Asymmetric Alternating Bit Protocol

April 2004

1 Introduction

Alternating Bit Protocol is well known. But, it is a symmetric protocol, for two participants of a same rank/level.

The protocol describet here is an asymmetric variant of the alternating bit protocol. It is to be used when one of the participants is polling the other. That participant is usually called the **master**. Thus we need to examine both the master and the slave side on the communication bus.

One master can communicate with several slaves over the same bus. For each slave, the master has a separate protocol instance.

In the following description, the check for correct messages transfer is assumed to be done via CRC, but, that is just a common use case. The protocol does not define what method will be used for this check - for example, the Internet Checksum is an option.

2 From the master viewpoint

When polling the slave for the first time, the master sets the FIB bit to 1.

Whenever it's polling the slave, the master checks if there are any messages to send to said slave and sends one if there are. If there are not, master will send a fill-in (empty) message. In any case, it sends the FIB bit as intended.

After transmitting, the master awaits a response from the slave.

If the response times out, or it arrives with an error (CRC error, foremost), FIB bit will stay "as it were" and message sent will not be released. Thus, it would be sent again on the next poll for this slave. Of course, the message with an error is ignored.

If the response arrives, master checks the received FIB bit against the sent one. If they are the same, it is as if the response timed out for the purposes of this protocol (for diagnostics, the slave is "live").

If the received FIB bit is different (inverted/alternated) from the sent one, then master assumes that the message was received correctly on the slave side and the message sent will be released. Master will then invert the FIB bit for this slave, to prepare for the next poll.

Thus ends a poll for one slave and the master goes on to the next one. Once all other slaves are polled, master will cycle back to this one.

For diagnostic purposes, if a slave does not respond to the poll several times in a row, it is deemed "down".

3 From the slave viewpoint

During initialization, the slave sets its FIB bit to 1.

If it received a message correctly (CRC check good), slave checks if the received FIB bit is the same as or different then its own FIB bit.

If the FIB is as expected, this means that the received message is a new one and that our message (from the last poll) had been accepted. The slave releases that message from the last poll and inverts/alternates the FIB bit. Then it sends, in response to the poll, its own message, or a fill-in (empty) message with such FIB bit.

If the FIB is different from expected, this means that the master did not receive the response correctly on the last poll. The just received message is ignored and the slave sends the same message as in the last poll, keeping the same FIB bit as in that last poll.

If the message is received with a CRC error, the handling is similar - slave sends the same message as in the last poll, with the same FIB bit (and of course ignores the message with CRC error).

4 Discussion

If the slave sets FIB to 0 during initialization, protocol will suffer the loss of the very first message. But, this need not be the case in practice, because the initialization of the slave is asynchronous to the initialization of the master. Thus, the actual setting of the FIB to 1 during slave initialization is not a 100% guarantee that the first message would not get lost.

Therefore, when the master does the first poll, it should send an empty (fill-in) message, so there are no qualms about losing it.

Unfortunately, even that will not solve *every possible case*. Since the initialization can take a lot of time and during it a lot of stuff is done, it is still possible that timing is such that a messages gets lost. Solving that would be possible, but it is assumed that the slave does not get initialized often, so it's not worth the effort.

5 SPIN model of the protocol

We present the Promela source code for the SPIN model of this protocol. It should be saved to a seperate file, say `FIB_BIT_KOM` (Promela files usually don't have an extension). If such file is "run through" SPIN model checker, it will show that the protocol is sound.

This source code is written in the SRCE system "lingo", thus KOP is the master and RP is the slave in the context of the protocol we describe in this document.

Model has three processes:

1. KOP - the master
2. RP - the slave (only one, as it is the same for all and any RP)
3. communication bus, there to simulate communication errors

SPIN model checker can establish that the protocol is transferring messages correctly (in order) and it is not deadlocking. To avoid having SPIN report an infinitely bad communication bus (which loses *all* messages) as a protocol deadlock, we have labeled the apropriate lines of code with `progress...` labels.

```
/** Max number of messages, must be more than two. */
#define MAX   10

/** Helper symbolic definition of "bad CRC" */
#define BAD_CRC  0

/** Helper symbolic definition of "correct/good CRC" */
#define GOOD_CRC 1

/** KOP, the master, is polling RPs, the slaves on the bus. For
    this model, we need only one RP, as it is the same for all.
    @param in Channel on which KOP receives messages
    @param out Channel on which KOP sends messages
    */
proctype kop(chan in, out)
{
    byte by; /* Message to send - for our test, a number */
    bit fib; /* "Our" FIB */
    bit fib_prim; /* FIB in the received response message */
    byte by_prim; /* The received response message */

    by = MAX - 1; /* In the first iteration becomes 0 */
    fib = 1;

    do
    :: by = (by + 1) % MAX;
poll:
        out!fib, by, GOOD_CRC;

        if
        :: timeout -> goto poll /* no response */
        :: in?fib_prim, by_prim, BAD_CRC -> goto poll
        :: in?fib_prim, by_prim, GOOD_CRC ->
            if
            :: (fib == fib_prim) -> goto poll
            :: (fib != fib_prim) ->
                assert(by_prim == (by + 1) % MAX);
                goto progress
            fi
        fi;
progress:
        fib = !fib
    od
}
```

```
/** RP process on the bus. RP responds to polling. This model is
    not completely accurate, RP has a message queue, thus the
    response can't be to the just received message. But, that is
    not of the essence here, messages here are just helper
    counters to check the validity of the protocol.
    @param in Channel on which RP receives messages
    @param out Channel on which RP sends messages
    */
proctype rp(chan in, out)
{
    bit fib; /* "Our" FIB */
    byte by; /* Message to receive - for our test, a number */
    bit fib_prim; /* FIB of the received message */
    byte by_prim; /* The received message */

    fib = 1;
    by = 0;

    do
    :: in?fib_prim,by_prim,BAD_CRC -> out!fib,by,GOOD_CRC
    :: in?fib_prim,by_prim,GOOD_CRC ->
        if
        :: (fib_prim != fib) -> out!fib,by,GOOD_CRC
        :: (fib_prim == fib) ->
            assert(by_prim == by);
            by = (by + 1) % MAX;
progress:   fib = !fib;
            out!fib,by,GOOD_CRC;
        fi
    od
}


/** The communication bus. It introduces random errors,
    either message loss or message corruption which causes
    bad CRC check. Of course, this doesn't simulate the
    errors that CRC would not catch, but this model does
    not define the actual (CRC) check, so it's just up to
    the implementer to use the best possible check to catch
    the most amount of errors.

    @param kop_in Channel on which KOP receives messages
    @param kop_out Channel on which KOP sends messages
    @param rp_in Channel on which RP receives messages
    @param rp_out Channel on which RP sends messages
    */
proctype commbus(chan kop_in, kop_out, rp_in, rp_out)
{
    bit fib;
```

```
    byte by;
    bit crc;

    do
    ::kop_out?fib,by,crc ->
        if
        ::rp_in!fib,by,GOOD_CRC
        ::rp_in!fib,by,BAD_CRC; progress_err1:skip;
        ::skip; progress_err1:skip;
        fi
    ::rp_out?fib,by,crc ->
        if
        ::kop_in!fib,by,GOOD_CRC
        ::kop_in!fib,by,BAD_CRC; progress_err2:skip;
        ::skip; progress_err2:skip;
        fi
    od
}


init {
    chan kop_tx = [1] of {bit, byte, bit};
    chan rx_kop = [1] of {bit, byte, bit};
    chan rp_tx = [1] of {bit, byte, bit};
    chan rx_rp = [1] of {bit, byte, bit};

    atomic {
        run kop(rx_kop, kop_tx);
        run commbus(rx_kop, kop_tx, rx_rp, rp_tx);
        run rp(rx_rp, rp_tx);
    }
}
```