```
/*
 * To find Dynamic Shift count feature's valid combination type
 */
```

We already have valid four types of packets when not combined and which can be received by an end during Dynamic Shift Count exchange. These are:
01. REQ
02. RSP_NOK
03. RSP_OK
04. WIN

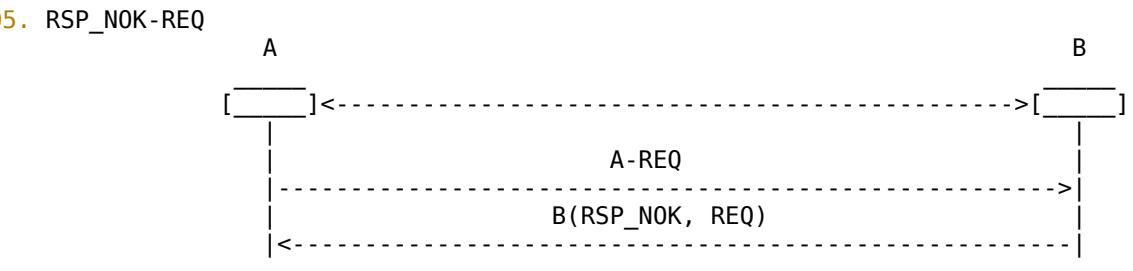
Since any end can raise a request **while** still a Dynamic Shift Count Exchange process in process, we need to find out which combinations are feasible under the constraint that during an exchange only single request can initiated by either end. This condition is ensured by variables flags namely SND_SHF_REQ and RCV_SHF_REQ. These variables are used as flags to ensure that there is not more than one pending REQ accepted or generated thus serializing Dynamic shift count exchange process w.r.t either end.

SND_SHF_REQ !=**NULL**, then new TCP stack cannot accept a REQ
RCV_SHF_REQ !=**NULL**, then new TCP stack cannot generate a REQ

Using these core types we can get total combinations = 4C1 + 4C2 + 4C3 + 4C4 = 15

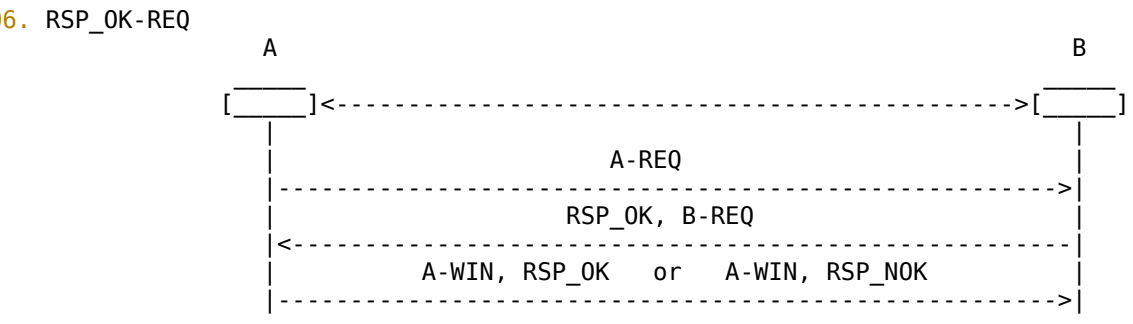
==========
We explore all these cases below except **for** the 4 basic core types which are valid :
==========

05. RSP_NOK-REQ



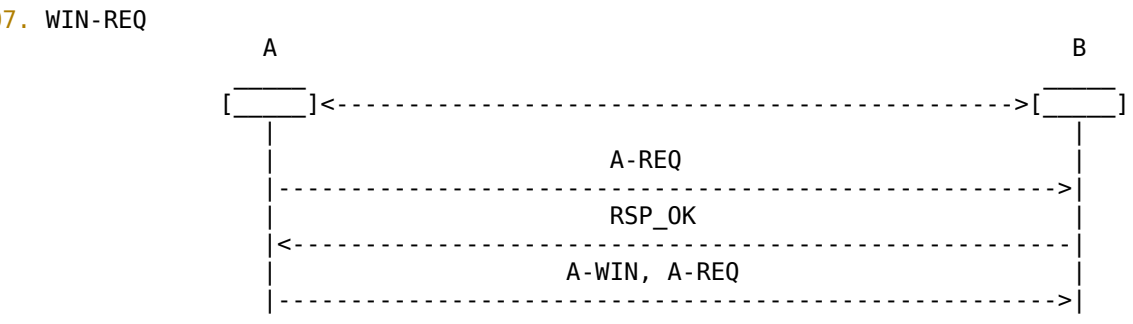
Result: Cannot combine since EXP value will conflict.

06. RSP_OK-REQ



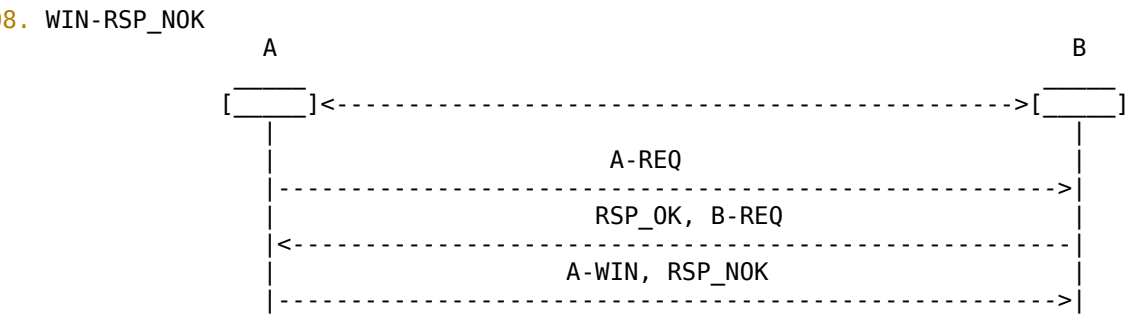
Result: By definition & use of EXP flag combine the request and response will conflict.
It MAY combine **if** the REQ shift count >= RSP shift count.
Design decision is to combine and send **if** it meets condition RCV_SHF_REQ >= SND_SHF_REQ.
Order of processing at receiving end B = {RSP_OK, REQ}

07. WIN-REQ



Result: Yes we can combine. Order of processing at receiving end B = {WIN, REQ}

08. WIN-RSP_NOK

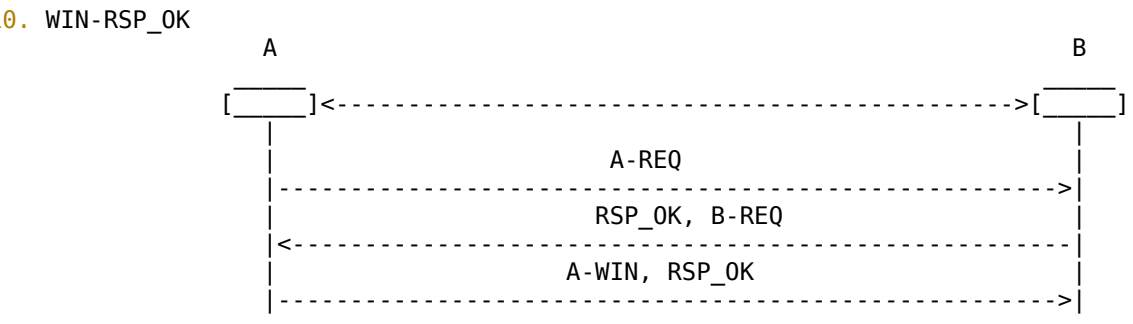


Result: This combination has its dependency on **case** 07. If they can be combined then so is this. Order of processing at receiving end B = {WIN, RSP_NOK}

09. RSP_NOK-RSP-OK

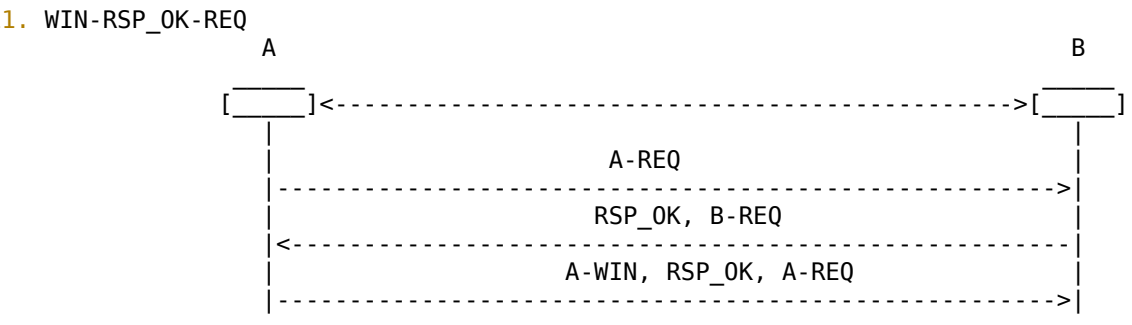
Result: Cannot combine since its hypothetical **case**. These response types are mutually exclusive.

10. WIN-RSP_OK



Result: This combination has its dependency on **case** 07. If they can be combined then so is this. Order of processing at receiving end B = {WIN, RSP_OK}

11. WIN-RSP_OK-REQ

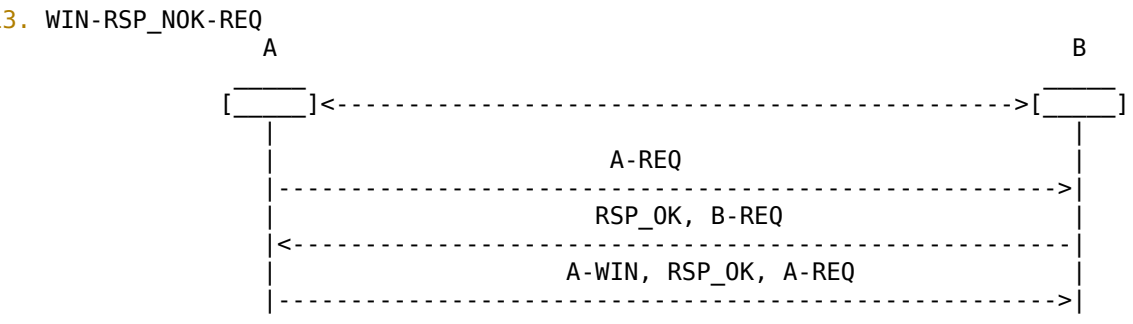


Result: Yes we can be combine. Order of processing at receiving end B = {WIN, RSP_OK, REQ}

12. RSP_OK-RSP_NOK-REQ

Result: Cannot combine since its hypothetical **case**. These response types are mutually exclusive.

13. WIN-RSP_NOK-REQ



Result: It cannot be combined since REQ and RSP_NOK cannot be differentiated with aPkt.WSOpt different than in **case** 07 (i.e. RSP_OK-REQ) in which conflicting condition of request EXP=1 and response EXP=0 can be relaxed since aPkt.WSOpt can be used to distinguish.

14. WIN-RSP_OK-RSP_NOK

Result: Cannot combine since its hypothetical **case**. These response types are mutually exclusive.

15. REQ, RSP.OK, RSP.NOK, WIN

Result: Cannot combine since its hypothetical **case**. These response types are mutually exclusive.

Summary: Out of 15 total cases explored above (including 4 core type) only 9 are permissible

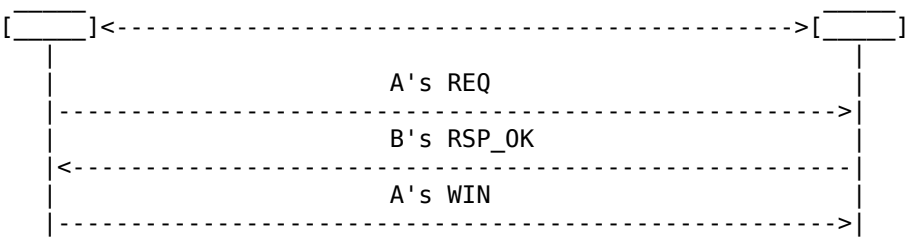
combinations and hence valid types. Following are 9 types of DSPktType:

S.No.	Types	Use-Case
01.	REQ	done
02.	RSP_NOK	done
03.	RSP_OK	done
04.	WIN	done
06.	RSP_OK-REQ	done
07.	WIN-REQ	done
08.	WIN-RSP_NOK	assumed done
10.	WIN-RSP_OK	done
11.	WIN-RSP_OK-REQ	done

Describing type of packets:

1. REQ : packet is used to carry dynamic shift count change request to other end TCP stack
2. RSP_NOK : packet carries NOK acknowledgment from other end which declined the request because of system constraints (like its current CPU load, etc.)
3. RSP_OK : packet carries OK acknowledgment indicating the acceptance of other end TCP stack of the requested shift count
4. WIN : packet carries window update calculated by requester using new shift count approved by other end TCP stack. This is the final message of a dynamic shift count exchange process.
5. RSP_OK-REQ : packet is same as RSP_NOK but caries REQ from other end combined or piggybacked along with it.
6. WIN-REQ : packet is same as WIN but caries REQ from requester end combined or piggybacked along with it
7. WIN-RSP_NOK : packet is same as WIN but caries NOK acknowledgement (i.e. combined with RSP_NOK) to the request of other end TCP stack
8. WIN-RSP_OK : packet is same as WIN but caries OK acknowledgement (i.e. combined with RSP_OK) to the request of other end TCP stack
9. WIN-RSP_OK-REQ: packet is same as WIN but caries OK acknowledgement (i.e. combined with RSP_OK) to the request of other end TCP stack and also a new dynamic shift count change request.

For understanding purpose only the following diagram depicts the flow **if** packets are not combined:




```
/*
 * It carries the value of DSPktType and used
 * when sending to construct particular type of packet
 * Its reset to DSPktType::NODSTYPE after successfully
 * sending
 */
uint32_t combine = DSPktType::NODSTYPE;
/*
 * Shift count reflecting system 'rBufMem'
 * available for receiving packets. Its set by SYS and
 * is reset to NULL after every successful REQ sent
 */
uint32_t SHF = NULL;
/*
 * macros used to form a 'combine' flag
 */
#define REQ     (1 << 0)  /* 000001 */
#define RSP_NOK (1 << 1)  /* 000010 */
#define RSP_OK  (1 << 2)  /* 000100 */
#define WIN     (1 << 3)  /* 001000 */

/*
 * Dynamic Shift Count packet type
 */
typedef enum
    {
    /* single packet type */
    REQ             = 1,
    RSP_NOK         = 2,
    RSP_OK          = 4,
    WIN             = 8,
    /* combined packet type */
    RSP_OK-REQ      = 5,
    WIN-REQ         = 9,
    WIN-RSP_NOK     = 10,
    WIN-RSP_OK,     = 12
    WIN-RSP_OK-REQ, = 13
    /* not recognized */
    NODSTYPE        = 0,
    } DSPktType;
/*
 * Here we find what type a packet belong to by checking the contents
 * of the packet as well as TCP state at the time of receiving.
 * If packet doesn't match the acceptable state then it MUST be dropped
 *
 * @param   aPkt represents received packet
 *
 * @return  DSPktType type enumerations
 */
enum DSPktType isPktType(struct TCPPacket& aPkt)
    {
    if(aPkt.EXP==1 && aPkt.WSopt && !SND_SHF_REQ)
        {
        // this cond. is for REQ sanity check
        //
        if(!RCV_SHF_REQ && !RSP_SEQ && !REQ_SEQ)
            {
            return DSPktType::REQ;
            }
        }
    else if(aPkt.EXP==0 && RCV_SHF_REQ && aPkt.WSopt && (RCV_SHF_REQ > aPkt.WSopt))
        {
        // this cond. is for RSP sanity check
        //
        if(REQ_SEQ && !SND_SHF_REQ && !RSP_SEQ)
            {
            return DSPktType::RSP_NOK;
            }
        }
    else if(RCV_SHF_REQ && aPkt.WSopt && (RCV_SHF_REQ <= aPkt.WSopt))
        {
        if(aPkt.EXP==0)
            {
            // this cond. is for RSP sanity check
            //
            if (REQ_SEQ && !SND_SHF_REQ && !RSP_SEQ)
                {
                return DSPktType::RSP_OK;
                }
            }
        else if (aPkt.EXP==1 && !SND_SHF_REQ)
            {
            // this cond. is for comb:RSP-REQ sanity check
            //
            if(REQ_SEQ && !RSP_SEQ)
                {
                return DSPktType::RSP_OK-REQ;
                }
            }
        }
    else if(SND_SHF_REQ && RSP_SEQ && ((RSP_SEQ + aPkt.SEG.LEN) <= aPkt.ACK))
        {
        // this cond. is for WIN sanity check
        //
        if (!RCV_SHF_REQ && !REQ_SEQ && !aPkt.WSopt && !aPkt.EXP)
            {
            return DSPktType::WIN;
            }
        else if(aPkt.EXP==1 && aPkt.WSopt)
            {
            if (RCV_SHF_REQ && (RCV_SHF_REQ <= aPkt.WSopt))
                {
                // this cond. is for WIN-RSP_OK-REQ sanity check
                //
                if (REQ_SEQ)
                    {
                    return DSPktType::WIN-RSP_OK-REQ;
                    }
                }
            else if (!RCV_SHF_REQ)
                {
```

```
                // this cond. is for WIN-REQ sanity check
                //
                if (!REQ_SEQ)
                    {
                    return DSPktType::WIN-REQ;
                    }
                }
            }
        else if(aPkt.EXP==0 && RCV_SHF_REQ && aPkt.WSopt)
            {
            if(RCV_SHF_REQ > aPkt.WSopt)
                {
                // this cond. is for WIN-RSP_NOK sanity check
                //
                if(REQ_SEQ)
                    {
                    return DSPktType::WIN-RSP_NOK;
                    }
                }
            else if (RCV_SHF_REQ <= aPkt.WSopt)
                {
                // this cond. is for WIN-RSP_OK sanity check
                //
                if(REQ_SEQ)
                    {
                    return DSPktType::WIN-RSP_OK;
                    }
                }
            }

    /* if control flow reaches here then this
     * means that the packet must be dropped
     */
    return NODSTYPE;
    }

/*
 * @return TRUE if can raise REQ otherwise FALSE
 */
bool canSendREQ()
    {
    if (SND_DS_CAP && RCV_SHF_REQ == NULL && SHF > own::shf.cnt)
        {
        return TRUE;
        }
    else
        {
        SHF = NULL;
        }

    return FALSE;
    }

/*
 * Following methods process DS packet types WIN, RSP_OK/ RSP_NOK, REQ
 *
 * @param  aPkt represents received WIN packet
 */
void ProcessWIN(struct TCPPacket& aPkt, bool deferSend=FALSE)
    {
    oEnd::shf.cnt = SND_SHF_REQ;
    own::SND.WND  = oEnd::rWnd << oEnd::shf.cnt;
    RSP_SEQ = NULL;
    SND_SHF_REQ = NULL;
    SND.WL1 = aPkt.SEQ;
    SND.WL2 = aPkt.ACK;

    if(!deferSend)
        {
        Send(combine);
        }
    }
void ProcessRSP_OK(bool deferSend=FALSE)
    {
    own::shf.cnt = RCV_SHF_REQ;
    own::rWnd = own::rBufMem >> own::shf.cnt;
    own::RCV.WND = own::rWnd << own::shf.cnt;
    tcpPktToSend.window = own::rWnd;
    RCV_SHF_REQ = NULL;
    REQ_SEQ = NULL;
    combine= WIN;

    if(SYS_TRIGGER && canSendREQ())
        {
        combine += REQ;
        }

    if(!deferSend)
        {
        Send(combine);
        }
    }

void ProcessRSP_NOK(bool deferSend=FALSE)
    {
    REQ_SEQ = NULL;
    RCV_SHF_REQ = NULL;

    if(!deferSend)
        {
        Send(combine);
        }
    }

void ProcessRSP_OK-REQ()
    {
    ProcessRSP_OK(TRUE);
    ProcessREQ();
    }

void ProcessWIN-RSP_OK()
    {
    ProcessWIN(TRUE);
    ProcessRSP_OK();
    }

void ProcessWIN-RSP_OK-REQ()
    {
    ProcessWIN(TRUE);
    ProcessRSP_OK(TRUE);
    ProcessREQ();
    }

/*
 * @param aPkt  represents received packet
 */
void ProcessREQ(struct TCPPacket& aPkt, bool deferSend=FALSE)
    {
    if(isShfCntFeasible(rBufMem, aPkt.WSopt))
        {
        SND_SHF_REQ = aPkt.WSopt;
        combine = RSP_OK;
        }
    else
        {
        // setting it here is compulsory ever if
        // its a RSP_NOK to support asynchronous
        // send process
        //
        SND_SHF_REQ = aPkt.WSopt;
        tcpPktToSend.WSopt = oEnd.shf.cnt
        combine = RSP_NOK;
        }

    if((combine == DSPktType::RSP_OK) && SYS_TRIGGER && canSendREQ())
        {
        combine += REQ;
        }

    if(!deferSend)
        {
        Send(combine);
        }
    }

/*
 * Checks the feasibility whether RSP and REQ can be combined
```

```
 * This overrides the understanding that when EXP=1 it
 * represents request and is mutually exclusive of EXP=0
 * which represent response.
 *
 * @param combine Its a flag which carries DSPktType
 */
bool CanCombineRSP-REQ(combine)
    {
    // if we have to combine with request then :
    // the rule is that we can generate only request for increasing shift count at present.
    //
    bool canCombine = FALSE;
    switch(combine):
        {
        case DSPktType::RSP_OK-REQ:
            {
            if(RCV_SHF_REQ >= SND_SHF_REQ)
                {
                canCombine = TRUE;
                }
            break;
            }
        default:
            {
            //
            // do nothing aon
            //
            }
        }

    return canCombine;
    }

/*
 * Form a packet for sending to other end TCP stack
 * based on the Dynamic Shift count packet type.
 *
 * @param combine Its a flag which carries DSPktType
 */
void Send(uint32_t combine)
    {
    // create and send combine type of packet
    // reset combine
    switch(combine)
        {
        case DSPktType::REQ:
            {
            if (canSendREQ())
                {
                // aShf : system discovered shf.cnt which can be supported at current.
                RCV_SHF_REQ = SHF;

                // Send code
                // todo

                // after successful sending reset SHF to NULL
                SHF = NULL;
                }
            break;
            }
        case DSPktType::RSP_NOK:
            {
            SND_SHF_REQ = NULL;
            // todo
            break;
            }
        case DSPktType::RSP_OK:
            {
            // todo
            break;
            }
        case DSPktType::WIN:
            {
            // todo
            break;
            }
        case DSPktType::RSP_OK-REQ:
            {
            RCV_SHF_REQ = SHF;
            if(CanCombineRSP-REQ(combine))
                {
                EXP = 1;
                }
            else
                {
                RCV_SHF_REQ = NULL;
                // reset combine back to RSP_OK
                combine  = RSP_OK;
                }

            // Send code
            // todo

            // after successful sending reset SHF to NULL
            SHF = NULL;
            break;
            }
        case DSPktType::WIN-REQ:
            {
            RCV_SHF_REQ=SHF;

            // Send code
            // todo

            // after successful sending reset SHF to NULL
            SHF = NULL;
            break;
            }
        case DSPktType::WIN-RSP_NOK:
            {
            break;
            }
        case DSPktType::WIN-RSP_OK:
            {
            // todo
            break;
            }
        case DSPktType::WIN-RSP_OK-REQ:
            {
            // todo
            break;
            }
        case DSPktType::NODSTYPE:
        default:
            {
            //
            // do nothing aon
            //
            }
        } // end of Switch block

combine = NULL;
}


            // ============================================================================//
            //                                                                             //
            //                            End of document                                  //
            //                                                                             //
            // ============================================================================//
```