	11	01
Alice sets:	10	11	01	Bob sets:	11	01	10

The same is true for Bob, which is shown on the right hand side. But this situation is impossible, as Alice sees what Bob set, and Bob sees what Alice set. If, for example, Alice sees 01, then she set 10, and then Bob sees this, and responds by 11– and not by 01. Thus it cannot happen that both of them are waiting in step 3, one of them will go further, say it is Alice.

The value what Alice reads in in step 3 must be the value what Bob wrote on step 2 – as we assumed that Bob is not over step 3 yet. Nevertheless Bob has started the protocol, and in the first step he read either the original value on the wire, or the value what Alice wrote in the second step. In the fourth step Alice chooses a value which makes Bob's input change in both cases. Thus Bob – sooner or later – will recognize that his input changed, and continues at step 4. The value `yy` what he has read is either the value Alice wrote just right now, or it could also be Alice's output from step 2. Nevertheless, Bob changes his output. Alice waits for this in step 5, and writes `00` – changing Bob's input once again. Bob recognizes the change, writes `00`, and then the protocol ends.

Our analysis shows that in step 4 the input cannot be `00`, thus we do not need to handle this case. Also, Alice and Bob finishes the protocol about the same time, similarly to the previous protocol 1.

Step 5 can be executed first by either Alice or Bob, and who is the latter one sees `00` as his (or her) input, does not wait at step 7, and finishes the protocol. But who will it be depends on several random factors, such as starting time, speed of machines, how the steps of the two executions overlap. Nevertheless we do need that the one who sets first the output to zero is always Alice, and then Bob responds by setting his output to zero as well. To achieve this we have to break the symmetry of the protocol as follows.

1–5. same as in protocol 2;
6. let the new input be `zz`. If `zz` is not zero, then go to step 11, otherwise continue at step 7;
7. output `00`;
8. wait until the input becomes `11`;
9. output `11`;
10. wait until the input becomes `01`;
11. output `00`;
12. wait until the input becomes `00`.

Alice's steps

1–5. same as in protocol 2;
6. let the new input be `zz`. If `zz=00`, then go to step 13, otherwise continue at step 7;
7. output `00`;
8. wait until the input becomes `00`;
9. output `11`;
10. wait until the input becomes `11`;
11. output `01`;
12. wait until the input becomes `00`;
13. output `00`.

Bob's steps

Protocol 3: Improved handshaking

If Alice's input at step 5 is not `00`, then she set her output to `00` first, and then the protocol can be finished in a straightforward way. If she reads `00` then Bob is ahead of her, and in step 8 he waits for Alice to write out `00`. She does it in step 7, for which Bob responds by `11`. This is followed by Alice's `11`, then Bob's `01`. Now Alice outputs `00`, and Bob answer this in step 13 by writing `00`. We plot the events on a time-scale what happens when Bob reads a value different from `00` in step 6:

step:	4.	5.	6.	7.	8.	9.	10.	11.	12.
Alice:	`zz`	`..`	`..`	`00`	`..`	`11`	`..`	`00`	`..`
Bob:	`..`	`..`	`00`	`..`	`11`	`..`	`01`	`..`	`00`
step:	5.	6.	7	8.	9.	10.	11.	12.	13.

3 Data transmission

Sending data can be started as soon as the connection has been established. In protocol design, we aimed one-way transmission only. In all cases the sender sends a complete file, and after the connection will be disconnected. We do not have any special procedure for breaking the connection, simply quit the communication program. As always Alice wants to send data to Bob, we should not handle the case when both parties start the sending protocol simultaneously, or within a very short time frame.

3.1 Sending a bit

When the connection has been established, in both directions the wire contains `00`. When Alice wants to send something, she changes her value. Bob recognizes the change, and using his wires, tells Alice that the data has arrived. Now Alice reset her data to `00` and waits until Bob does the same. During such a cycle Alice can send three different values depending on how she changes her output. The time flow is depicted on figure 4. Here `xx` is one of `01`, `10`, or `11`; Bob signals receiving data by writing `11` to the output. Using this protocol we can send bits – for example `01` means

Alice:	`00`	`..`	`xx`	`..`	`00`	`..`
Bob:	`..`	`00`	`..`	`11`	`..`	`00`

Protocol 4: Sending a bit

0, while `10` means 1. The complete file should be rearranged from the received bits by Bob.

3.2 A full byte at once

Protocols sending a full byte at a single round are better suited for file transmission. We can design one by modifying the previous bit sending protocol. When the protocol starts, both wires contain `00`. There are eight bits in a byte, a bit is encoded by either `01` or `10`. Alice writes the value corresponding to the first bit, then waits for Bob to say 'I've received it'. Alice responds by writing `11` which tells Bob she is willing to send the next bit. Bob responds by changing his output for ‘I am ready for it’. They do it for the eight bits of the byte to be transmitted. After that Alice sends a parity check bit, namely a value which tells Bob whether the byte contains an even or odd number of `1`’s. After receiving the parity check bit, Bob says either ‘everything is OK’, or ‘there were errors’. Finally both Alice and Bob reset the wire to `00`, and they can restart the cycle with the next byte.

			1. bit				2. bit					8. bit				parity			
			↓				↓					↓				↓			
Alice:	00	..	bb	..	11	..	bb	..	11	..	...	bb	..	11	..	bb	..	00	..
Bob:	00	..	..	01	..	10	..	01	..	10	...	..	01	..	10	..	xx	..	00
																	↑		
																	OK/wrong		

Protocol 5: Sending a byte

When we programmed this protocol, we made the mistake to encode Bob’s OK message by `00`. We though this trick shortens the total time of a cycle. Indeed, the final `00` is there automatically, and should not be sent separately. We spent hours on figuring out what went wrong. Let’s see what happens between two cycles:

			parity				1. bit from next byte
			↓				↓
Alice:	11	..	bb	..	00	..	bb
Bob:	..	10	..	00	..	..	..
				↑			
				OK			

Alice set the parity bit, and Bob signalled by `00` that the parity is fine. Alice reset the line to `00`, and started sending the next byte. First, she checked whether her input is `00` (it was), thus she wrote to the line the value of the first bit of the next byte. Meanwhile Bob was in the process of completing the previous cycle, namely he was waiting for `00` to arrive from Alice. As this `00` was present only for a very short time, Bob missed it. Thus Bob was still waiting for `00`, and Alice was waiting for Bob’s response `01` saying that he received the first bit. When we printed tracing information, everything went smoothly, as this time Bob has had enough time to receive Alice’s signal.

This phenomenon explains why the handshaking protocol should conclude by setting the line to `00` by Alice first, while Bob’s `00` is the answer to it. Only this way can we ensure that the Bob won’t mistake the first bit of the first byte for an element of the handshaking protocol.

3.3 Doing it faster

Protocol 5 worked perfectly, however it was quite slow. Can it be done faster? The protocol transmits bits as `01` and `10`, and uses `11` as separator between the bits. Why do we need this separator? As the other party observes only the change on the line, the output must change in each step. But no matter what is on the line, it can change in more than one way. Can we use this multitude of possibilities to transmit a bit at each step? It seems so, this is exactly what we were coming up with.

First Alice sends the first bit of the byte in the usual way, encoding as `01` or `10`. If, say, the bit was encoded as `01`, then the next value can be either `10` or `11`; if it was `10`, then Alice can choose between `11` and `01`. This way Alice can tell Bob what the second bit is. When Alice sends `11`, then the next bit is encoded again by `01` or `10`. The complete protocol looks like this:

Alice:	`00 .. b1 .. b2 .. b3 .. b4 .. b5 .. b6 .. b7 .. b8 .. b9 .. 00 ..`
Bob:	`.. 00 .. 10 .. 01 .. 10 .. 01 .. 10 .. 01 .. 10 .. 01 .. 1p .. 00`

Protocol 6: Sending a byte faster

Alice determines the values `b1` though `b9` as follows. The first value is either `01` or `10` depending on whether the first bit of the byte is 0 or 1. Subsequent values are computed from the bit to be sent, and the value on the wire using the following table:

value on the line:	`01`	`01`	`10`	`10`	`11`	`11`
next bit:	0	1	0	1	0	1
next value to send:	`10`	`11`	`11`	`01`	`01`	`10`

Alice can easily compute the value to be put on the wire: she adds the next bit the value present on the wire, takes the remainder when divided by 3, and adds one. To recover the bit, Bob can use the table above, or perform the following computation: subtract the previous value from 2, add the present value, and take the remainder when divided by 3.

The ninth bit `b9` is the parity check bit, and Bob's answer is `10` if the parity bit was fine, and he sends `11` otherwise. In this latter case Alice resends the last byte.

If Alice receives an erroneous value, she can abort the cycle by setting her output to `00`. Bob recognizes this, he also writes `00` to his wire, and the cycle can be restarted. In the case Bob receives a wrong value, if this value is `00`, he also writes `00`, and then waits for the byte again. If the wrong value is something else, then his next output will be `11`, to which Alice must respond by `00`. Bob sets his line to `00`, and the cycle is started over again.

3.4 The fastest version

The last version is the fastest among our protocols. This version relies more on the correctness of the cable; in return it is almost twice as fast as the protocol 6. This version differs from it in two significant details. First, in each step Alice uses all three possibilities, not just two; second, we let Bob to send the parity check bit as he can do it without an extra step.

During two steps Alice can send nine different configurations. We use eight of them to code three bits, and use the ninth to signal that the previous parity bit sent by Bob was wrong. Alice can send three bits in two steps, and 24 bits in sixteen steps.

After sixteen steps Alice and Bob takes a short break, before that Bob sends back Alice the parity check bit. The break and the parity check ensures that Alice and Bob are still together, they do not lost synchronization.

Before starting the cycle of sixteen steps, Alice's output is either `00` or `11`, and Bob's output is `00`. The values on the lines are changing as follows:

Alice:	xx .. d1 .. d2 .. d3 .. d4 .. .. d15 .. d16 .. yy ..
Bob:	.. 00 .. 01 .. 10 .. 01 .. 10 ... 10 .. 01 .. 1p .. 00

Protocol 7: Sending without error

As we have remarked above, `xx` is either `00` or `11`. When Alice wants to send the next 0, 1, or 2 value, she computes her output `di` as follows: she adds one to her previous output, then adds the value to be transmitted, finally takes the remainder when divided by 4. Or she may look up the value in this table:

previous output:	00	00	00	01	01	01	10	10	10	11	11	11
value to transmit:	0	1	2	0	1	2	0	1	2	0	1	2
new output:	01	10	11	10	11	00	11	00	01	00	01	10

Alice chooses the closing `yy` to be `00` whenever possible, that is when `d16` is not `00`; otherwise it will be `11`. Bob computes the parity bit and sends it back to Alice: if the parity bit is zero, he sends `10`, if it is one, then he sends `11`.

Whenever Alice discovers some problem, namely if the parity bit is in error, or receives an input from Bob which is impossible by the protocol, she transmits value 2 until Bob sets his line to `00`. Sending two consecutive 2's is invalid, as it codes $2 \cdot 3 + 2 = 8$, which is not a 3-bit binary number. When Bob receives two consecutive 2's, he considers it as the message from Alice that cycle should be aborted. As the error might occur in the previous transmission of 24 bits, not the actual transmission should be restarted, but the previous one.

If Bob sees some problem with the execution, he sets his output to `11`. Alice must respond by sending either `00` or `11`. Then Bob writes `00`, by which he instructs Alice to restart sending the previous 24 bits.

Alice:	xx	..	d1	..	d2	..	yy	..	
Bob:	..	00	..	01	..	11	..	00	
					↑				
					Bob discovers and error here				

Using protocol 7 in a cycle Alice changes her output 17 times, and she transmits three bytes. Protocol 6 requires 30 changes in Alice's output for transmitting three bytes, thus the increase in speed is almost twofold.

4 Putting ideas into practice

We manufactured the cable, and wrote the communication programs. The communication was excellent between machines of different type and speed. Between a Pentium

I and a Pentium II machine protocol 6 transmitted 7 kilobytes in each second. Using protocol 7, according to our expectations, the transfer rate raised to 13 kbyte/second. We did not experience any error during transmission of several megabytes. Programs were first implementated in BASIC, the final ones were written in C. We tried them under DOS and Windows operating systems; the programs run under Linux as well.

Our protocols share the special feature that they do not contain any timing. Thus we did not need to deal with the case when one of the programs is suspended. (This happened indeed, when during a longer transmission the screen saver come up. Touching the mouse, the communication continued with no problem at all.) The lack of timing made our protocols simpler, and more transparent, too.

At the end of the project, we were surprised by the difficulty and the beauty of the solution of the simple task we set forth before ourselves: design a protocol which transmits files in one direction using only two bits in both directions. Even relatively simple ideas turned out to be quite efficient as the initial transmission rate of 60 bytes per second went up to 13 kbyte per second. The stability and reliability of our protocols are also noteworthy.