Copy Number .....

FM3TR Decomposition

P6957-11-005
13-April-2000 Issue 1

Written by

.....

E.D.Willink

Reviewed and Approved by

Project Manager

.....

C.Waugh

Authorised by

Project Director

Date ....

C.P.Ash

DERA Portsdown
Portsdown Hill Road
Fareham
Hants
PO17 6AD

Configuration Management	
CM Tool/Version	Visual SourceSafe v5.0 and v6.0
Library Location	Q:\P6957\Vss\…
Library Project	$/P6957/Project Library/11 Technical Reports & Notes/ 005 FM3TR Decomposition
Baseline	
Elements	`FM3TRDecomposition.doc` - words `Fm3trTop.vsd` - top level pictures `Fm3trDlc.vsd` - DLC layer pictures `Fm3trMac.vsd` - MAC layer pictures `Fm3trPhl.vsd` - PHL layer pictures `Fm3tr.wdl` - syntax checked examples `*.wsg` - picture icon templates

Modification History		
Author	**Date**	**Description**
EDW	25 February 2000	Preliminary draft made available to interested parties
EDW	13-April 2000	End of phase 1 : Rx and Nwk still missing

Table of Contents

1 Introduction

A decomposition of version 2.0 of the FM3TR Waveform using the Waveform Description Language is presented in this document. At present, the focus is on demonstrating how WDL can be used to provide improved specifications, rather than on providing that improved specification. There are therefore some significant sections of editorial content that could be beneficially copied from the existing specification in order to make this specification easier to appreciate.

1.1 Further Work

Text with a 10% grey background indicates areas where there is more work to be done.

Text with a 25% grey background indicates extensions to, or resolutions of problems in, the FM3TR specification.

The FM3TR specification is devoid of all conventional specifications such as signal to noise ratios, receive sensitivity, transmit spectrum, or error rates. These specifications are therefore missing from this specification as well. They may be provided for Fm3tr, at the root of the specification, or delegated almost entirely to the Fm3tr.Phl.Radio entity. Some specifications such as signal to noise could exploit yet to be defined built-in properties. Others may need to specify the behaviour of a test harness.

The following entities have not been decomposed because there is no specification for them

```
Fm3tr.Phl.Radio
Fm3tr.Phl.RxFsm ...
```

It has not been possible to perform the decomposition of the following entities due to problems with the FM3TR specification.

```
Fm3tr.Nwk ...
Fm3tr.Dlc.Nwk
```

The following ‘library’ entities have yet to be properly defined

```
F(t), F1(t)
MultiGet
Nco
PriorityScheduler, RoundRobinScheduler
Uniform
```

Some parts of the specification such as the AX25 protocol could be converted to reusable library entities.

This document is not complete. It has not yet been thoroughly cross-checked against the FM3TR specification. Many of the WDL constructs have been syntax checked, but since there is no semantic checker yet, there is ample scope for spelling mistakes and meaningless statements.

This document will need revisiting once a version of the FM3TR specification has been produced that resolves the many ambiguities and contradictions.

1.2 Further Documents

This is the first specification to be expressed in WDL, and so there is a temptation to explain why it is presented in the way that it is. This is not the role of a specification. The specification is therefore largely restricted to specification issues. A separate tutorial document explaining why certain specification approaches were adopted would no doubt

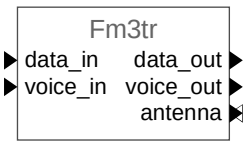
be valuable.

1.3 Notes

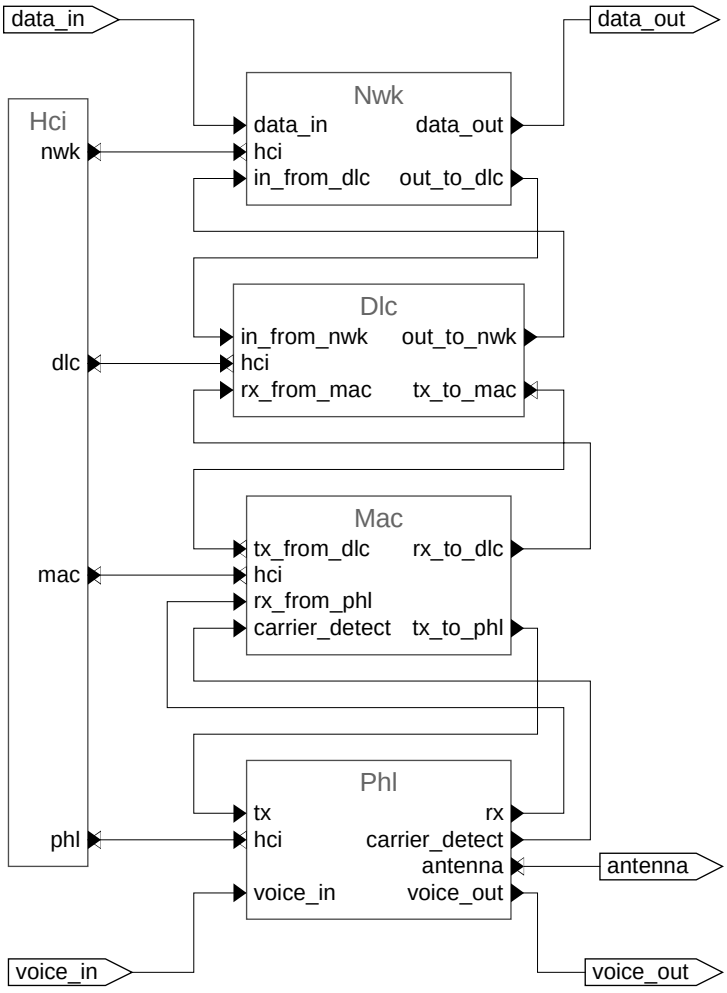
The port descriptions of each entity identify the flow types and data types to ease understanding. These have all been entered manually. With proper tool support, most of these could be left unspecified, allowing a deduction system to make appropriate choices automatically, in particular avoiding the need for the sometimes almost arbitrary choice of event or token flow.

Annotations on diagrams such as `[N]` are intended to aid understanding by showing the array dimensionality of the adjacent flow. These annotations are not part of the specification. The `[]` annotation indicates a scalar flow.

2 Fm3tr



The FM3TR radio specifies a realisation of the lower three layers of an OSI protocol stack using four layers: Network, Data Link Control, Medium Access Control and Physical.



The FM3TR specification identifies a variety of configuration and operation parameters, but does not specify their interface. An Hci entity is therefore introduced to terminate these unspecified interactions.

2.1 Constants

```
constant ARQ_CHANNELS = 10 : natural;
constant MAX_K = 7 : natural;
constant MAX_INFO = 2048 : natural; // Longest valid data body.
constant DEFAULT_T1 : T1Time;
constant DEFAULT_T2 : T2Time;
constant DEFAULT_T3 : T3Time;
constant DEFAULT_N2 : N2Type;
```

Omitted values for constants mandate a specification by each implementation.

2.2 Types

Types that are used in message flows between layers are declared at the root of the decomposition hierarchy where all child entities can see them.

2.2.1 Utility Types

```
type Octet : natural { bits 8; };
type Word : natural { bits 16; };

record ValidBit
{
   attribute bit : boolean;
   attribute valid : boolean;
};
```

2.2.2 Configuration Types

```
type Address : natural { minimum 1; maximum 9999; };
type ChannelNumber : natural { minimum 0; maximum ARQ_CHANNELS; };
enumeration ChannelStatus { S1, S2, S3, S4, S5, S6, S7 };
type CrcErrorCount : natural { minimum 0; maximum 10000; };
type PacketCount : natural { minimum 0; maximum 10000; };

type CdTime : Time;
type PersistenceTime : Time;
type SlotTime : Time;
type T1Time : Time;
type T2Time : Time;
type T3Time : Time;

type N2Type : natural { maximum 255; };
type KType : natural { maximum MAX_K; overflow wrap; };
```

Ktype is defined for the limited number range 0 to 7 inclusive, with numeric wraparound for out of range arithmetic. Addition and subtraction of KType values therefore specifies wraparound conversion automatically without any further specification complexity.

```
type IsRobust : boolean;
enumeration TwNumber { TW1A, TW1B };
```

2.2.3 Interface Types

```
type VoiceIn : ValidBit;
type VoiceOut : ValidBit;

record DataIn;
record DataOut;

type Antenna;

type CarrierDetect : boolean;
```

```
type Info : Octet[*];
type NwkPacket : Octet[*];
type DlcPacket : Octet[*];
type MacPacket : Octet[*];
```

2.2.4 Hci to Dlc Types

```
record ChannelAddress
{
    attribute number : ChannelNumber;
    attribute address : Address;
};

enumeration HciToDlcEnums
{
    SET_LOCAL_ADDRESS, GET_LOCAL_ADDRESS,
    SET_K, GET_K,
    SET_N2, GET_N2,
    SET_T1, GET_T1,
    SET_T2, GET_T2,
    SET_T3, GET_T3,
    GET_STATUS
};

union HciToDlcMessages : HciToDlcEnums
{
    case SET_LOCAL_ADDRESS :
        Address;
    case GET_LOCAL_ADDRESS :
        void : Address;
    case SET_K :
        KType;
    case GET_K :
        void : KType;
    case SET_N2 :
        N2Type;
    case GET_N2 :
        void : N2Type;
    case SET_T1 :
        T1Time;
    case GET_T1 :
        void : T1Time;
    case SET_T2 :
        T2Time;
    case GET_T2 :
        void : T2Time;
    case SET_T3 :
        T3Time;
    case GET_T3 :
        void : T3Time;
    case GET_RX_PACKETS, GET_TX_PACKETS:
        void : PacketCount;
    case GET_STATUS :
        ChannelStatus : Status;
};
```

2.2.5 Hci to Mac Types

```
enumeration HciToMacEnums
{
    SET_T_SLOT, GET_T_SLOT,
    SET_T_PERSISTENCE, GET_T_PERSISTENCE,
    GET_RX_PACKETS,
    GET_TX_PACKETS
};
```

```
union HciToMacMessages : HciToMacEnums
{
    case SET_T_SLOT :
        SlotTime;
    case GET_T_SLOT :
        void : SlotTime;
    case SET_T_PERSISTENCE :
        PersistenceTime;
    case GET_T_PERSISTENCE :
        void : PersistenceTime;
    case GET_RX_PACKETS, GET_TX_PACKETS :
        void : PacketCount;
};
```

2.2.6 Hci to Phl Types

```
enumeration HciToPhlEnums
{
    SET_T_CD_ON, GET_T_CD_ON,
    SET_T_CD_OFF, GET_T_CD_OFF,
    GET_RX_PACKETS,
    GET_TX_PACKETS,
    GET_CRC_ERRORS,
    SET_ROBUST, GET_ROBUST
    SET_TW, GET_TW
};

union HciToPhlMessages : HciToPhlEnums
{
    case SET_T_CD_ON, SET_T_CD_OFF :
        CdTime;
    case GET_T_CD_ON, GET_T_CD_OFF :
        void : CdTime;
    case GET_RX_PACKETS, GET_TX_PACKETS :
        void : PacketCount;
    case GET_CRC_ERRORS :
        void : CrcErrorCount;
    case SET_ROBUST :
        IsRobust;
    case GET_ROBUST :
        void : IsRobust;
    case SET_TW :
        TwNumber;
    case GET_TW :
        void : TwNumber;
};
```

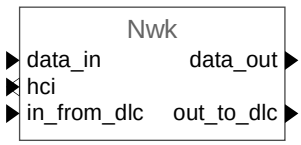
2.3 Ports

```
in:token voice_in : VoiceIn;
out:token voice_out : VoiceOut;
```

The clock synchronisation issue at the voice interface is left unresolved by specification of token flows. The voice source must therefore provide a continuous flow to avoid a stall that will result in a missed hop. The voice sink need not respond at all, which may require the implementation to support an infinite capacity output FIFO.

```
in:event data_in : DataIn;
out:event data_out : DataOut;
inout:signal antenna : Antenna;
```

3 NWK Layer

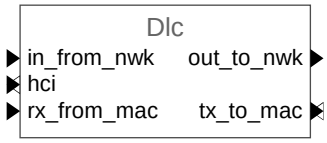


The FM3TR specification requires an implementation to use a COTS PC to support the TCP/IP protocol stack, and specifies the use of a COTS PPP protocol stack to communicate between the PC and the main FM3TR radio. There are a significant number of problems with the way in which this requirement is imposed, and it is not appropriate to perform decomposition of the Nwk layer until these structural issues have been resolved.

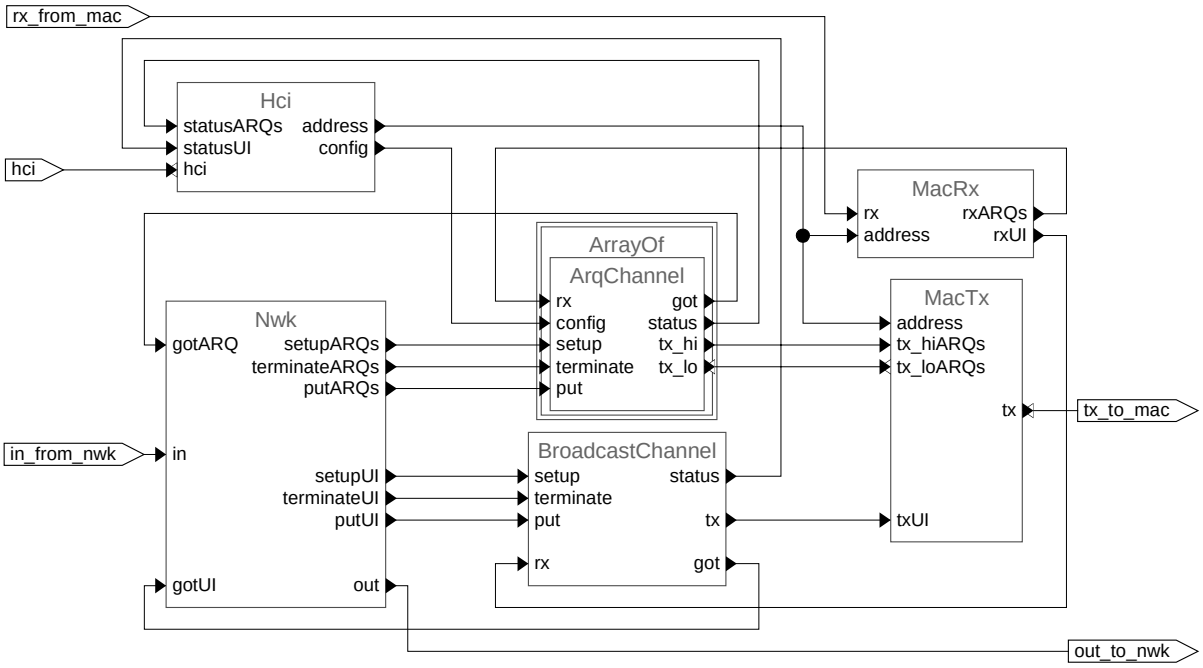
Ports

```
in:event data_in : DataIn;
in:token in_from_dlc : NwkPacket;
in:event hci : HciToNwkMessages;
out:event data_out;
out:event out_to_dlc : NwkPacket;
```

4 DLC layer



A discussion of the imperfections in the DLC specification appears as part of P6957-11-009. The description here is of a plausible bespoke specification. There may be some opportunity for trying to generalise it into a tailored X25 protocol module.



The DLC layer comprises a broadcast channel and (an array of) 10 ARQ channels. An interface to the HCI supports control and interrogation of the layer and of the channels. The interface to the NWK layer multiplexes the channels to form a single link. The MacTx interface to the MAC layer arbitrates the demands of the 11 channels for the transmit resource and MacRx dispatches received packets to the appropriate channels.

Ports

```
in:event hci : HciToDlcMessages;
in:event rx_from_mac : DlcPacket;
in:event in_from_nwk : NwkPacket;
out:event tx_to_mac : DlcPacket;
out:event out_to_nwk : NwkPacket;
```

Entities

```
entity channels : ArqChannel[ARQ_CHANNELS];
```

ArrayOf is a purely graphical annotation. It is not needed as part of WDL where arrays are declared as easily as scalars.

4.1 Fm3tr.Dlc Types

A substantial number of types are needed to define the bit-truth of DLC messages.

4.1.1 Configuaration

The `DlcConfig` bundle type supports distribution of layer configuration parameters as a single bundle rather than multiple connections.

```
bundle DlcConfig
{
    value t1 : T1Time;
    value t2 : T2Time;
    value t3 : T3Time;
    value n2 : N2Type;
    value k : KType;
};
```

4.1.2 Data

The `GotPacket` and `PutPacket` types describe data passed from and to individual channels.

```
type GotPacket : Octet[*];
type PutPacket : Octet[*];
```

4.1.3 Transmit information types

The transmit information types describe the information in partially constructed transmit messages.

```
record TxI
{
    attribute info : Info;
    attribute n_s : KType;
    attribute pf : Pf;
};

enumeration TxSEnums { RRc, RRr, REJc, REJr };
union TxS : TxSEnums
{
    case RRc, RRr, REJc, REJr : Pf;
};

enumeration TxUEnums { DISC, DM, FRMR, SABM, UA };
union TxU : TxUEnums
{
    case DISC, DM, FRMR, SABM, UA : Pf;
};

record TxUI
{
    attribute info : Info;
};

enumeration TxISEnums { I, S };
union TxIS : TxISEnums
{
    case I : TxI;
    case S : TxS;
};

record TxN : TxIS
```

```
{
    attribute n_r : KType;
};

enumeration TxMEnums { IS, U };
union TxM : TxMEnums
{
    case IS : TxN;
    case U : TxU;
    case UI : TxUI;
};

record TxMessage : TxM
{
    attribute source_address : Address;
};

record TxPacket : TxMessage
{
    attribute destination_address : Address;
};
```

4.1.4 Bit-true fields

It is convenient to define each message field individually, so that partial sets of fields can then be overlayed as required.

```
overlay DlcAddressType : DlcPacket
{
    attribute address_type = 0 : boolean[8] << 0;
};
overlay DlcDestinationAddress : DlcPacket
{
    attribute address : DlcAddress << 8;
};
overlay DlcSourceAddress : DlcPacket
{
    attribute address : DlcAddress << 24;
};
overlay DlcIsI : DlcPacket
{
    attribute is_i = 0 : boolean[1] << 40;
};
overlay DlcIsS : DlcPacket
{
    attribute is_s = 1 : boolean[2] << 40;
};
overlay DlcIsU : DlcPacket
{
    attribute is_u = 3 : boolean[2] << 40;
};
overlay DlcLo : DlcPacket
{
    attribute lo : boolean[2] << 42;
};
overlay DlcHi : DlcPacket
{
    attribute hi : boolean[3] << 45;
};
overlay DlcNs : DlcPacket
{
    attribute n_s : boolean[3] << 41;
};
overlay DlcPf : DlcPacket
```

```
{
    attribute pf : boolean[1] << 44;
};
overlay DlcNr : DlcPacket
{
    attribute n_r : boolean[3] << 45;
};
overlay DlcPid : DlcPacket
{
    attribute pid = 1 : boolean[7] << 48;
};
overlay DlcCmd : DlcPacket
{
    attribute cmd : boolean[1] << 55;
};
overlay DlcData : DlcPacket
{
  attribute data : octet[*] << 56;
};
overlay DlcNoData : DlcPacket
{
  constraint: DlcNoData._shape[0] = 7;
};
```

The command/response bit has restrictive usage, so its two values are identified by distinct types.

```
overlay DlcIsResponse : DlcCmd
{
    constraint: cmd = 0;
};
overlay DlcIsCommand : DlcCmd
{
    constraint: cmd = 1;
};
overlay DlcIsUI : DlcIsU
{
    attribute lo = 0 : DlcLo;
    attribute pf = 0 : DlcPf;
    attribute hi = 0 : DlcHi;
};

overlay DlcRR : DlcIsS
{
    attribute lo = 0 : DlcLo;
};
overlay DlcREJ : DlcIsS
{
    attribute lo = 1 : DlcLo;
};
overlay DlcRRc : DlcRR, DlcIsCommand;
overlay DlcRRr : DlcRR, DlcIsResponse;
overlay DlcREJc : DlcREJ, DlcIsCommand;
overlay DlcREJr : DlcREJ, DlcIsResponse;

overlay DlcSABM : DlcIsU, DlcIsResponse
{
    attribute lo = 3 : DlcLo;
    attribute hi = 1 : DlcHi;
};
overlay DlcDISC : DlcIsU, DlcIsCommand
{
    attribute lo = 0 : DlcLo;
    attribute hi = 2 : DlcHi;
};
```

```
overlay DlcDM : DlcIsU, DlcIsResponse
{
    attribute lo = 3 : DlcLo;
    attribute hi = 0 : DlcHi;
};
overlay DlcUA : DlcIsU, DlcIsResponse
{
    attribute lo = 0 : DlcLo;
    attribute hi = 3 : DlcHi;
};
overlay DlcFRMR : DlcIsU, DlcIsResponse
{
    attribute lo = 1 : DlcLo;
    attribute hi = 4 : DlcHi;
};
```

4.1.5 Outer Bit-true Receive Types

The first pass receive processing in `MacRx` need only validate some of the fields.

```
overlay RxFrame : DlcPacket, DlcAddressType,
          DlcDestinationAddress, DlcSourceAddress, DlcPid;
overlay RxUI : RxFrame, DlcIsUI, DlcData;
overlay RxARQ : RxFrame, DlcPf;
```

4.1.6 Inner Bit-true Receive Types

Complete definitions of fields are required for the per-channel receive processing.

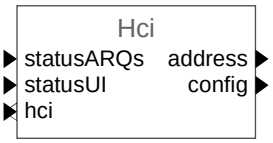
```
overlay RxIS : RxARQ, DlcNr;

overlay RxI : RxIS, DlcIsI, DlcNs, DlcData;

overlay RxS : RxIS, DlcIsS;
overlay RxRRc : RxS, DlcRRc, DlcNoData;
overlay RxRRr : RxS, DlcRRr, DlcNoData;
overlay RxREJc : RxS, DlcREJc, DlcNoData;
overlay RxREJr : RxS, DlcREJr, DlcNoData;

overlay RxU : RxARQ, DlcIsU {};
overlay RxSABM : RxU, DlcSABM, DlcNoData;
overlay RxDISC : RxU, DlcDISC, DlcNoData;
overlay RxDM : RxU, DlcDM, DlcNoData;
overlay RxUA : RxU, DlcUA, DlcNoData;
overlay RxFRMR : RxU, DlcFRMR, DlcNoData;
```

4.1.7 Bit-true Transmit Types

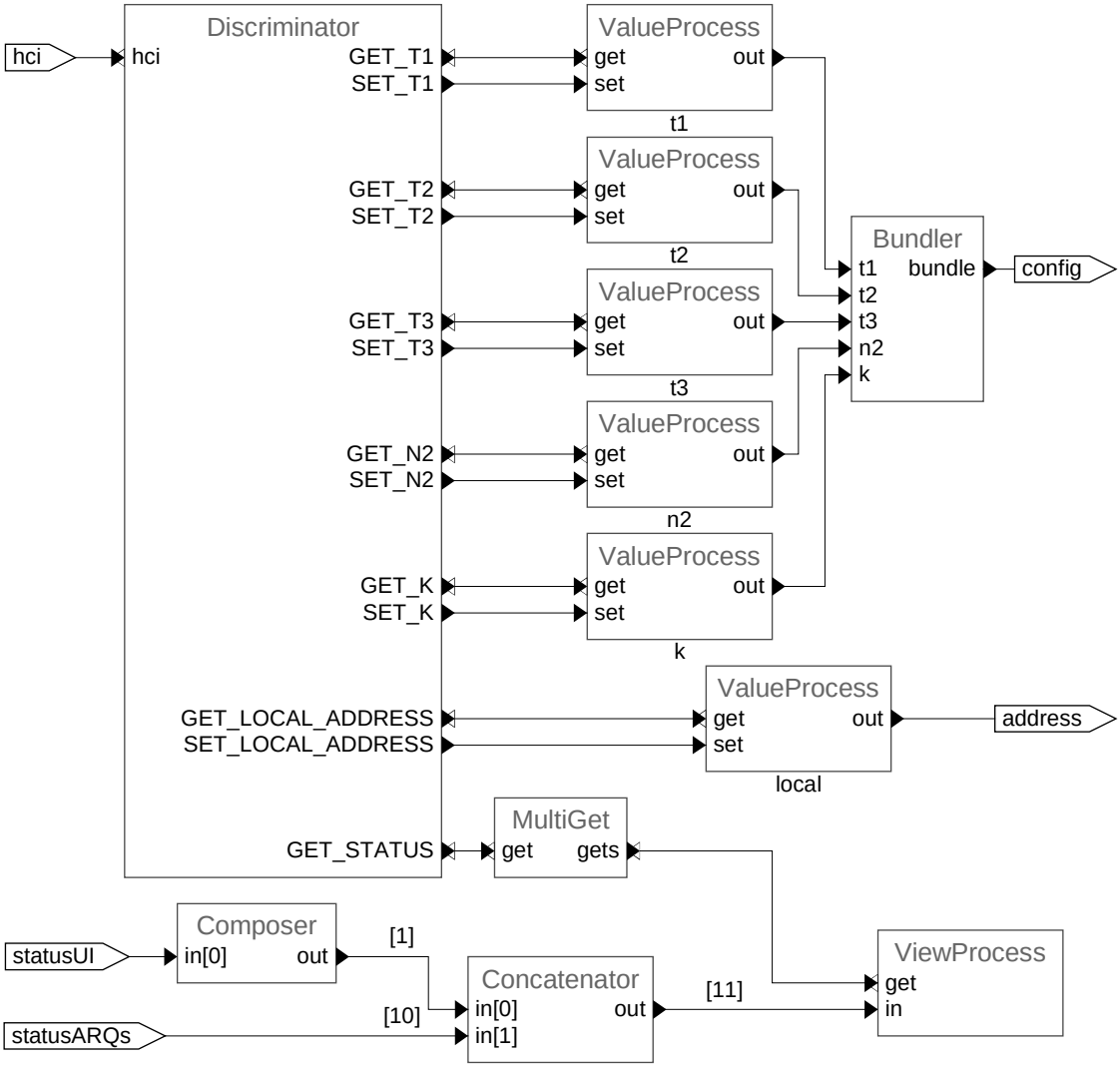
A single mapping from the TxPacket abstract information type to a bit-true type is needed for use within the MacTx entity.

```
layout DlcMessage = TxPacket : DlcPacket
{
    attribute address_type : DlcAddressType;
    attribute destination_address
        = TxPacket.destination_address : DlcDestinationAddress;
    attribute source_address
        = TxPacket.source_address : DlcSourceAddress;
    attribute pid : DlcPid;
    discriminate(TxPacket.TxM)
    {
        case IS:
        {
            attribute n_r = TxPacket.n_r : DlcNr;
            attribute pf = TxPacket.pf : DlcPf;
            discriminate(TxPacket.TxIS)
            {
                case I:
                {
                    attribute is_i : DlcIsI;
                    attribute n_s = TxPacket.n_s : DlcNs;
                    attribute info = TxPacket.info : DlcData;
                };
                case S:
                {
                    discriminate (TxPacket.TxS)
                    {
                        case RRc: { attribute is_s : DlcRRc; };
                        case RRr: { attribute is_s : DlcRRr; };
                        case REJc: { attribute is_s : DlcREJc; };
                        case REJr: { attribute is_s : DlcREJr; };
                    };
                };
            };
        };
        case U:
        {
            discriminate (TxPacket.TxU)
            {
                case SABM: { attribute is_u : DlcSABM; };
                case DISC: { attribute is_u : DlcDISC; };
                case DM: { attribute is_u : DlcDM; };
                case UA: { attribute is_u : DlcUA; };
                case FRMR: { attribute is_u : DlcFRMR; };
            };
        };
        case UI:
        {
            attribute is_ui : DlcIsUI;
            attribute info = TxPacket.info : DlcData;
        };
    };
};
```

4.2 Fm3tr.Dlc.Hci



The HCI interface for the DLC layer supports maintenance and interrogation of the configuration parameters: T1, T2, T3, N2, K, the local address and interrogation of the per-channel status.



The discriminator splits the union of possible HCI messages into those associated with each maintained context.

Ports

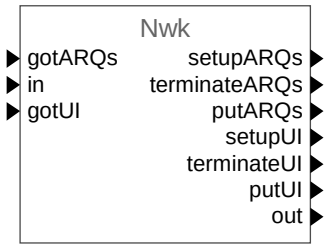
```
in:event hci : HciToDlcMessages;
in:value statusUI : ChannelStatus;
in:value statusARQs : ChannelStatus[ARQ_CHANNELS];
out:value config : DlcConfig;
out:value address : Address;
```

Configuration

```
constraint: t1.initial_value = DEFAULT_T1;
constraint: t2.initial_value = DEFAULT_T2;
constraint: t3.initial_value = DEFAULT_T3;
constraint: n2.initial_value = DEFAULT_N2;
constraint: k.initial_value = MAX_K;
constraint: local.initial_value = 0;
```

Latencies should be specified to avoid the need for unnecessarily tight coupling of the HCI interface.

4.3 Fm3tr.Dlc.Nwk

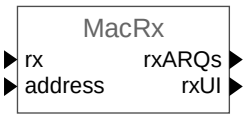


As noted under Fm3Tr.Nwk in Section 3, the current state of the FM3TR specification does not adequately define the communication between NWK and DLC layers, or how the 10 ARQ channels can be exploited through the PPP interface. There appears to be a missing router specification.

Ports

```
in:token gotARQs : GotPacket[ARQ_CHANNELS];
in:token gotUI : GotPacket;
in:event in : NwkPacket;
out:event setupARQs : Address[ARQ_CHANNELS];
out:event terminateARQs : void[ARQ_CHANNELS];
out:event putARQs : PutPacket[ARQ_CHANNELS];
out:event setupUI : void;
out:event terminateUI : void;
out:event putUI : PutPacket;
out:event out : NwkPacket;
```

4.4 Fm3tr.Dlc.MacRx



The receive interface from the MAC layer specifies preliminary protocol validation and the routing of messages to the Broadcast or (all) ARQ channels.

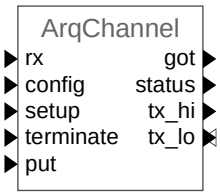
Ports

```
in:event rx : DlcPacket;
in:value address : Address;
out:event rxARQs : RxARQ[ARQ_CHANNELS];
out:event rxUI : RxUI;
```

Specification

```
response rx
{
    specification
    {
        layin (rx)
        {
            case RxUI: { rxUI(rx); };
            case RxARQ: { rxARQs(rx); };
            default: {};
        };
    };
};
```

4.5 Fm3Tr.Dlc.ArqChannel



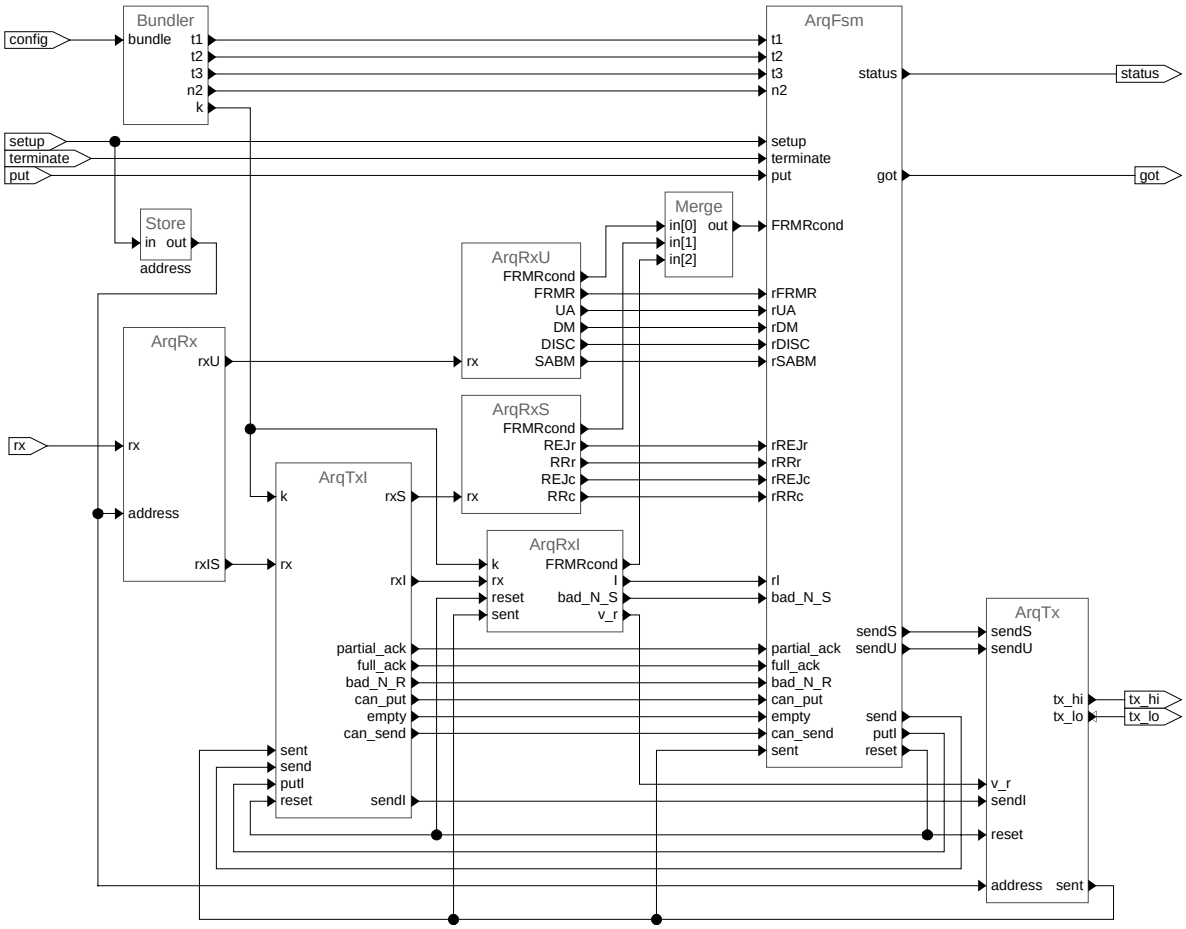
The behaviour of each ARQ channel is largely determined by its Finite State Machine (ArqFsm). Channel status is controlled by the `setup` and `terminate` events and made available via `status`.

Response to a received packet provided as `rx` is assisted by three levels of refinement for each received message from the MAC layer. ArqRx specifies initial validation, and passes Unnumbered messages direct to ArqRxU for further interpretation. Other messages are passed through ArqTxI which exploits any message acknowledgement before passing Supervisory messages on for further interpretation by ArqRxS or Information messages to ArqRxI. Information packets are propagated by `got`.

Transmit processing in response to `put` is managed by the state machine with ArqTxI maintaining the buffer and context of information messages awaiting transmission or acknowledgement.

ArqTx manages requests for physical transmission. High priority control messages are routed to the `tx_hi` output and lower priority information messages at `tx_lo`.

The remote address for the channel is configured by the channel setup message.



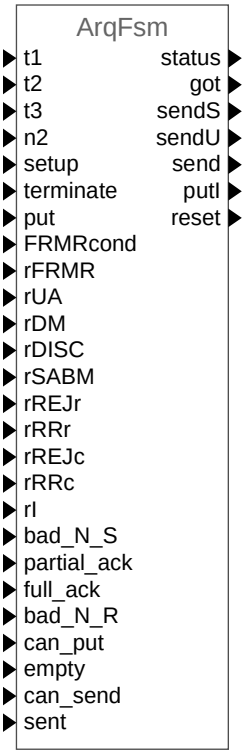
Ports

```
in:event setup : Address;
in:event terminate : void;
in:event put : PutPacket;
in:bundle config : DlcConfig;
in:event rx : RxARQ;
out:value status : ChannelStatus;
out:token got : GotPacket;
out:token tx_hi : TxMessage;
out:token tx_lo : TxMessage;
```

Configuration

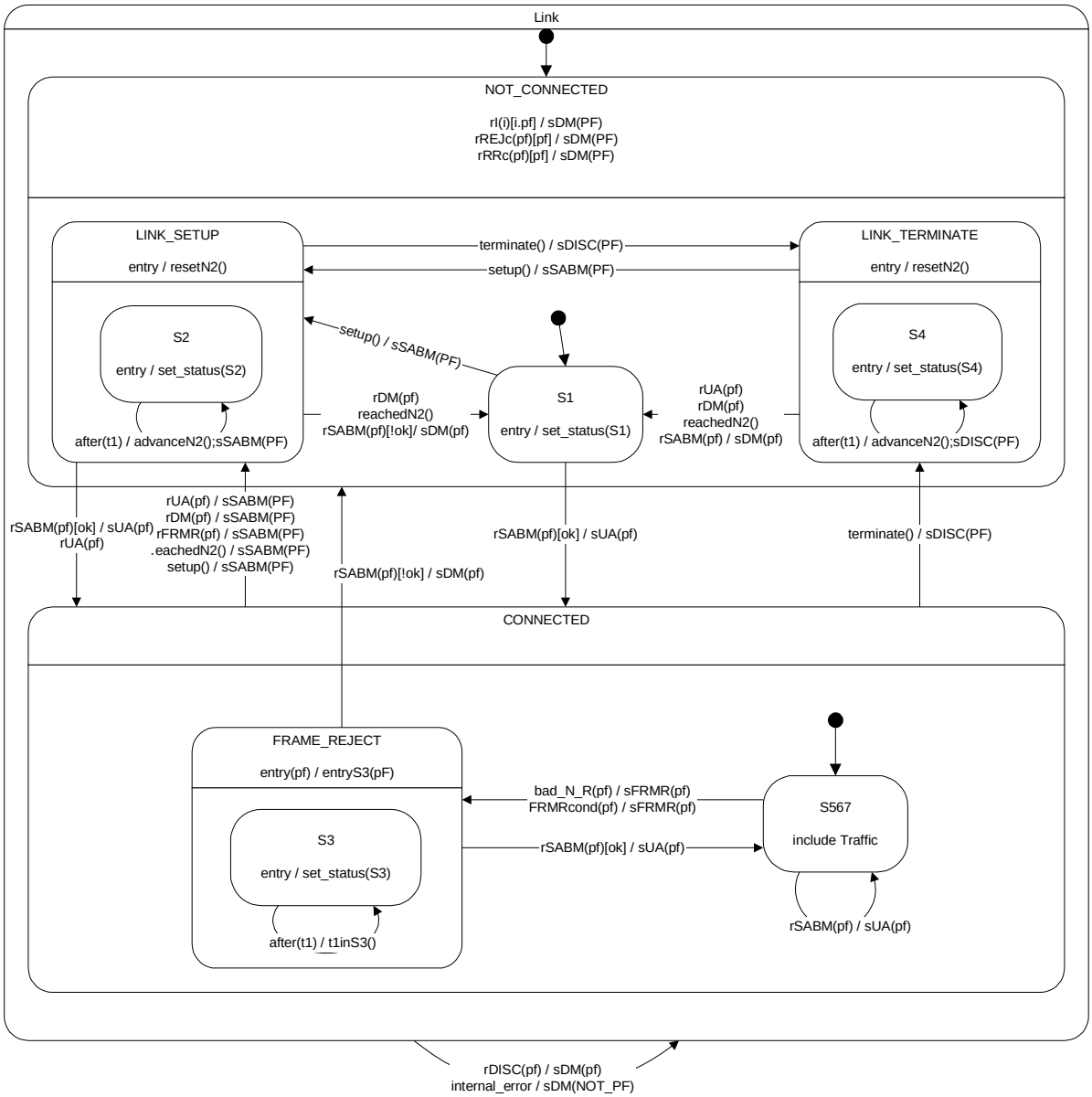
```
constraint: address.initial_value = 0;
```

4.5.1 Fm3Tr.Dlc.ArqFsm



The state machine operates at three levels

- an outer connection level exhibiting the specified S1/S2/S4, S3/5/6/7 behaviour
- intermediate levels
  - a not connected level exhibiting the S1, S2 and S4 behaviour
  - a connected level exhibiting the S3, S5/S6/S7 behaviour
- an inner level exhibiting the S5, S6, S7 behaviour.



`rSABM` transitions are guarded by `ok`, corresponding to the FM3TR phraseology, "if a connection can be accepted". When can a connection not be accepted?

Ports

HCI control and response

```
in:value t1 : T1Time;
in:value t2 : T2Time;
in:value t3 : T3Time;
in:value n2 : N2Type;
out:value status = _status : ChannelStatus;
```

NWK control and response

```
in:event setup : Address;
```

```
in:event terminate : void;
in:event put : Info;
out:token got : Info;
```

MAC receive packets

```
in:event rFRMR : Pf;
in:event rUA : Pf;
in:event rDM : Pf;
in:event rDISC : Pf;
in:event rSABM : Pf;
in:event rREJr : Pf;
in:event rRRr : Pf;
in:event rREJc : Pf;
in:event rRRc : Pf;
in:event rI : RxI;
in:event FRMRcond : Pf;
in:event bad_N_S : Pf;
in:event bad_N_R : void;
```

Transmit control

```
in:value can_put : boolean;
in:value empty : boolean;
in:value can_send : boolean;
in:event partial_ack : void;
in:event full_ack : void;
in:event sent : KType;
out:token sendS : TxS;
out:token sendU : TxU;
out:event send : void;
out:token putI : Info;
out:event reset : void;
```

Expiry of the N2 time-out generates an internal event:

```
internal:event reachedN2 : void;
```

State Variables

```
attribute _status = S1 : ChannelStatus;
```

The actual states are hidden attributes of the various state machines. The state that the FM3TR specification requires to be published is a derived attribute updated by invocation of the `set_status` operation from relevant entry actions.

```
attribute _n2 = 0 : N2Type;
```

The counter of N2 T1 time-outs is not needed in the idle state. However it is easier to specify it for all states.

Operations

set_status

The published status is maintained by invocation of `set_status` by relevant entry actions.

```
operation set_status(in channelStatus : ChannelStatus) : void
{
    _status := channelStatus ;
};
```

s*XX*

It is convenient to use a short form such as `sDM(pf)` for action code on the statechart. These operations are defined as:

```
operation sDISC(in pf : Pf) : void { sendU(TxU(DISC, pf)); };
operation sDM(in pf : Pf) : void { sendU(TxU(DM, pf)); };
operation sFRMR(in pf : Pf) : void { sendU(TxU(FRMR, pf)); };
```

```
operation sREJr(in pf : Pf) : void { sendU(TxS(REJr, pf)); };
operation sRRc(in pf : Pf) : void { sendU(TxS(RRc, pf)); };
operation sRRr(in pf : Pf) : void { sendU(TxS(RRr, pf)); };
operation sSABM(in pf : Pf) : void { sendU(TxU(SABM, pf)); };
operation sUA(in pf : Pf) : void { sendU(TxU(UA, pf)); };
```

resetN2

The N2 time-out is restarted by invocation of `resetN2` from transition actions.

```
operation resetN2() : void
{
    n2 := 0;
};
```

advanceN2

N2 is advanced after each T1 expiry. After N2 advances the `reachedN2` internal event is generated.

```
operation advanceN2() : void
{
    par
    {
        if (n2 + 1 >= N2) then { reachedN2(); };
        n2 := n2 + 1;
    };
};
```

State-specific state variables

The FRAME_REJECT state requires a cache to support emission of the poll/fail accompanying the entry transition as part of the exit action.

```
state FRAME_REJECT
{
   attribute _pf = NO_PF : Pf;
};
```

State-specific operations

entryS3

Entry to S3 requires multiple actions, which are expressed as an operation to reduce diagram clutter.

```
state FRAME_REJECT
{
    operation entryS3(in pf : Pf) : void
    {
        _pf := pf;
        resetN2();
    };
};
```

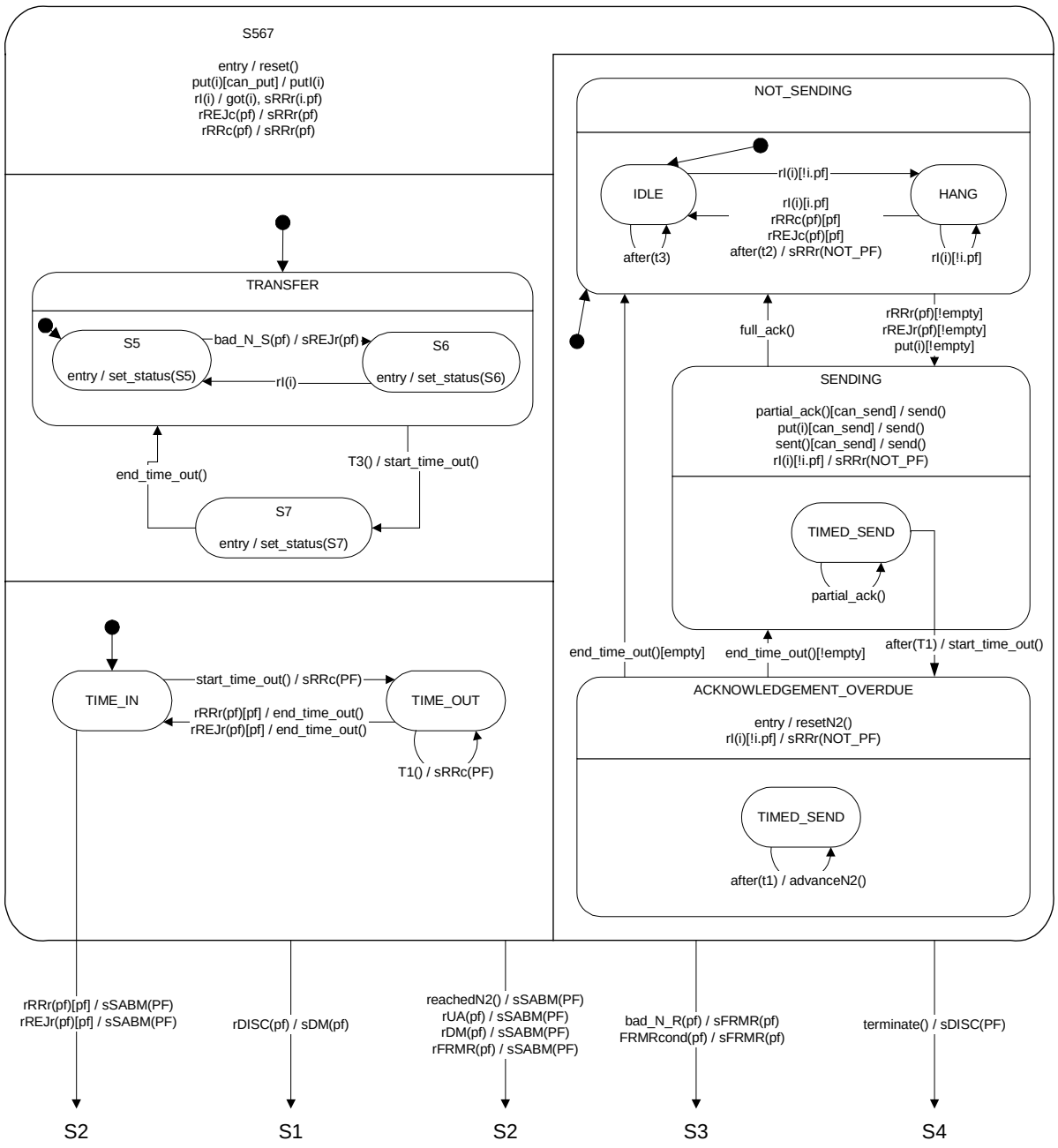
t1inS3

Handling a T1 time-out in S3 requires multiple actions, which are expressed as an operation to reduce diagram clutter.

```
state FRAME_REJECT
{
    operation t1inS3() : void
    {
        if (_n2 < n2) then { sFRMR(_pf); }
        else { reachedN2(); };
    };
};
```

4.5.2 Fm3Tr.Dlc.ArqFsm.Traffic

The inner state machine supports the 3 specified S5, S6 and S7 states, although these states do not adequately represent the required behaviour and do not form part of the base AX25 specification. The exposition here is therefore an attempt to express as much detail as possible of the FM3TR specification in spite of its many contradictions and ambiguities. It should be recognised that this area of the specification should be rewritten.



The TIME_IN/TIME_OUT machine is intended to provide the unified behaviour for the distinct T1 and T3 time-outs. (A T1 time-out occurs if messages fail to be acknowledged, whereas a T3 time-out occurs to tick over when nothing is happening). A revision of the design to something sensible should manage to eliminate one or both in favour of a

better-defined S7.

The right-hand machine manages the sending of Information packets, and comprises three high level states corresponding to nothing being sent, something being sent with acknowledgement expected, and something being sent with acknowledgement overdue. Each of these has sub-states.

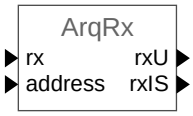
> Note that this partitioning enables T1, T2 and T3 to be realised purely by `after` transitions that are timed with respect to entry into the source state of the transition. N2 is also localised to a state, which is responsible for starting or resetting and killing it. There is no cross-control, merely hierarchical reaction to time-outs.

Ports

Two internal events are required to support communication between the concurrent state machines.

```
internal:event start_time_out : void;
internal:event end_time_out : void;
```

4.5.3 Fm3tr.Dlc.ArqRx



Description

The ArqRx entity specifies the first stage of per-channel ARQ receive processing. Inappropriate `rx` messages are discarded by validating the remote `address`. The received data of correctly addressed messages is passed on by generating either a `rxIS` or `rxU` message.

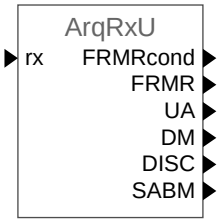
Ports

```
in:event rx : RxARQ;
in:value address : Address;
out:event rxIS : RxIS;
out:event rxU : RxU;
```

Responses

```
response rx
{
    if (rx.source_address = address) then
    {
        layin (rx)
        {
            case RxI, RxS: { rxIS(rx); };
            case RxU: { rxU(rx); };
        };
    };
};
```

4.5.4 Fm3tr.Dlc.ArqRxU



Description

The ArqRxU entity completes per-channel ARQ receive interpretation of Unnumbered received messages, causing an appropriate event message to be generated. If the message length is wrong or corresponds to an unrecognised command or response a FRMRcond event is produced. Otherwise the corresponding command or response is generated.

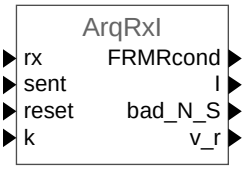
Ports

```
in:event rx : RxU;
out:event FRMRcond : Pf;
out:event SABM : Pf;
out:event DISC : Pf;
out:event DM : Pf;
out:event UA : Pf;
out:event FRMR: Pf;
```

Responses

```
response rx
{
    layin (rx)
    {
        case RxSABM: { SABM(rx.pf); };
        case RxDISC: { DISC(rx.pf); };
        case RxDM: { DM(rx.pf); };
        case RxUA: { UA(rx.pf); };
        case RxFRMR: { FRMR(rx.pf); };
        default: { FRMRcond(rx.pf); };
    };
};
```

4.5.5 Fm3tr.Dlc.ArqRxI



The ArqRxI entity completes per-channel validation of Information messages. Excessive length or an out-of-sequence receive results in a `FRMRcond`. A wrapped around receive sequence results in a `bad_N_S`. Otherwise the data is passed on as an `I` message.

Two local state variables are required to keep track of the expected sequence. The sequence number of the next received information message is made available as `v_r` for incorporation in transmissions as an acknowledgement of reception. The second variable is maintained locally for overflow checking and identifies the last value of `v_r` that was acknowledged. The variables are reset at start up or by a `reset` message.

Ports

```
in:value k : KType;
in:event rx : RxI;
in:event reset : void;
in:event sent : KType;
out:event FRMRcond : Pf;
out:event bad_N_S: Pf;
out:event I: RxI;
out:value v_r = _v_r : KType;
```

State variables

```
attribute last_tx_N_R = 0 : KType;
attribute _v_r = 0 : KType;
```

Responses

A `reset` resets the state variable.

```
response reset
{
    par
    {
        last_tx_N_R := 0;
        _v_r := 0;
    };
};
```

A `sent` updates the last transmitted N(R) state variable.

```
response sent
{
    last_tx_N_R := sent;
};
```

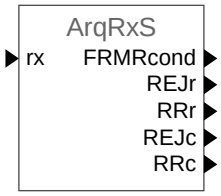
A receive validates and dispatches the packet.

```
response rx
{
    if (rx.n_s != _v_r) then { FRMRcond(rx.pf); }
    else if ((rx.n_s - last_tx_N_R) >= k) then { bad_N_S(rx.pf); }
    else
    {
        par
        {
            I(rx);
            _v_r := rx.n_s;
        };
```

```
        };
    };
```

The comparison test differs from the equality test in the FM3TR specification in order to achieve predictable behaviour if k is changed during operation.

4.5.6 Fm3tr.Dlc.ArqRxS



The ArqRxS entity completes validation of received Supervisory messages. Illegal length or unrecognised messages result in a FRMRcond. Valid messages result in the corresponding event.

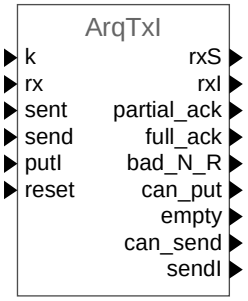
Ports

```
in:event rx : RxS;
out:event FRMRcond : Pf;
out:event RRc : Pf;
out:event REJc : Pf;
out:event RRr : Pf;
out:event REJr : Pf;
```

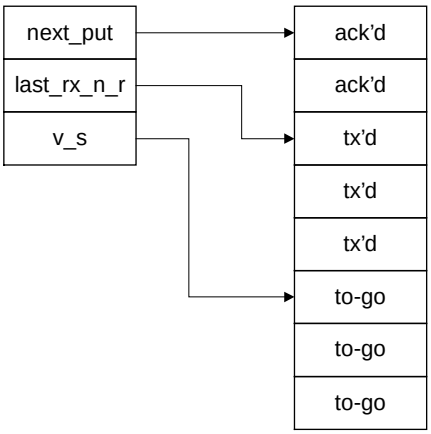
Responses

```
response rx
{
    layin (rx.format)
    {
        case RxRRc: { RRc(rx.pf); };
        case RxREJc: { REJc(rx.pf); };
        case RxRRr: { RRr(rx.pf); };
        case RxREJr: { REJr(rx.pf); };
        default: { FRMRcond(NO_PF); };
    };
};
```

4.5.7 Fm3tr.Dlc.ArqTxI



The transmit information packet manager maintains the at least k+1 (8) element buffer of transmitted packets and associated status variables. These are shown informally (not WDL).



The 8 element buffer contains Information messages with three distinct contexts. Messages are added by a `putI` event, and dispatched to the transmit queue by a `sendI` event in response to a `send`. A `sent` message signals that transmission has begun. Acknowledgement is detected by intercepting the `rxIS` receive events before passing them on as `rxI` or `rxS` messages. Those between `v_s` and `last_rx_n_r` have been transmitted but have not been acknowledged. Those between `last_rx_n_r` and `next_put` have been transmitted and acknowledged and so may be overwritten. Those between `next_put` and `v_s` are awaiting transmission. The three 'pointers' identify the thresholds by index into the buffer. Each pointer advances (downwards on the diagram) using modulo wraparound within the buffer.

Ports

```
in:value k = MAX_K : KType;
in:event reset : void;
in:event putI : Info;
in:event rx : RxIS;
in:event send : void;
in:event sent : KType;
out:event bad_N_R : Pf;
out:event rxI : RxI;
out:event rxS : RxS;
out:event full_ack : void;
out:event partial_ack : void;
out:event sendI : TxI;
```

The three value outputs provide context for use by the `ArqFsm.Traffic` transition guards.

```
out:value can_put = next_put != last_rx_n_r : boolean;
out:value empty = v_s = next_put : boolean;
out:value can_send = (v_s != next_put)
                        & ((v_s - last_rx_n_r) < k) : boolean;
```

`can_put` indicates whether the buffer can process a `putI` event.

`empty` indicates whether transmission should shut down.

`can_send` indicates whether the buffer can process a `send` event.

The comparison test differs from the equality test in the FM3TR specification in order to achieve predictable behaviour if k is changed during operation.

State Variables

The state variables comprise the buffer of messages and the three context pointers.

Note that indexes into the buffer are defined using modulo types, so arithmetic automatically wraps around without the need for explicit modulus operations in the subsequent equations.

```
attribute buffer = 0 : Info[MAX_K+1];
attribute v_s = 0 : KType;
attribute last_rx_n_r = 0 : KType;
attribute next_put = 0 : KType;
```

`v_s` is V(S) as defined in FM3TR; the N(S) in the next I message: the waiting-to-transmit threshold.

`last_rx_n_r` is the N(R) received in the last I or S message: the unacknowledged threshold.

`next_put` is the ready-to-put threshold. This does not form part of FM3TR, however there has to be some buffer for waiting to go messages and it seems sensible to use the dead part of the existing buffer. This avoids a discrepancy between first transmit and retransmit data sources.

Responses

reset

The pointer variables are reset to the buffer start in response to a `reset`. There is no need to reset the buffer contents as well.

```
response reset
{
    par
    {
        v_s := 0;
        last_rx_n_r := 0;
        next_put := 0;
    };
};
```

putI

A `putI` event puts an Information message into the buffer ready for transmission. Invocation of `putI` from ArqFsm is guarded by `can_put`.

```
response putI
{
    par
    {
        buffer[next_put] := putI;
        next_put := next_put + 1;
    };
```

```
};
```

send

When ArqFsm requests transmission, the next entry together with its V(S) is formatted for transmission. (Addresses are added by ArqTx and MacTx).

```
response send
{
    sendI(TxI(NOT_PF, buffer[v_s], v_s));
};
```

sent

Since there may be a significant queuing conflict between channels, update of V(S) must be deferred until the message is selected for transmission. `sent` is invoked from ArqTx to signal that transmission has started and will necessarily finish.

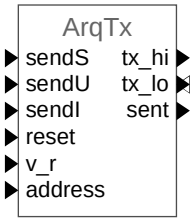
```
response sent
{
    v_s := v_s + 1;
};
```

rx

A received Information or Supervisory message contains an N(R) field to acknowledge previous transmissions. An `rx` event notifies the transmit buffer manager of the received message. If the received N(R) corresponds to a repeat or wraparound, a `bad_N_R` is generated. Otherwise either a `partial_ack` or `full_ack` is generated in accordance with how much of an acknowledgement has occurred. The valid N(R) is remembered and the message propagated on by generating either an `rxI` or `rxS` event.

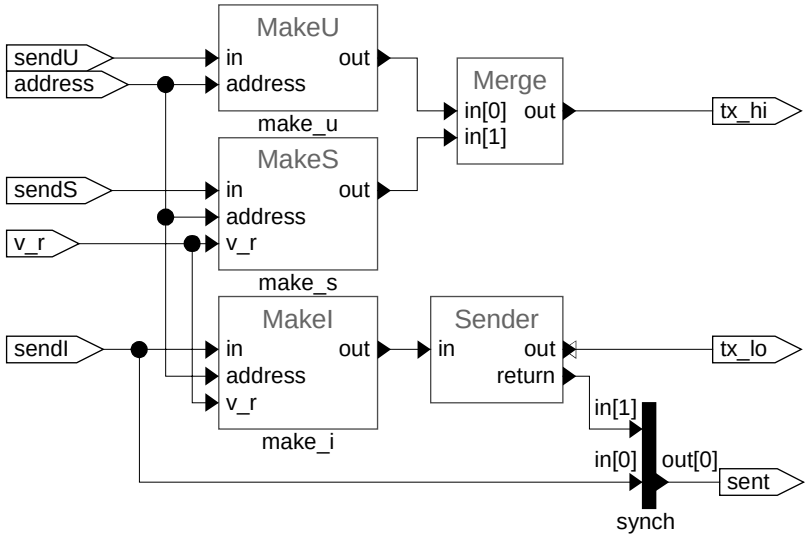
```
response rx
{
    if (rx.n_r = last_rx_n_r) then { bad_N_R(rx.pf); }
    else
    {
        par
        {
            if (rx.n_r != v_s) then { partial_ack(); }
            else { full_ack(); };
            last_rx_n_r := rx.n_r;
            layin (rx)
            {
                case RxI: { rxI(rx); };
                case RxS: { rxS(rx); };
                default: { FRMRcond(rx.pf); };
            };
        };
    };
};
```

4.5.8 Fm3tr.Dlc.ArqTx



The ArqTx entity specifies the expansion of message content from the minimal information supplied by state machine transitions to the full context associated with a particular ARQ channel.

Messages are partitioned into high and low priority to satisfy the indication that control messages should be sent promptly. The FM3TR specification is unclear on this issue. The specification here is therefore a suggestion.



The V(S) state variable is maintained by the `ArqTx` entity and incorporated as part of the `TxI` data type that forms the `sendI` flow. This value is incorporated into the `sent` message for by the acknowledgement checking in the receive path.

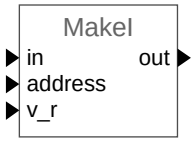
Ports

```
in:token sendI : TxI;
in:token sendS : TxS;
in:token sendU : TxU;
in:value v_r : KType;
in:value address : Address;
in:event reset : void;
out:token tx_hi : TxM;
out:token tx_lo : TxM;
out:event sent : KType;
```

Configuration

```
constraint: synch.out[0] = synch.in[0].n_s;
```

4.5.8.1 Fm3tr.Dlc.ArqTx.MakeI

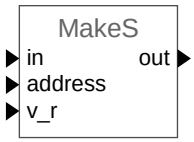


The `MakeI` entity specifies the assembly of the information fields to characterise an Information message.

Ports

```
in:token in : TxI;
in:value address : Address;
in:value v_r : KType;
out:token out = TxMessage(in, v_r, address) : TxMessage;
```

4.5.8.2 Fm3tr.Dlc.ArqTx.MakeS

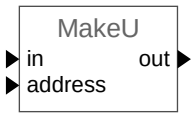


The `MakeS` entity specifies the assembly of the information fields to characterise a Supervisory message.

Ports

```
in:token in : TxS;
in:value address : Address;
in:value v_r : KType;
out:token out = TxMessage(in, v_r, address) : TxMessage;
```

4.5.8.3 Fm3tr.Dlc.ArqTx.MakeU

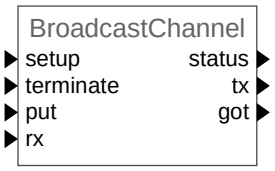


The `MakeU` entity specifies the assembly of the information fields to characterise an Unnumbered message.

Ports

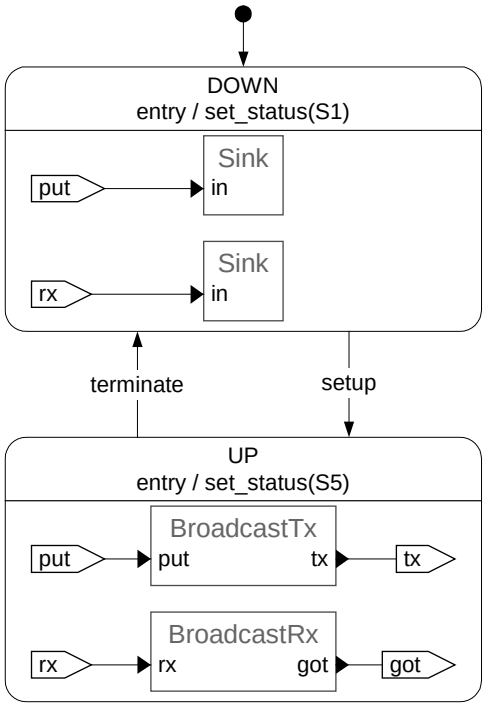
```
in:token in : TxU;
in:value address : Address;
out:token out = TxMessage(in, address) : TxMessage;
```

4.6 Fm3tr.Dlc.BroadcastChannel



The Broadcast channel provides for communication of Unnumbered Information frames. A much simpler protocol is used in comparison to the ARQ channel; there is no guarantee of reception.

The protocol is not defined by the FM3TR specification, and it is unclear whether the channel actually needs to be setup at all, and how many of the S1 to S7 states should be supported. The specification here is therefore a suggestion.



The broadcast channel is either up or down according to the most recent setup or terminate message. While down, all transmission and reception messages are discarded. While up, messages are passed through.

Ports

```
in:event setup : void;
in:event terminate : void;
in:token put : PutPacket;
in:token rx : RxUI;
out:token tx : TxUI;
out:token got : GotPacket;
out:value status = _status : ChannelStatus;
```

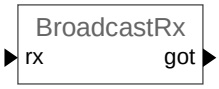
State Variables

```
attribute _status = S1 : ChannelStatus;
```

Operations

```
operation set_status(in channelStatus : ChannelStatus) : void
{
    _status := channelStatus;
};
```

4.6.1 Fm3tr.Dlc.BroadcastRx

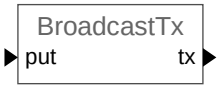


The BroadcastRx entity extracts the information content of Unnumbered Information frames for the Broadcast channel.

Ports

```
in:token rx : RxUI;
out:token got = GotPacket(rx.data) : GotPacket;
```

4.6.2 Fm3tr.Dlc.BroadcastTx

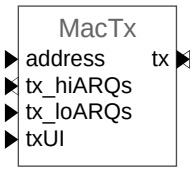


The BroadcastTx entity provides the local content of Unnumbered Information frames for the Broadcast channel.

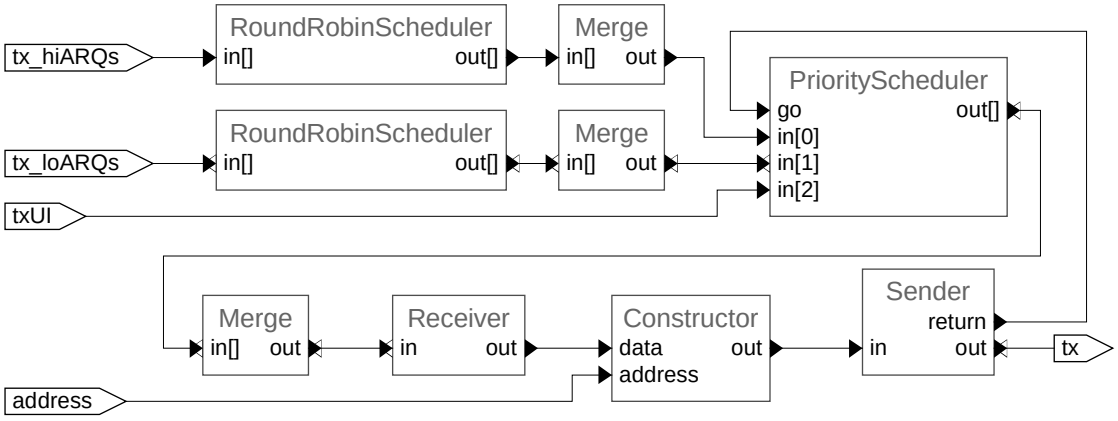
Ports

```
in:token put : PutPacket;
out:token tx = TxUI(put) : TxUI;
```

4.7 Fm3tr.Dlc.MacTx



The transmit interface to the MAC layer selects messages for transmission from the 11 logical channels, preferring high priority Supervisory messages.



The FM3TR specification does not specify how the conflicting requirements of the different channels should be resolved. The specification here is therefore a suggestion.

The high priority (Supervisory and Unnumbered) and low priority (Information) messages are independently selected in round-robin fashion to give fair service to each channel. The actual message to be transmitted is selected by giving priority first to high priority ARQ messages, then to low priority ARQ messages and finally to the broadcast channel.

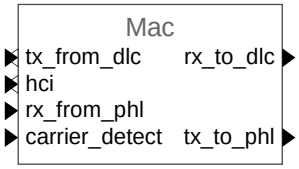
The local address is incorporated in the outgoing message.

This specification makes extensive use of return flows, which are ill-specified in WDL at present. Indeed this entity is the primary motivating influence. The `Sender` generates an outgoing flow, which when returned triggers the `PriorityScheduler` to select the next message. The `RoundRobinSchedulers` do not need triggering, since they can pre-select. However the `tx_loARQs` also involves a return flow, so there a significant number of entities that are exhibiting a requirement for polymorphism between simple and returned flows.

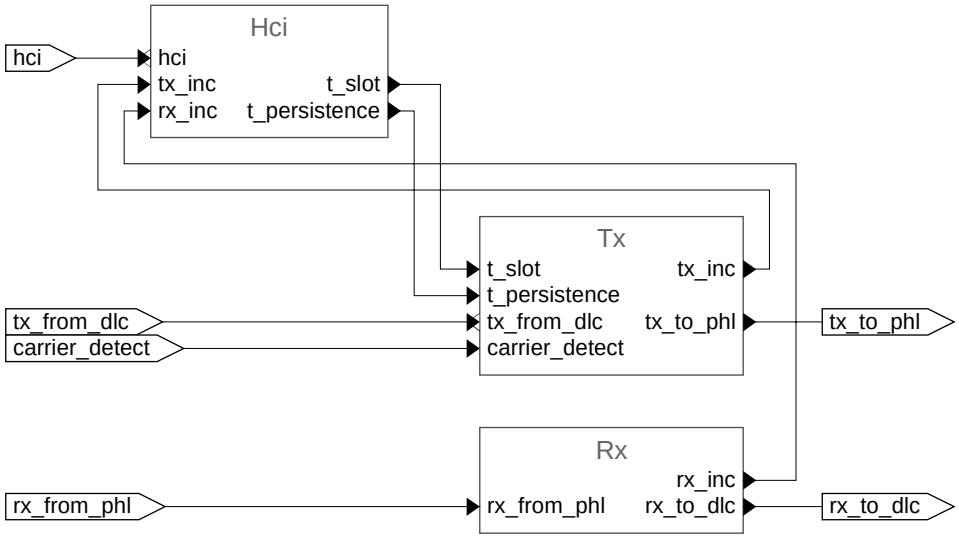
Ports

```
in:value address : Address;
in:token tx_hiARQs : TxMessage[ARQ_CHANNELS];
in:token tx_loARQs : TxMessage[ARQ_CHANNELS] : void;
in:token txUI : TxMessage;
out:token tx : DlcPacket : void;
```

5 MAC layer



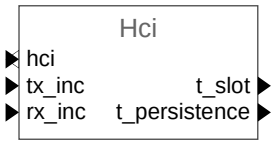
The MAC layer uses a pCSMA (Persistent Carrier Sense Multiple Access) algorithm to ameliorate the consequences and costs of transmit collisions. An Hci interface supports control and interrogation over the slot-time and persistence parameters of the algorithm and maintains receive and transmit packet counters. The Tx entity specifies the pCSMA algorithm. Rx and Tx entities monitor passing packets.



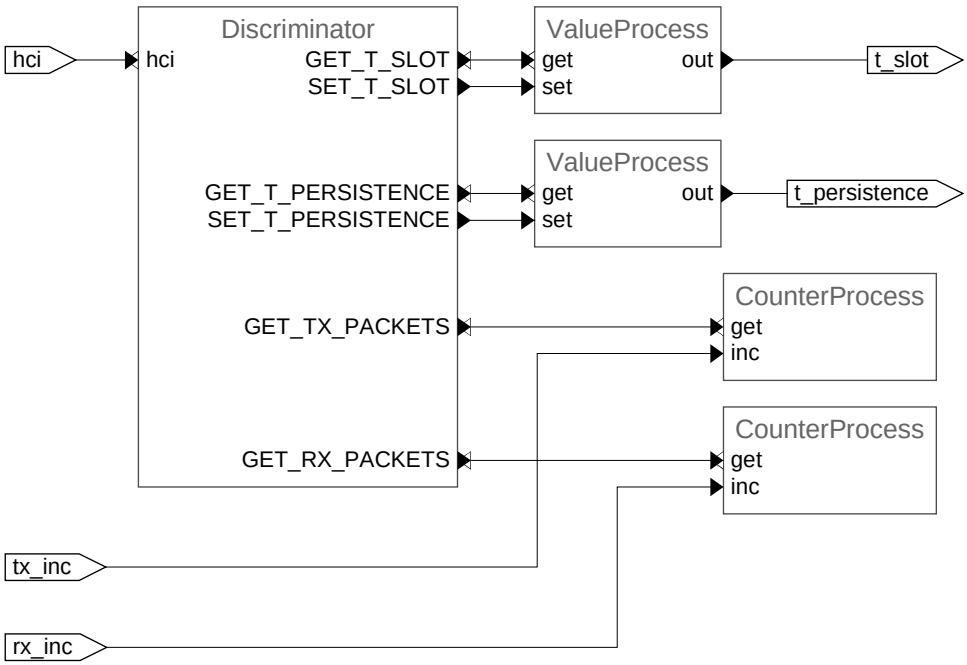
Ports

```
in:event hci : HciToMacMessages;
in:token tx_from_dlc : DlcPacket : void;
in:value carrier_detect : CarrierDetect;
in:event rx_from_phl : MacPacket;
out:event tx_to_phl : MacPacket;
out:event rx_to_dlc : DlcPacket;
```

5.1 Fm3Tr.Mac.Hci



The `Discriminator` splits the messages from the HCI into the messages to interact with the 4 processes maintaining state variables.

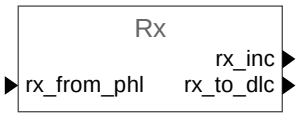


The slot time and persistence parameters are made available continuously by instances of ValueProcess. Transmit and receive packet counters are maintained by instances of CounterProcess responding to `rx_inc` and `tx_inc` events.

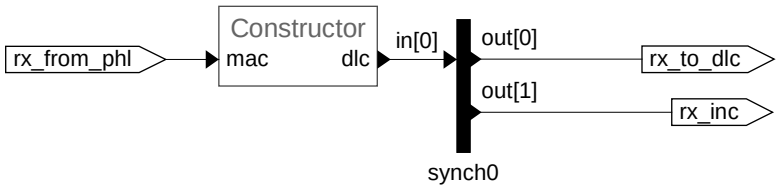
Ports

```
in:event hci : HciToMacMessages;
in:event tx_inc : void;
in:event rx_inc : void;
out:value t_slot : SlotTime;
out:value t_persistence : PersistenceTime;
```

5.2 Fm3tr.Mac.Rx



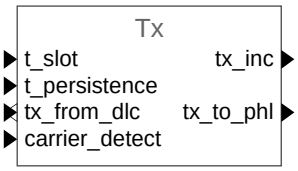
The MAC receive process ensures that an increment counter message is generated as each receive packet passes through the layer.



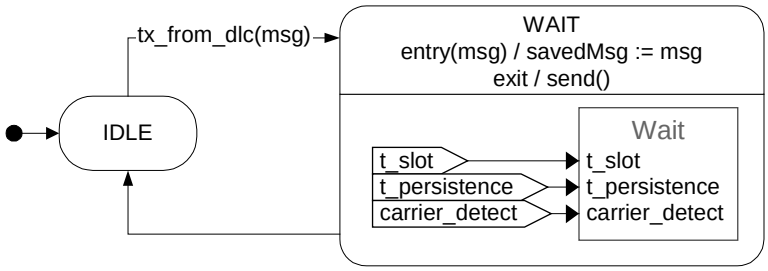
Ports

```
in:event rx_from_phl : MacPacket;
out:event rx_to_dlc : DlcPacket;
out:event rx_inc : void;
```

5.3 Fm3tr.Mac.Tx



The pCSMA functionality involves two states, only one of which has any associated behaviour. Receipt of `tx_from_dlc` changes to the active (waiting) state caching the message for emission as part of a `tx_to_phl` message when the waiting state is done.



Ports

```
in:token tx_from_dlc : DlcPacket : void;
out:event tx_to_phl : MacPacket;
in:value carrier_detect : CarrierDetect;
in:value t_slot : SlotTime;
in:value t_persistence : PersistenceTime;
```

Operations

On completion of the wait, acknowledgement is send back to the DLC layer and the packet is sent to the PHL layer..

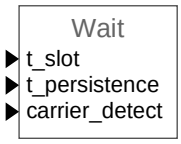
```
operation send() : void
{
    par
    {
        tx_to_phl(savedMsg);
        tx_from_dlc();
    };
};
```

State variables

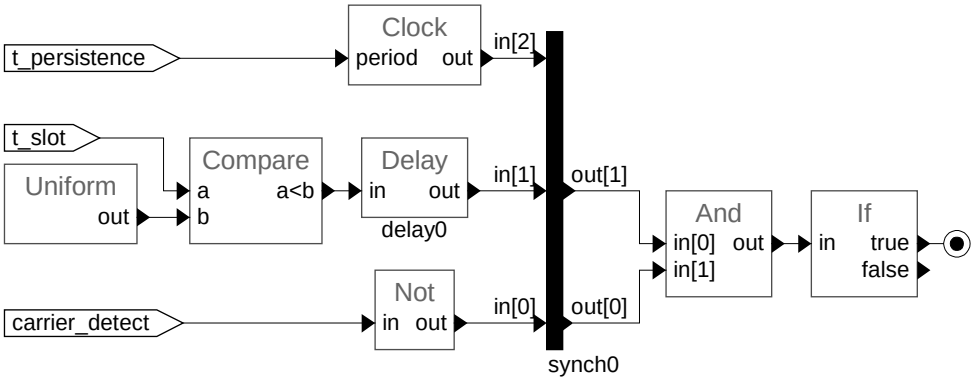
The triggering message must be saved for emission as the state exits.

```
state WAIT
{
    attribute savedMsg = 0 : DlcPacket;
};
```

5.4 Fm3tr.Mac.Wait



The wait message flow defines the polling and calculations for the pCSMA algorithm.



The synchronisation bar enforces the real time clocking characteristic necessary to poll the `carrier_detect` value at an interval determined by the `t_persistence` value.

The first poll occurs instantly, using a true value pre-loaded into the single element delay and the prevailing state of `carrier` to determine whether transmission may proceed. If the result of the And is true, the message flow to the end state symbol occurs causing an exit from the parent state, which in this case causes Fm3tr.Mac.Tx.Wait to complete triggering the transmission and state change to Fm3tr.Mac.Tx.IDLE.

Subsequent polls occur in response to the flow rate permitted by the synchronisation bar, that is one iteration every `period` time units, using the uniformly generated random number to determine whether a transmission attempt may be made.

Ports

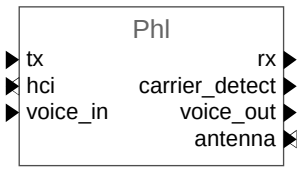
```
in:value carrier_detect : CarrierDetect;
in:value t_slot : SlotTime;
in:value t_persistence : PersistenceTime;
```

Configuration

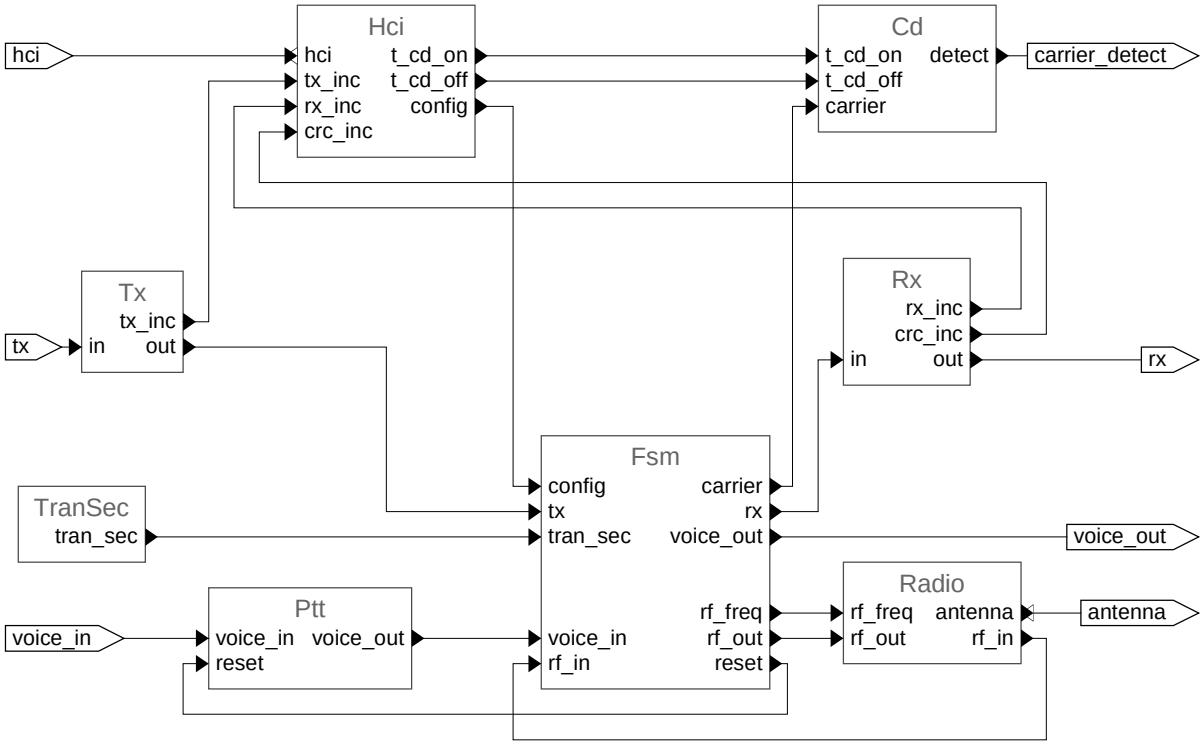
```
constraint: delay0.initial_value = true;
constraint: delay0.size = 1;
```

The `uniform` generator is automatically parameterised by propagation of the minimum and maximum values of the data type of its output, which is constrained by the comparator to match the data type of the `t_slot` flow.

6 PHL layer



The PHL layer supports transmission of either the CVSD `voice_in` or `tx` data packets and corresponding reception to produce `voice_out` or `rx` packets.



Configuration of the layer is provided through an `hci` message interface that the Hci entity decodes to provide control of carrier detect response and synchronisation robustness. Transmit and receive statistics may be interrogated.

A packet interface is provided by the Rx and Tx entities, with the Cd entity providing a `carrier_detect` signal for use by the pCSMA algorithm in the MAC layer.

Amplification, mixing and multiplexing of transmit and receive signals to the `antenna` is provided by the Radio entity.

Debounce filtering of the Push To Talk input is provided by the Ptt entity.

The TranSec entity is not specified by FM3TR, but it must clearly supply information to the Fsm that switches between the different operational modes of the physical layer.

Ports

```
in:token voice_in : VoiceIn;
in:event hci : HciToPhlMessages;
in:event tx : MacPacket;
out:value carrier_detect : CarrierDetect;
out:event rx : MacPacket;
```

```
        out:token voice_out : VoiceOut;
        inout:signal antenna : Antenna;
```

6.1 Fm3tr.Phl Types

Configuration

```
type PhlTime : Time;
type ModulationDepth : real { minimum 0.5; maximum 0.5; };
```

The Tw type defines the parameters of a specific Test Waveform Number.

```
record Tw
{
    attribute rs1 : SystematicRsParams;
    attribute rs2 : SystematicRsParams;
    attribute tuning_time : PhlTime;
    attribute rise_time : PhlTime;
    attribute fall_time : PhlTime;
    attribute bit_period : PhlTime;
    attribute hop_period : PhlTime;
    attribute mod_index : ModulationDepth;
    attribute codeword1 : boolean;
    attribute codeword2  : boolean;
};
```

The PhlConfig type defines the configuration of the physical layer..

```
record PhlConfig : Tw
{
    attribute is_robust : IsRobust;
};
```

Signals

The Frequency type identifies the channel frequency.

```
type Frequency : real
    { minimum 30‘MHz; maximum 400‘MHz; epsilon 25‘kHz; };
```

The FrequencyShift type identifies the depth of frequency shift modulation.

```
type FrequencyShift : Frequency : real
    { minimum -6.25‘kHz; maximum 6.25‘kHz; epsilon 6.25‘kHz; };
```

The Amplitude type identifies the transmit signal amplitude, for which unit values are a suitable arbitrary choice.

```
type Amplitude : integer { minimum -1; maximum +1; };
```

The Modulation type combines the three modulation attributes, so that they can be passed as a single value.

```
record Modulation
{
    attribute amplitude : Amplitude;
    attribute frequency_shift : FrequencyShift;
    attribute frequency : Frequency;
};
```

The TxSignal type describes the transmit signal, which is unspecified for FM3TR..

```
type TxSignal ;
```

The RxSignal type describes the receive signal, which is unspecified for FM3TR..

```
type RxSignal ;
```

Data

A BitPacket defines the composite data and frequency information fed to the BitModulator.

```
record BitPacket
{
```

```
    attribute bit : boolean;
    attribute frequency : Frequency;
};
```

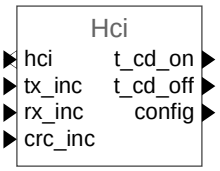
A `HopPacket` defines the composite data and frequency information fed to the `HopModulator`.

```
record HopPacket
{
    attribute hop : boolean[*];
    attribute frequency : Frequency;
};
```

A `SyncPacket` defines the data type of a Code A or Code S packet.

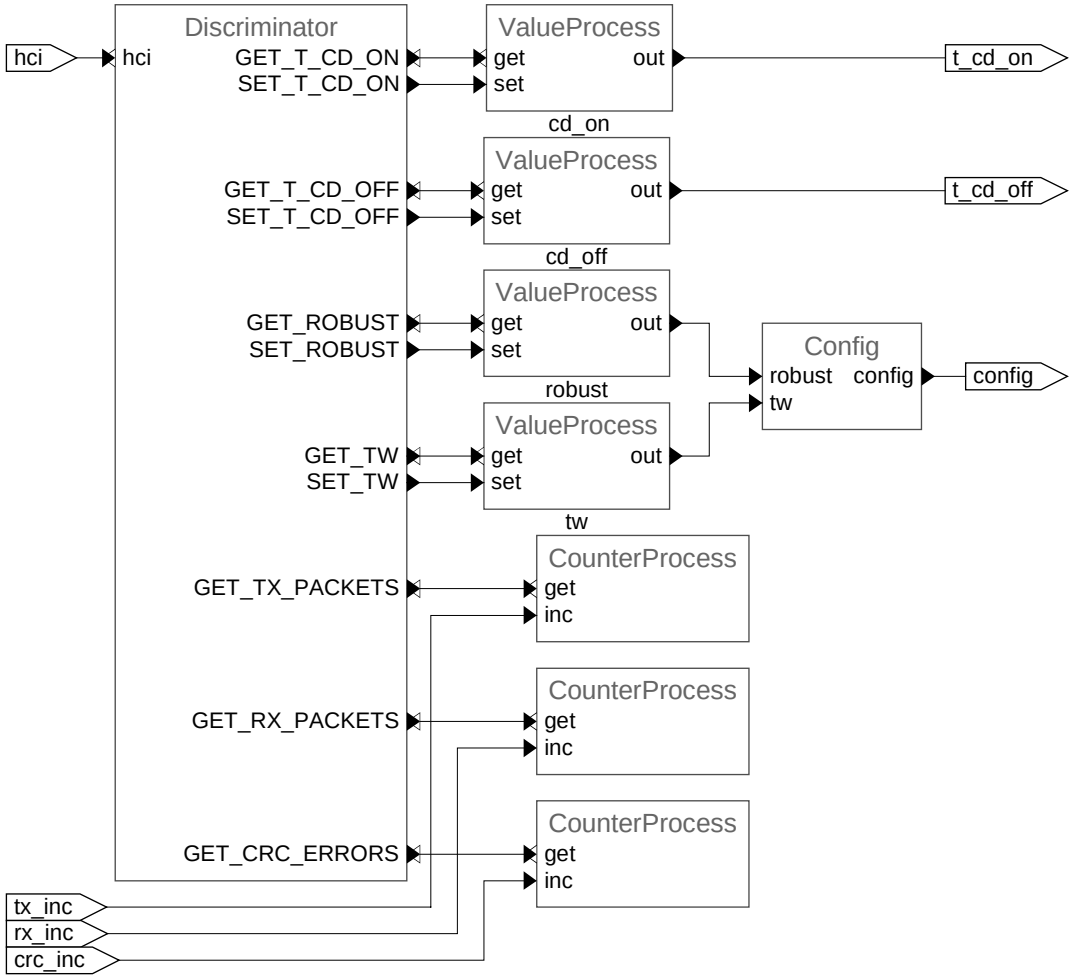
```
type SyncPacket : boolean[32];
```

6.2 Fm3tr.Phl.Hci



The HCI interface for the PHL layer maintains the on/off time configuration parameters for the carrier detector and statistics counters for transmit and receive packets and CRC errors.

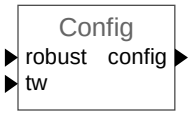
The interface also supports selection of the Test Waveform Number and robust synchronisation mode.



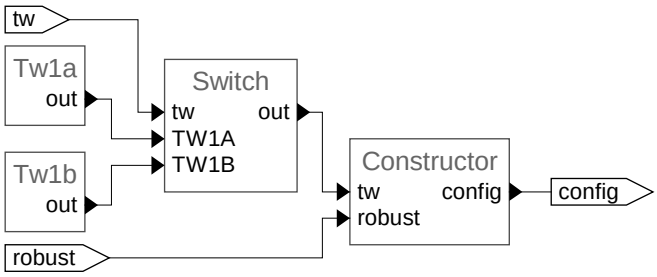
Ports

```
in:event hci : HciToPhlMessages;
in:event tx_inc : void;
in:event rx_inc : void;
in:event crc_inc : void;
out:value t_cd_on : CdTime;
out:value t_cd_off : CdTime;
out:value config : PhlConfig;
```

6.2.1 Fm3tr.Phl.Hci.Config



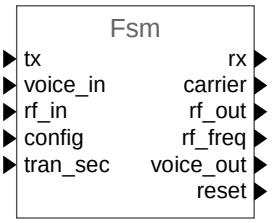
The Config entity specifies the use of either the TW#1a or TW#1b .configuration augmented by the chosen robust/normal setting.



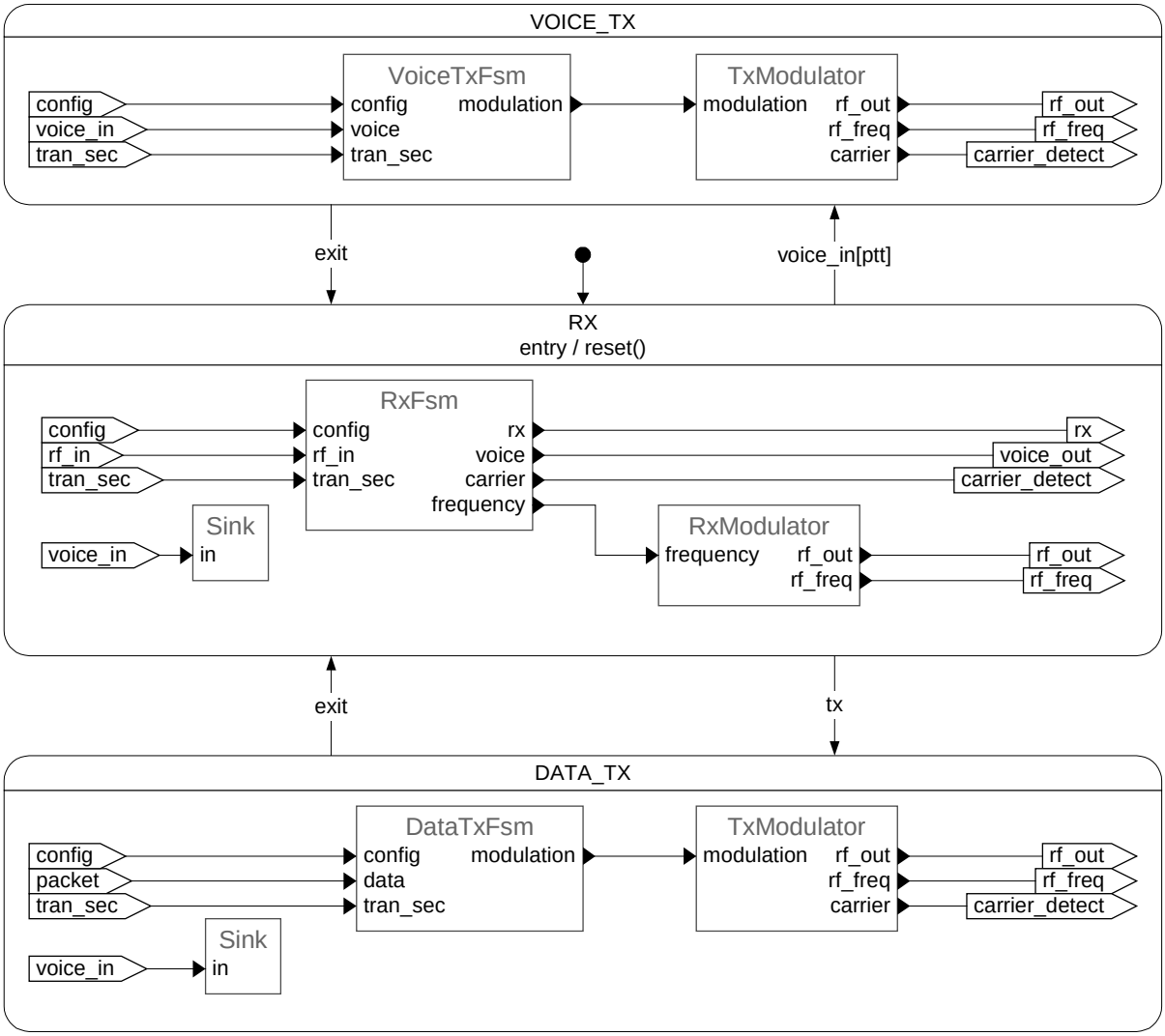
Ports

```
in:value robust : IsRobust;
in:value tw : TwNumber;
out:value config : PhlConfig;
```

6.3 Fm3tr.Phl.Fsm



The `Fsm` entity specifies half duplex operation; the FSM idles in the `RX` state, entering either `VOICE_TX` or `DATA_TX` in response to an appropriate transmit request. Entry to `RX` issues a reset to the `Ptt` monostable. Each state specifies distinct receive and transmit activity.

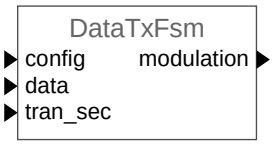


Ports

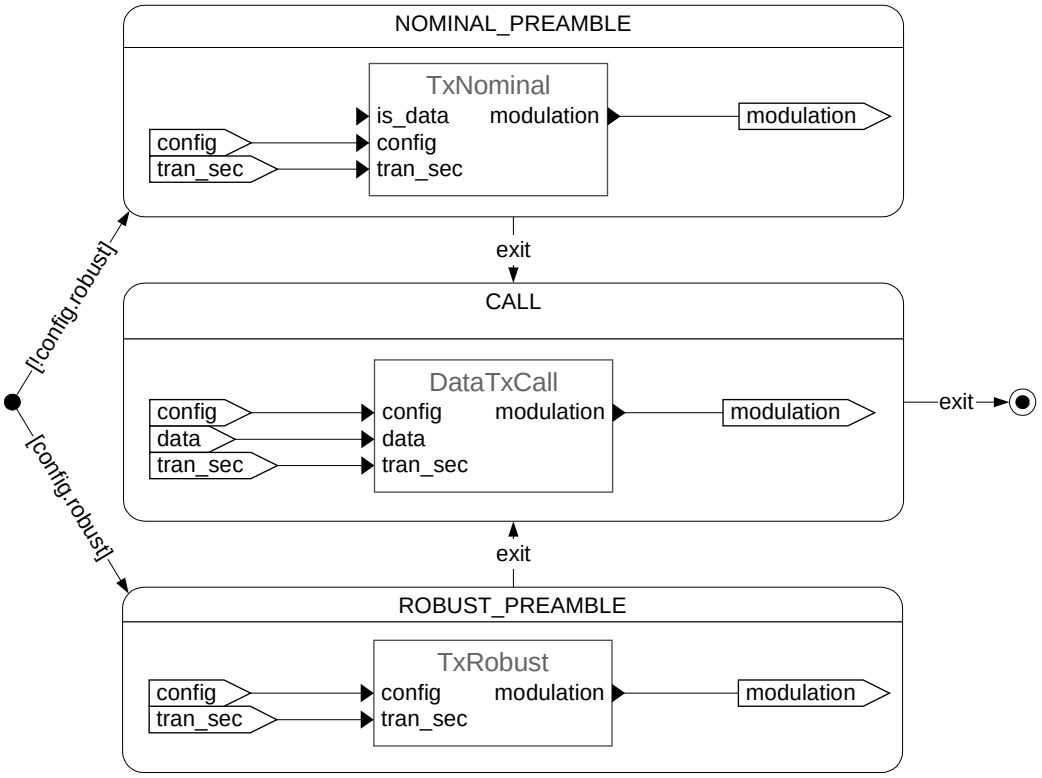
```
in:token tx : ValidBit;
in:token voice_in : VoiceIn;
in:signal rf_in : RxSignal;
in:value config : PhlConfig;
```

```
        in:token tran_sec : TranSec;
        out:token rx : Octet[*];
        out:token carrier : CarrierDetect;
        out:signal rf_out : TxSignal;
        out:signal rf_freq : Frequency;
        out:token voice_out : VoiceOut;
        out:event reset : void;
```

6.4 Fm3tr.Phl.DataTxFsm



The `DataTxFsm` entity specifies the two alternate synchronisation pre-ambles for robust and nominal calls that precede the common data call behaviour.



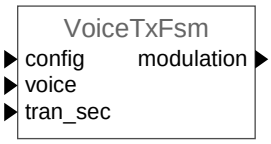
Ports

```
in:value config : PhlConfig;
in:token data : TxPacket;
in:token tran_sec : TranSec;
out:signal modulation : Modulation;
```

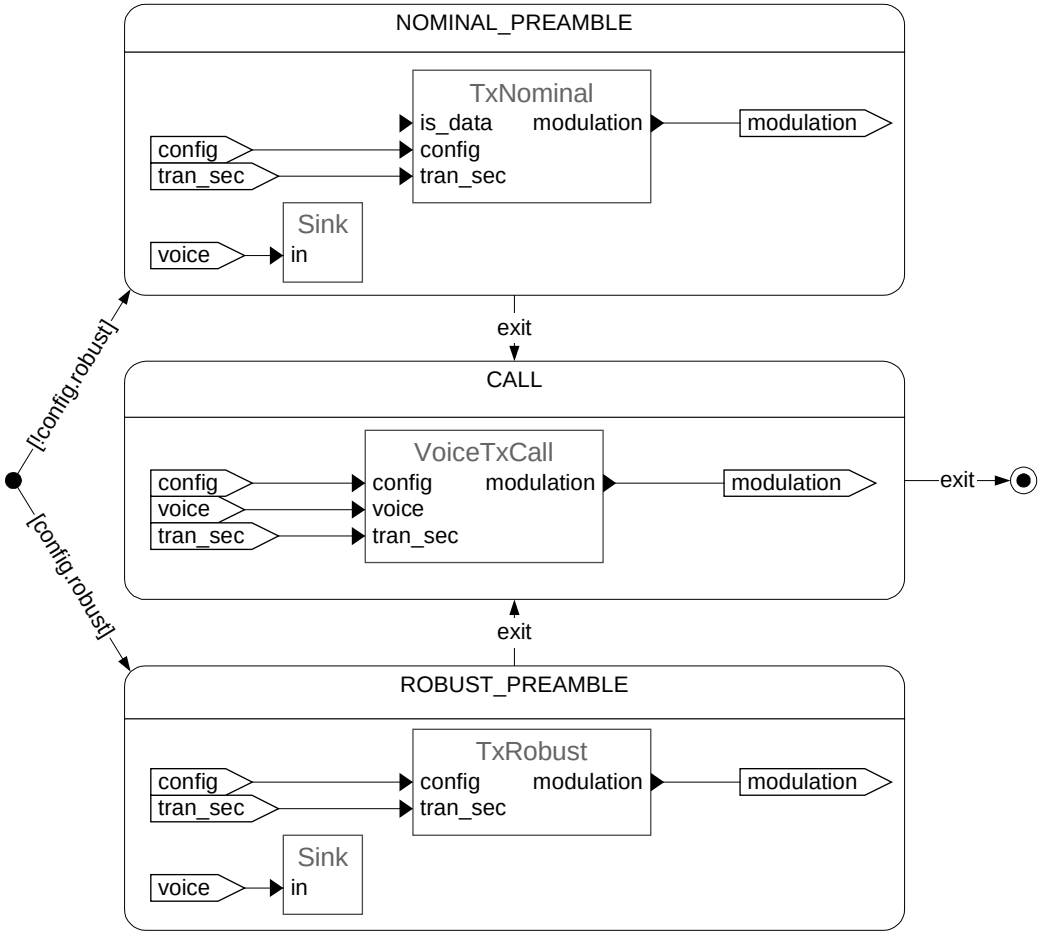
Configuration

```
constraint: nominal.is_data = true;
```

6.5 Fm3tr.Phl.VoiceTxFsm



The `VoiceTxFsm` entity specifies the two alternate synchronisation pre-ambles for robust and nominal calls that precede the common data call behaviour.



The behaviour is similar to the data call, differing mainly in the discard of voice samples during the synchronisation pre-amble.

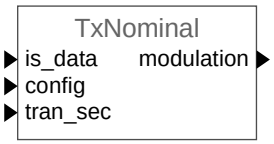
Ports

```
in:value config : PhlConfig;
in:token voice_in : VoiceIn;
in:token tran_sec : TranSec;
out:signal modulation : Modulation;
```

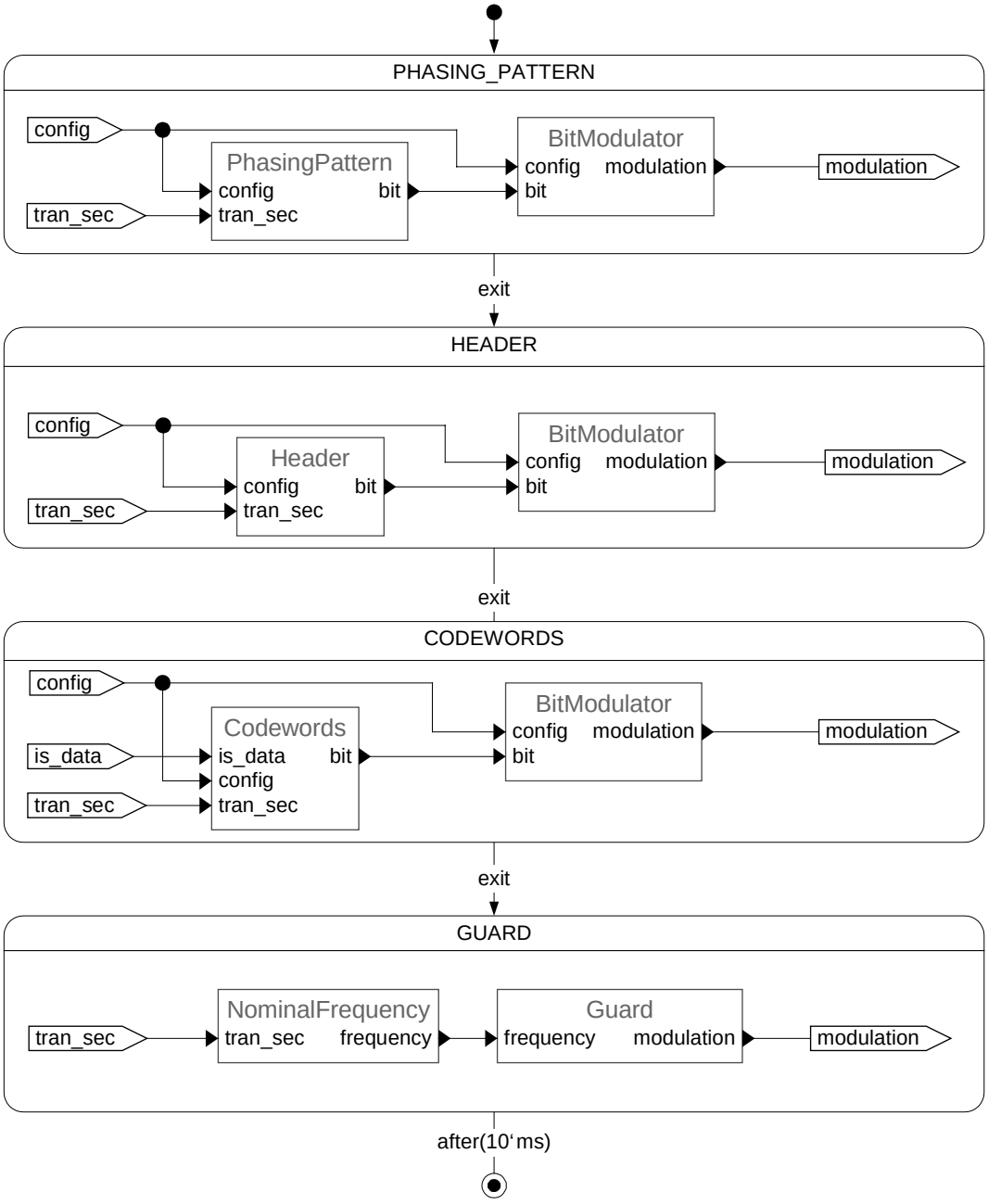
Configuration

```
constraint: nominal.is_data = false;
```

6.6 Fm3tr.Phl.Fsm.TxNominal



`TxNominal` specifies the pre-amble for a nominal (non-robust) call.



The synchronization consists of 2350 bits of reversals followed by the 32 bit CODE_A word, followed by 7 true or complement copies of the CODE_S word that define the subsequent call format. Finally there is a guard time before the pre-amble completes and

the call commences.

The modulation frequency during the guard is not significant. The nominal sync call frequency is arbitrarily selected to ensure a legal value.

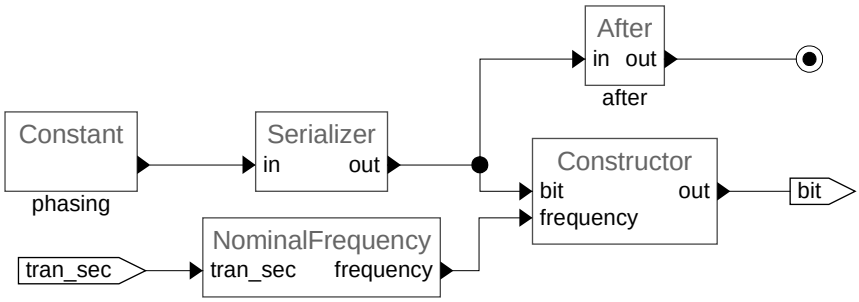
It should be noted that this specification faithfully follows the FM3TR specification; there is no shaping at the ends of the nominal preamble.

Ports

```
in:value is_data : boolean;
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:signal modulation : Modulation;
```

6.6.1 Fm3tr.Phl.Fsm.TxNominal.PhasingPattern

The `PhasingPattern` specifies the phasing reversals that start a nominal (non-robust) call.



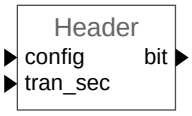
Ports

```
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:token bit : BitPacket;
```

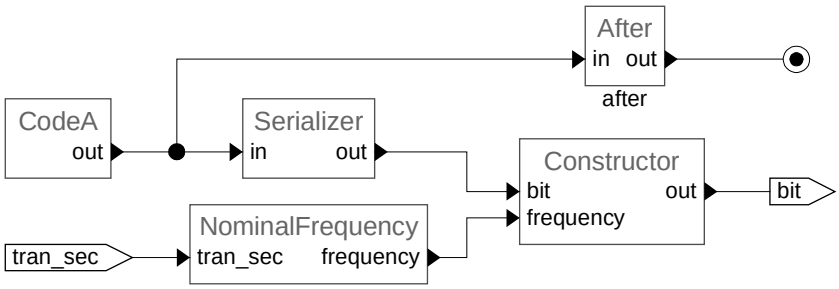
Configuration

```
constraint: phasing.value = [true, false];
constraint: after.value = 2350+1;
```

6.6.2 Fm3tr.Phl.Fsm.TxNominal.Header



The `Header` entity specifies the emission of the A codeword following the alternate phase pre-amble of a nominal (non-robust) call.



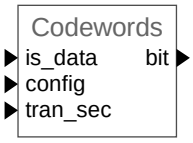
Ports

```
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:token bit : BitPacket;
```

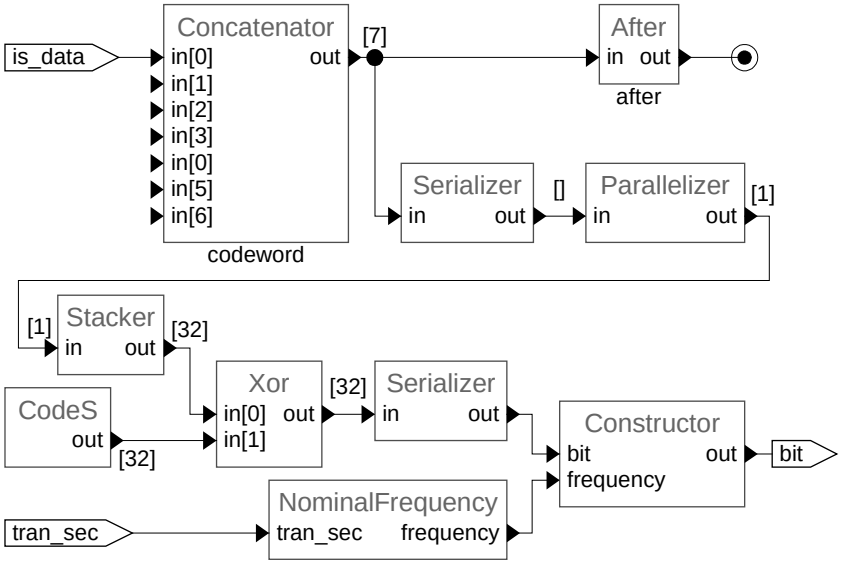
Configuration

```
constraint: after.value = 2;
```

6.6.3 Fm3tr.Phl.Fsm.TxNominal.Codewords



The `Codewords` entity specifies the emission of the 7 repetitions of the S codeword to complete the pre-amble of a nominal (non-robust) call. Each repetition is modulated by one of seven configuration bits.



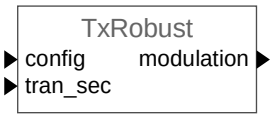
Ports

```
in:value is_data : boolean;
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:token bit : BitPacket;
```

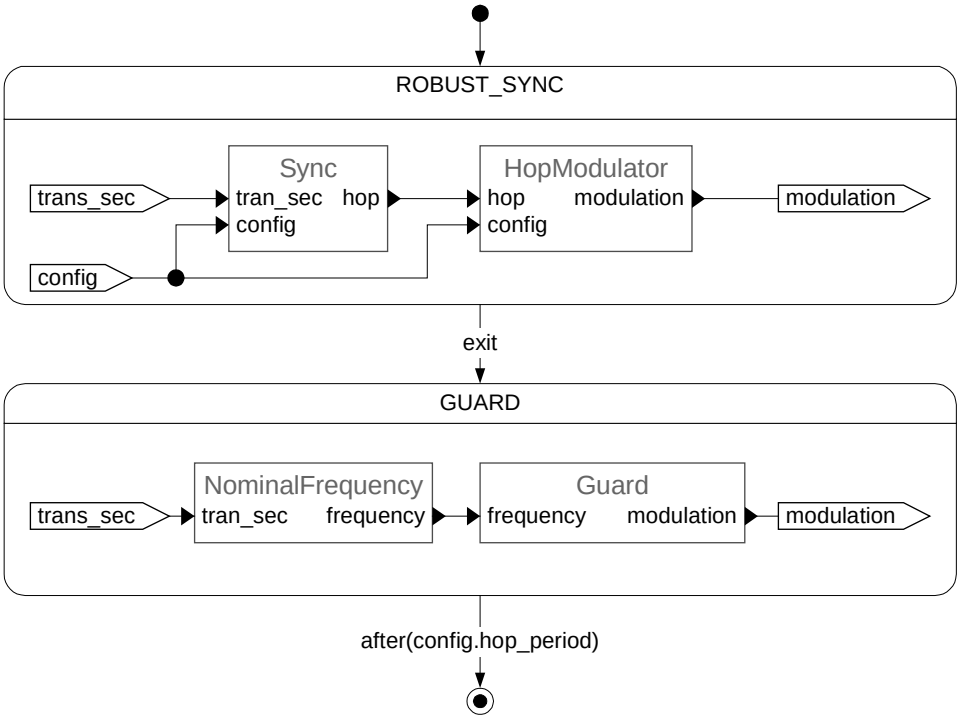
Configuration

```
constraint: codeword.in[1] = config.codeword1;
constraint: codeword.in[2] = config.codeword2;
constraint: codeword.in[3] = false;
constraint: codeword.in[4] = false;
constraint: codeword.in[5] = false;
constraint: codeword.in[6] = false;
constraint: after.value = 2;
```

6.7 Fm3tr.Phl.Fsm.TxRobust



A robust synchronisation call starts with a sequence of 17 iterations of hops over 16 frequencies followed by a guard time.

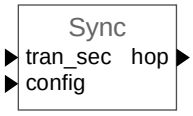


The modulation frequency during the guard is not significant. The nominal sync call frequency is arbitrarily selected to ensure a legal value.

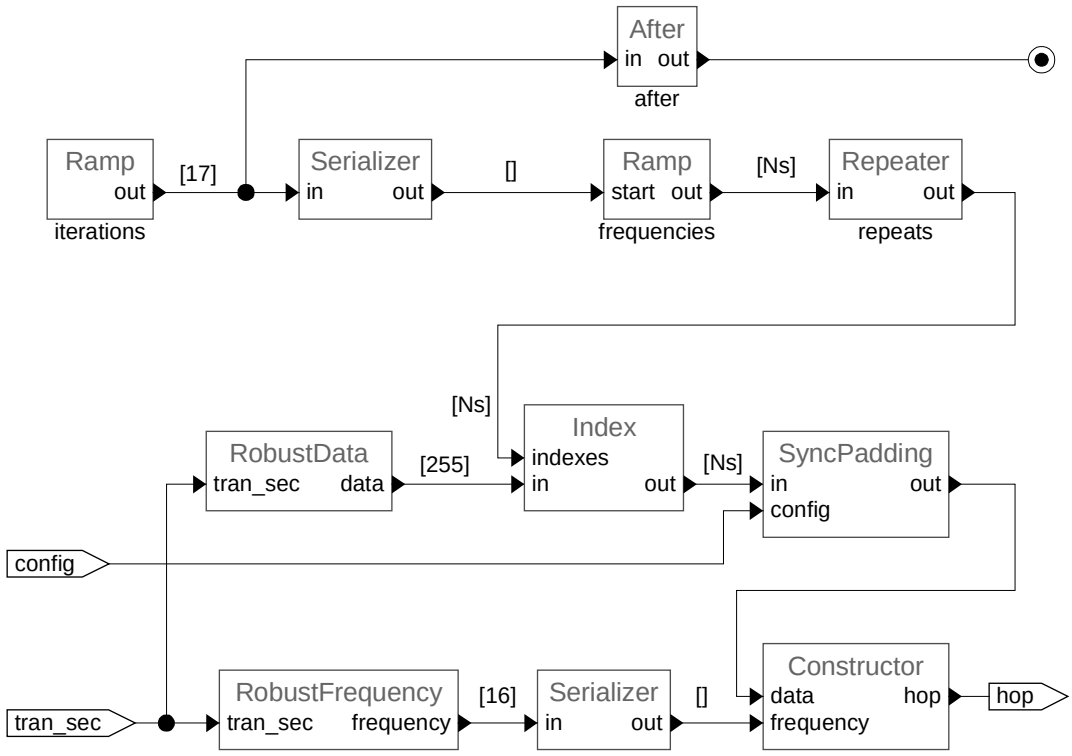
Ports

```
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:signal modulation : Modulation;
```

6.7.1 Fm3tr.Phl.Fsm.TxRobust.Sync



A robust synchronisation call starts with a sequence of 17 iterations of hops over 16 frequencies.



The RobustFrequency entity provides the 16 frequency set, which is serialized so that the resulting hop sequence repeatedly scans through each frequency in turn. The hop comprises the selected frequency and the padded sync data. The sync data comprises a 32/64 bit segment of a 255 bit pattern. The segment start advances by one bit position every 16th hop.

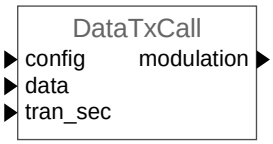
Ports

```
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:token hop : HopPacket;
```

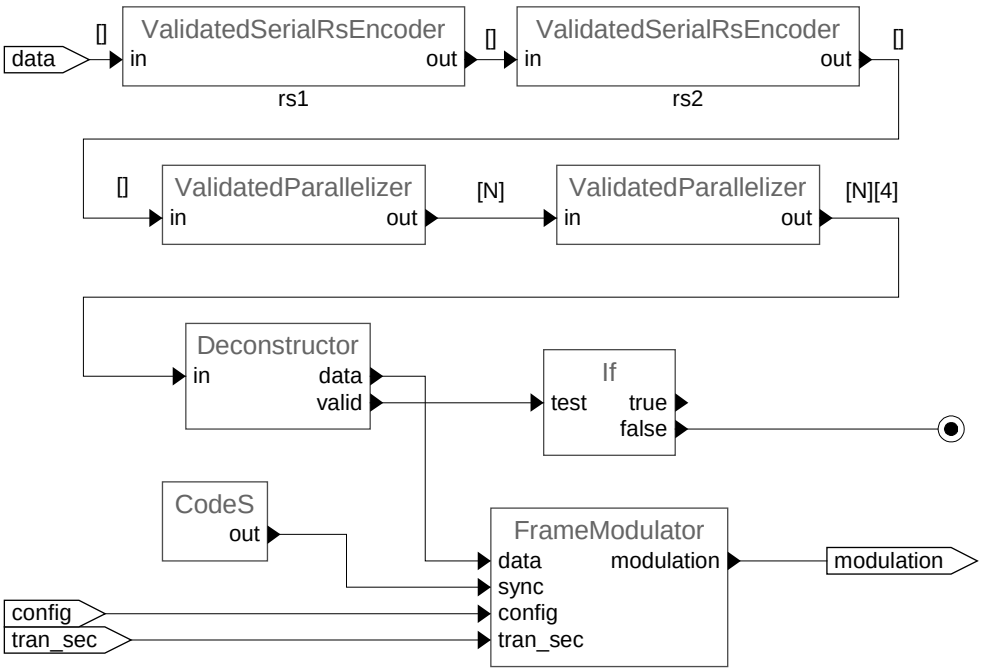
Configuration

```
constraint: iterations.length = 17;
constraint: frequencies.length = 64;
constraint: repeats.value = 16;
constraint: after.value = 2;
```

6.8 Fm3tr.Phl.Fsm.DataTxCall



The `DataTxCall` entity specifies the data encoding, frame interleaving and hop modulation.



The data coding involves 2 levels of Reed-Solomon encoding, followed by two levels of packing to create the four hop frames required by the `FrameModulator`.

Source data is accepted as a serial bit stream with an accompanying valid bit corresponding to the presence of PTT. The incoming PTT is assumed to exhibit only a single transition from valid to not valid during a transmission. The Ptt filtering entity enforces this assumption in the Phl layer.

Message termination occurs when there is no further data to send. The valid bit accompanies each incoming data bit to indicate whether the bit forms part of, or is beyond the end of, the message. The `ValidatedSerialRsEncoder` and `ValidatedParallelizer` maintain this information so that the validity indicator, extracted by the `Deconstructor`, signals valid so long as any source bit needs transmission, even if that involves just a single bit in an obscure interleaving position. Loss of validity generates an exit, requiring the parent state machine in `DataTxFsm` to advance and exit requiring the main layer `Fsm` to advance to its RX state.

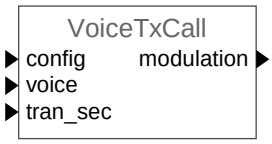
Ports

```
in:token data : ValidBit;
in:token tran_sec : TranSec;
in:value config : PhlConfig;
out:signal modulation : Modulation;
```

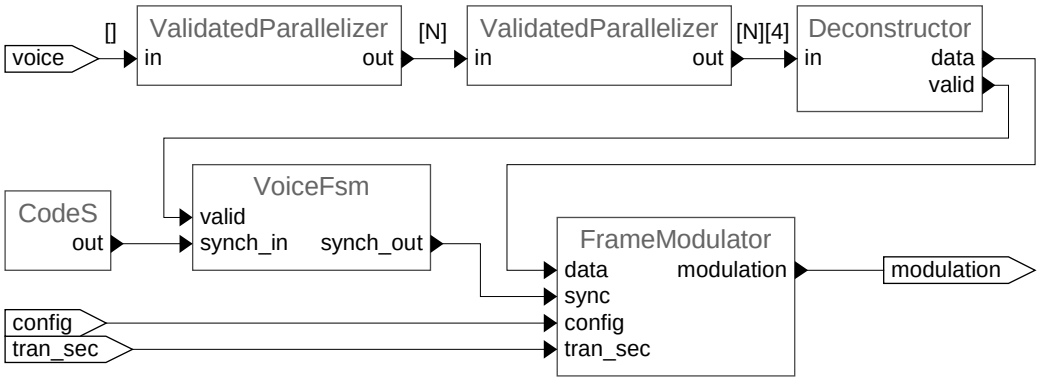
Configuration

```
constraint: rs1.params = config.rs1;
constraint: rs2.params = config.rs2;
```

6.9 Fm3tr.Phl.Fsm.VoiceTxCall



The `VoiceTxCall` entity specifies the data packing, frame interleaving, EOM sequencing and hop modulation.



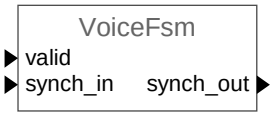
Voice processing differs from data processing in two ways. The Reed Solomon encoders are eliminated (and replaced by a data buffer), and End of Message frames are added.

End Of Message sequencing is specified by a state machine within the `SynchHopData` entity, assisted by validity information propagated through the data coder. A PTT bit accompanies each incoming voice bit to indicate whether the bit forms part of, or is beyond the end of, the message. The `ValidatedParallelizers` maintain this information so that the validity indicator, extracted by the `Deconstructor`, signals valid so long as any source bit needs transmission, even if that involves just a single bit in an obscure interleaving position. Loss of validity triggers the EOM sequencer to append two frames of inverted sync. After the EOM frames have been generated, the `SynchHopData` state machine exits, and consequently this message flow also exits causing the parent state machine in `VoiceTxFsm` to advance and exit causing the main layer `Fsm` to advance to its RX state.

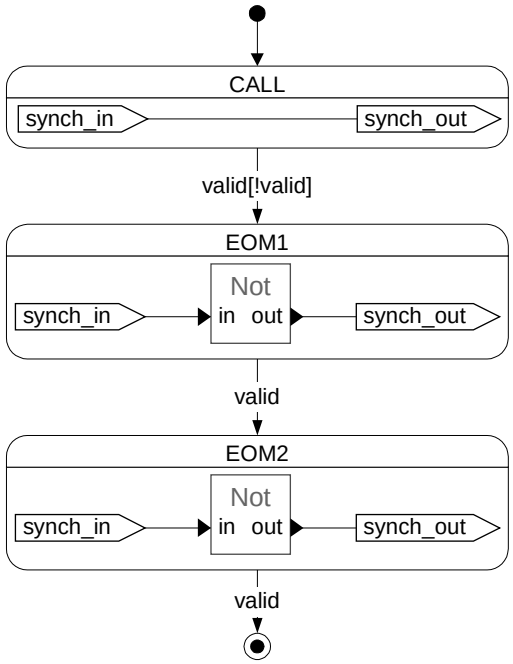
Ports

```
in:token voice : VoiceIn;
in:token tran_sec : TranSec;
in:value config : PhlConfig;
out:signal modulation : Modulation;
```

6.9.1 Fm3tr.Phl.VoiceTxCall.VoiceFsm



The `VoiceFsm` provides the true/complement synch data and sequences EOM frames at the end of transmission.



The CALL state is used for the bulk of a transmission. The state progresses in order to append two EOM frames at the end of normal transmission. The state machine exits upon completion of the two frames. This exit should cause the parent state machine(s) to exit terminating the transmission and returning to a receive mode.

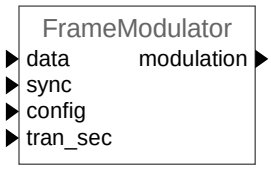
Note that `valid` as the transition event responds to the occurrence of the valid message, whereas `valid` as the transition guard makes use of the boolean state transferred as the `valid` message.

Within each state the true of complement of the synch input is fed to the synch output, using an embedded message flow that is sufficiently simple to express without introducing a further level of decomposition.

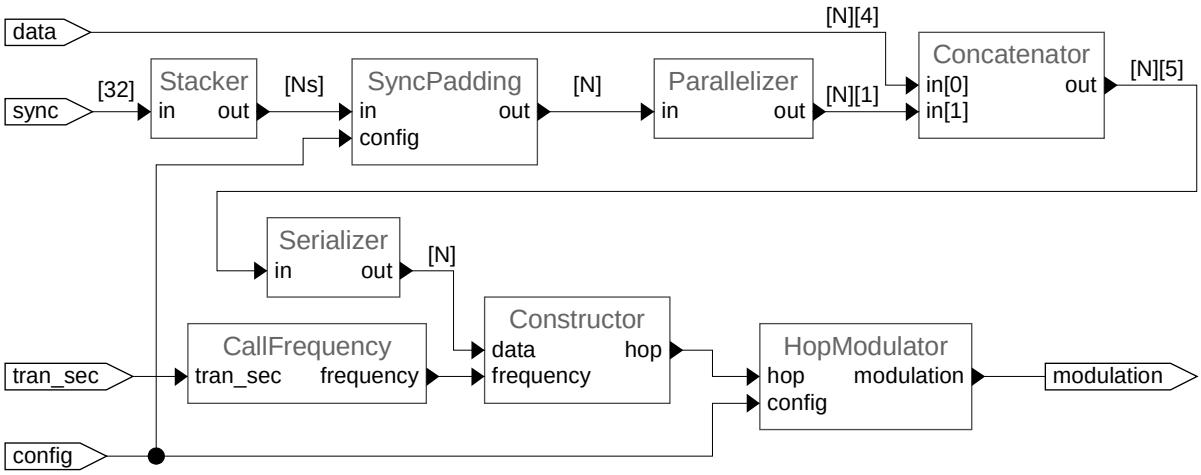
Ports

```
in:event valid : boolean;
in:token synch_in : SyncPacket;
out:token synch_out : SyncPacket;
```

6.10 Fm3tr.Phl.FrameModulator



The `FrameModulator` specifies the frame interleaving and subsequent modulation.



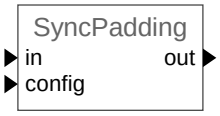
A hop sequence is derived from 4 data frames interleaved with 1 sync frame, which is formed by stacking an appropriate number of repeats of the 32 bit sync pattern and padding to fill the information requirements of a hop.

The conversion of information from the `tran_sec` input to the hop frequency is not defined by FM3TR.

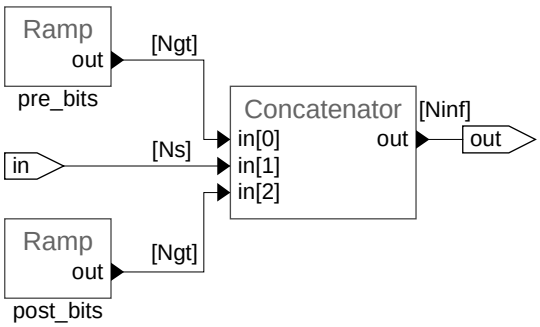
Ports

```
in:token data : boolean[config.ninf][4];
in:value sync : SyncPacket;
in:value config : PhlConfig;
in:value tran_sec : TranSec;
out:signal modulation : Modulation;
```

6.10.1 Fm3tr.Phl.FrameModulator.SyncPadding



The SyncPadding entity specifies the positioning of sync bits within the information bits of a hop.



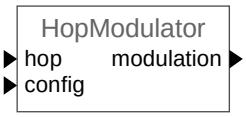
Ports

```
in:value config : PhlConfig;
in:token in : SyncPacket;
out:token out : boolean[config.ninf];
```

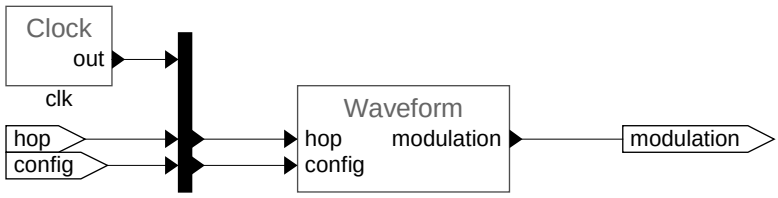
Configuration

```
constraint: pre_bits.length = config.ngt;
constraint: pre_bits.step = 0;
constraint: post_bits.length = config.ngt;
constraint: post_bits.step = 0;
```

6.11 Fm3tr.Phl.HopModulator



The `HopModulator` specifies real-time constraints for the conversion of a hop packet to a ‘continuous’ waveform..



The Clock requires synchronization of the `hop` (and `config`) flows to the real-time hop-rate.

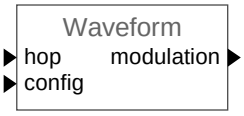
Ports

```
in:value config : PhlConfig;
in:token hop : HopPacket;
out:signal modulation : Modulation;
```

Configuration

```
constraint: clk.period = config.hop_period;
constraint: waveform._after = clk;
```

6.11.1 Fm3tr.Phl.HopModulator.Waveform

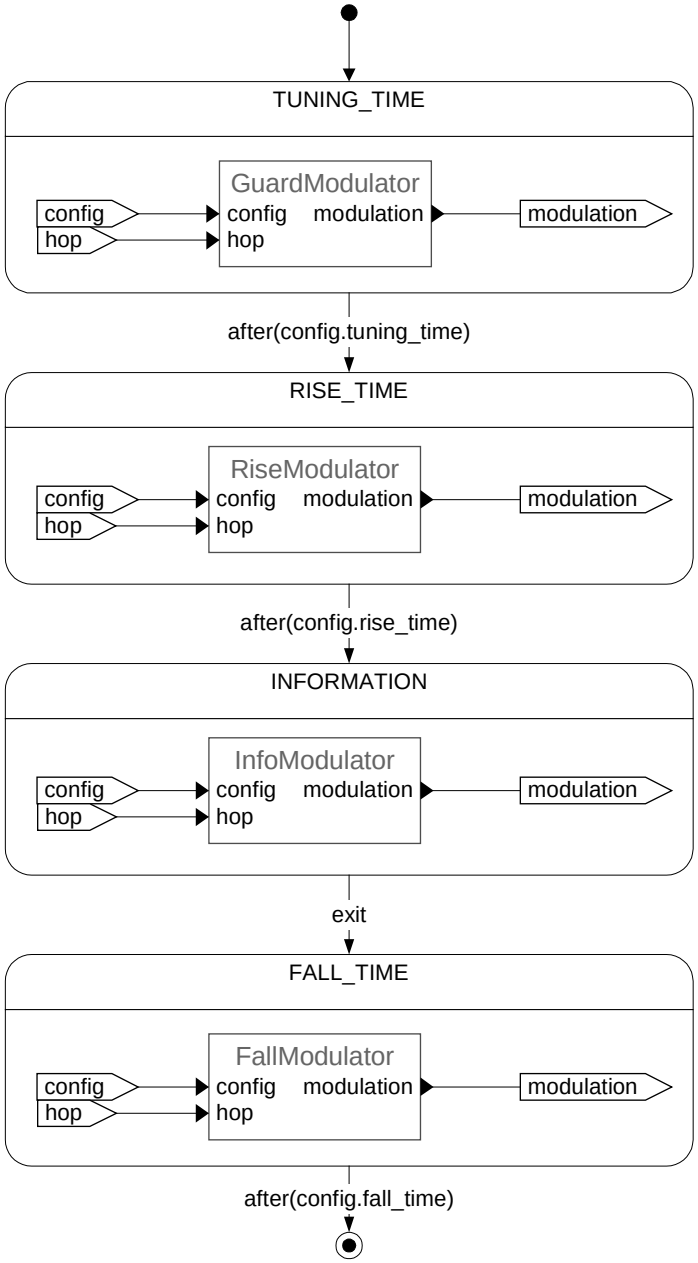


The Hop modulator creates the 'continuous' time waveform from the hop control packet. The modulated waveform comprises a tuning time, rise-time, information period and fall-time, each of which is generated during a corresponding state of the state machine specifying the modulator.

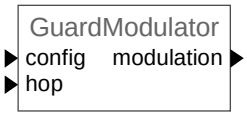
> The tuning-time appears at the end of the hop in the FM3TR spec. It may appear first or last. A practical implementation must tune before (rather than after) each hop, so writing the specification in terms of a preceding tuning time reduces the time skewing necessary in practice.

Ports

```
in:value config : PhlConfig;
in:event hop : HopPacket;
out:signal modulation : Modulation;
```


TUNING_TIME
GuardModulator
config
hop
config
hop
modulation
modulation
after(config.tuning_time)
RISE_TIME
RiseModulator
config
hop
config
hop
modulation
modulation
after(config.rise_time)
INFORMATION
InfoModulator
config
hop
config
hop
modulation
modulation
exit
FALL_TIME
FallModulator
config
hop
config
hop
modulation
modulation
after(config.fall_time)

6.11.2 Fm3Tr.Phl.HopModulator.GuardModulator

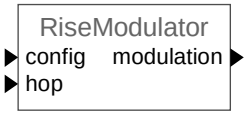


During the tuning time the amplitude is zero and no phase change occurs.

Ports

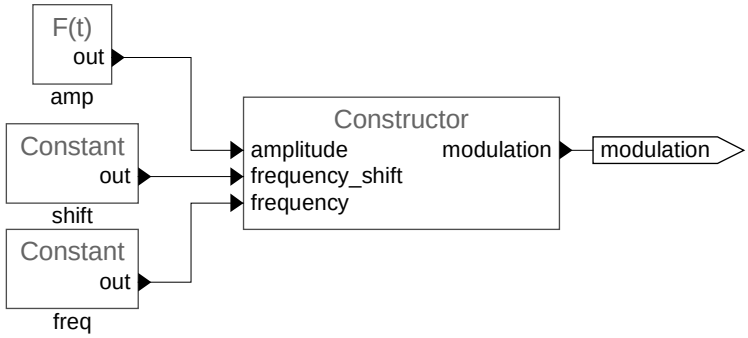
```
in:value config : PhlConfig;
in:value hop : HopPacket;
out:signal modulation = 0 : Modulation;
out:signal frequency_shift = 0 : FrequencyShift;
out:signal frequency = hop.frequency : FrequencyShift;
```

6.11.3 Fm3Tr.Phl.HopModulator.RiseModulator



During the rise time the amplitude rises from zero to full amplitude, and no phase change occurs.

The FM3TR specification provides no guidance upon the rise shape or frequency offset. It should presumably satisfy certain spectral leakage properties. The exposition here arbitrarily chooses a raised cosine shape and provides an implementation tolerance.



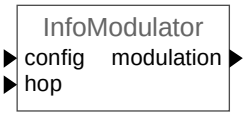
Ports

```
in:value config : PhlConfig;
in:value hop : HopPacket;
out:signal modulation : Modulation;
```

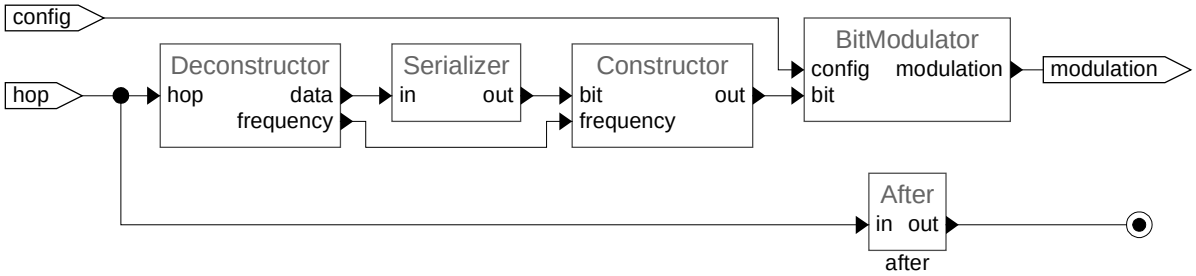
Configuration

```
constraint: amp.out = range
    {
        minimum 0;
        value 0.5 * (1 - cos(2*pi*t/config.rise_time));
        maximum 1;
    };
constraint: shift.out =0;
constraint: freq.out = hop.frequency;
```

6.11.4 Fm3Tr.Phl.HopModulator.InfoModulator



During the information time the hop data is serialized and modulated by the `BitModulator`.



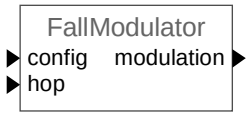
Ports

```
in:value config : PhlConfig;
in:value hop : HopPacket;
out:signal modulation : Modulation;
```

Configuration

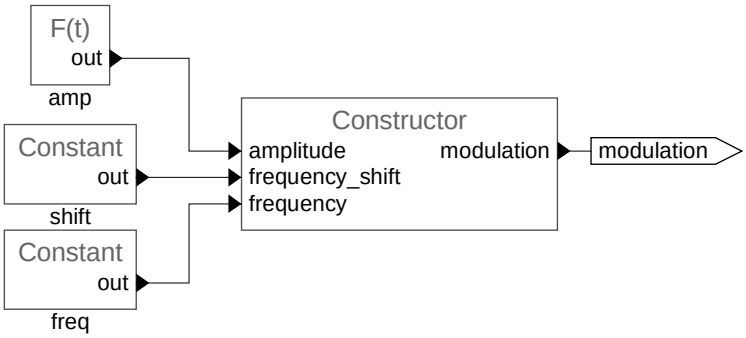
```
constraint: after.value = 2;
```

6.11.5 Fm3Tr.Phl.HopModulator.FallModulator



During the fall time the amplitude falls from full amplitude to zero, and no phase change occurs.

The FM3TR specification provides no guidance upon the fall shape or frequency offset. It should presumably satisfy certain spectral leakage properties. The exposition here arbitrarily chooses a raised cosine shape and provides an implementation tolerance.



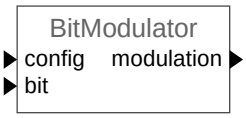
Ports

```
in:value config : PhlConfig;
in:value hop : HopPacket;
out:signal modulation : Modulation;
```

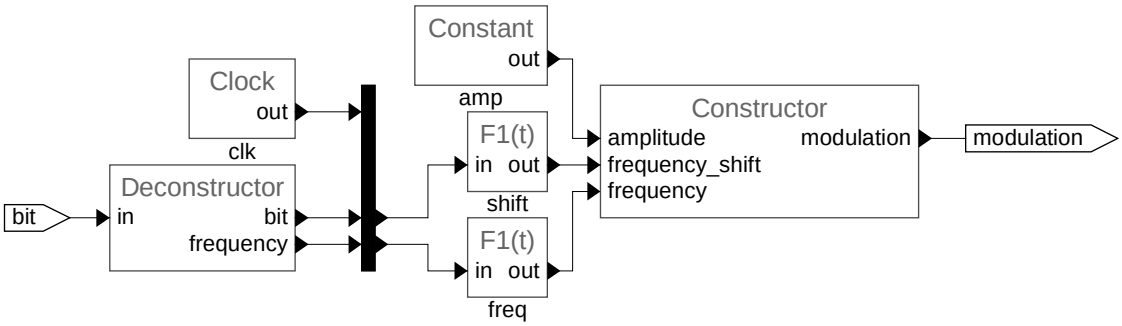
Configuration

```
constraint: amp.out = range
    {
        minimum 0;
        value 0.5 * (1 + cos(2*pi*t/config.rise_time));
        maximum 1;
    };
constraint: shift.out =0;
constraint: freq.out = hop.frequency;
```

6.12 Fm3tr.Phl.BitModulator



The `BitModulator` specifies the frequency shift associated with a boolean data bit.



Although this specification defines the `frequency_shift` as a zero IF style modulation, implementations may use a non-zero IF by transferring part of `frequency` to `frequency_shift`.

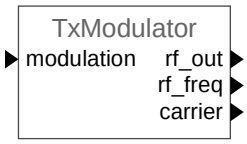
Ports

```
in:token bit : BitPacket;
out:signal modulation : Modulation;
```

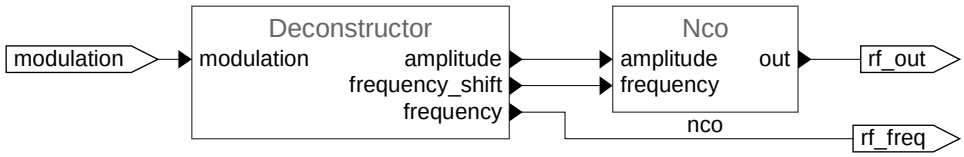
Constraint

```
constraint: clk.period = config.bit_period;
let bit_value = 1 - 2 * shift.in;
let freq_shift = 0.5 * config.bit_rate * config.mod_index;
constraint: amp.out = 1;
constraint: shift.out = bit_value * freq_shift;
constraint: freq.out = freq.in;
```

6.13 Fm3tr.Phl.TxModulator



The `TxModulator` combines the modulation fields to produce fine and coarse-grained signals to control the modulator in a `Radio`.



Although this specification treats the `frequency_shift` as a zero IF style modulation, implementations may use a non-zero IF by transferring part of `frequency` to `frequency_shift`.

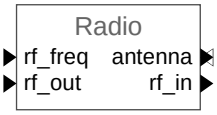
Ports

```
in:signal modulation : Modulation;
out:signal rf_out : TxSignal;
out:signal rf_freq : Frequency;
out:signal carrier = false : CarrierDetect;
```

Configuration

```
constraint nco.frequency_stability < 1.0e-6;
```

6.14 Fm3tr.Phl.Radio

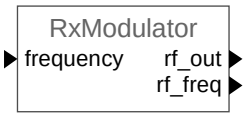


The Radio entity should specify the analogue parts of a radio, without inhibiting any particular implementation approach. Since there are many valid approaches, none is specified, and since FM3TR provides no analogue behaviour specifications it is not even possible to specify the signal qualities required on the signal and frequency signals.

Ports

```
in:signal rf_out : TxSignal;
in:signal rf_freq : Frequency;
out:signal rf_in : RxSignal;
inout:signal antenna : Antenna;
```

6.15 Fm3tr.Phl.RxModulator



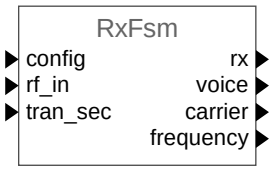
The `RxModulator` specifies the receive frequency and the spurious emissions during receive operation.

FM3TR provides no specification for spurious emissions.

Ports

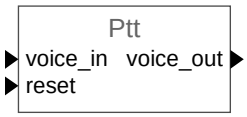
```
in:signal frequency : Frequency;
out:signal rf_out  = 0 : RxSignal;
out:signal rf_freq = frequency : Frequency;
```

6.16 Fm3tr.Phl.DataRxFsm

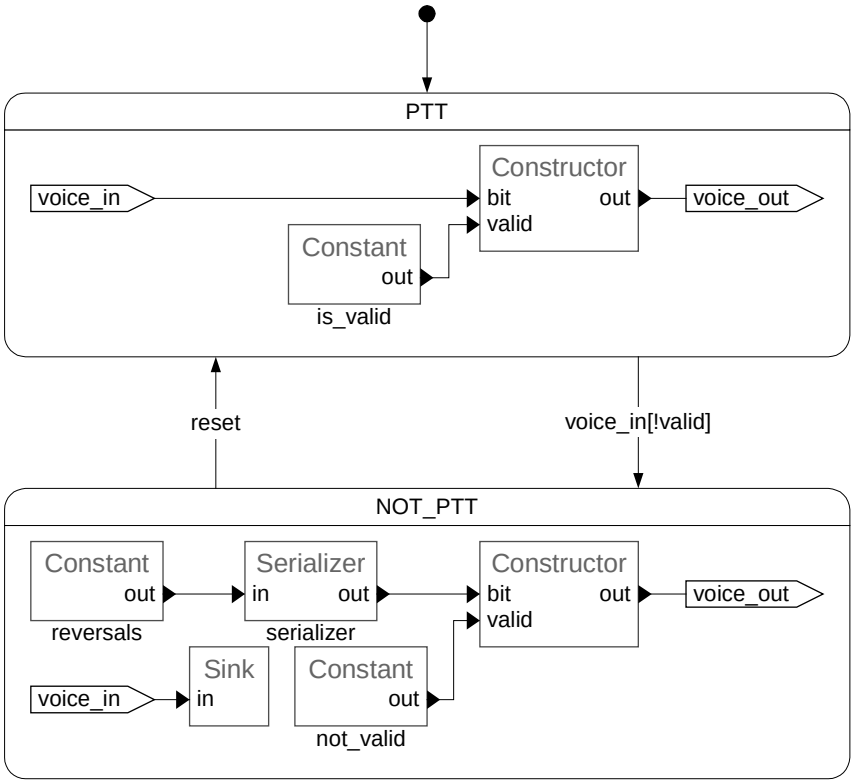


The receive processing is not specified by FM3TR. However some form of specification should be provided in WDL to define a reference implementation.

6.17 Fm3Tr.Phl.Ptt



The integrity of the PTT (push-to-talk) signal is not addressed by the FM3TR specification. The specification of the coder presented in VoiceTxCall assumes a clean signal and so a monostable clean-up is defined here to satisfy that requirement.



While in the PTT state voice samples are passed through with an accompanying valid bit. Once PTT is de-asserted, input samples are to be discarded, and output samples must indicate invalid.

The FM3TR specification does not specify the padding bits to follow the end of transmission. The specification here suggests padding with reversals since this is the idle pattern for CVSD.

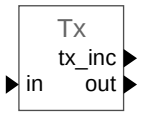
Ports

```
in:token voice_in : VoiceIn;
in:event reset : void;
out:token voice_out : VoiceIn;
```

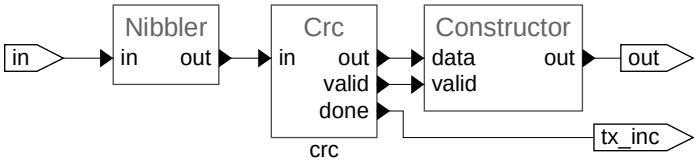
Configuration

```
constraint: is_valid.value = true;
constraint: not_valid.value = false;
constraint: reversals.value = [false,true];
```

6.18 Fm3tr.Phl.Tx



The Tx entity specifies the conversion of the data packet formatted by the Dlc layer and passed through the Mac layer into a Phl data packet.



A CRC is appended to the source data and data validity applied to each bit so that the padding bits necessary to fill Reed Solomon blocks can be distinguished. An increment message is generated on completion to maintain the statistics counter.

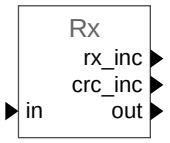
Ports

```
in:token in : MacPacket;
out:token out : ValidBit;
out:token tx_inc : void;
```

Configuration

```
constraint crc.polynomial
           = Reverser([1, 0,0,0,1, 0,0,0,0, 0,0,1,0, 0,0,0,1]);
```

6.19 Fm3tr.Phl.Rx



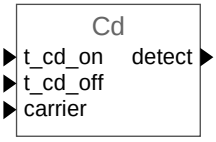
The Rx entity specifies checking and removal of the CRC from received data packets.

This entity has not yet been decomposed. It would be relatively straightforward given a byte-wide CRC block. The bit-wide block used for Tx is not very helpful.

Ports

```
in:token in : Octet[*];
out:token out : MacPacket;
out:token crc_inc : void;
out:token rx_inc : void;
```

6.20 Fm3tr.Phl.Cd

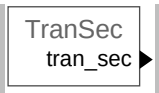


The behaviour of the carrier detect filtering in FM3TR is not specified beyond defining the names of the $PHL_{t\ CD\ on}$ and $PHL_{t\ CD\ off}$ control variables. It is not clear whether these refer to exponential attack and decay times, or resettable time-out intervals.

Ports

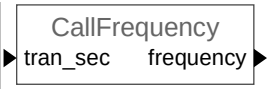
```
in:token carrier : CarrierDetect;
in:value t_cd_on : CdTime;
in:value t_cd_off : CdTime;
out:value detect : CarrierDetect;
```

6.21 Fm3tr.Phl.TranSec



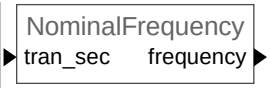
The behaviour of the TranSec is not specified by FM3TR, indeed the requirement for one is largely overlooked. The lack of specification is highlighted here by identifying where the TranSec might contribute.

6.21.1 Fm3tr.Phl.TranSec.CallFrequency



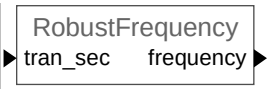
While a call is in progress, the `CallFrequency` entity should identify the frequency to be used for a hop.

6.21.2 Fm3tr.Phl.TranSec.NominalFrequency



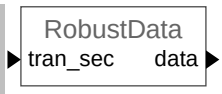
At the start of a call using a nominal pre-amble, the NominalFrequency entity should identify the frequency to be used for the preamble.

6.21.3 Fm3tr.Phl.TranSec.RobustFrequency



At the start of a call using a robust pre-amble, the RobustFrequency entity should identify the frequencies to be used for the preamble.

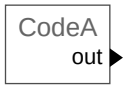
6.21.4 Fm3tr.Phl.TranSec.RobustData



At the start of a call using a robust pre-amble, the RobustData entity should identify the synchronisation pattern to be used for the preamble.

6.22 Fm3tr.Phl Data Values

6.22.1 Fm3tr.Phl.CodeA

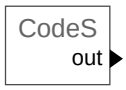


The CodeA bit pattern is used at the start of a nominal call.

Ports

```
out:constant = [0,0,0,0, 0,0,0,1, 0,1,0,1, 1,0,1,0,
                1,1,0,0, 1,1,1,1, 1,0,0,1, 1,1,1,0] : SyncPacket;
```

6.22.2 Fm3tr.Phl.CodeS

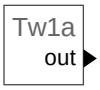


The CodeS bit pattern is used for synch patterns.

Ports

```
out:constant = [0,1,1,0, 1,1,0,1, 1,1,1,0, 1,0,1,0,
                0,1,1,1, 0,1,1,1, 0,0,0,1, 1,1,1,0] : SyncPacket;
```

6.22.3 Fm3tr.Phl.Tw1a



The Tw1a entity specifies the configuration parameters for TW#1a.

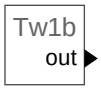
Ports

```
out:constant : Tw;
```

Configuration

```
constraint: out.rs1.Symbol.polynomial = Reverser([1,0,0,1,0,1]);
constraint: out.rs1.user_symbols = 14;
constraint: out.rs1.parity_symbols = 16-14;
constraint: out.rs2.Symbol.polynomial
                      = Reverser([1,0,0,0,1,0,0,1]);
constraint: out.rs2.user_symbols = 72;
constraint: out.rs2.parity_symbols = 105-72;
constraint: out.code_word1 = 0;
constraint: out.code_word2 = 0;
constraint: out.bit_rate = 25‘kHz;
constraint: out.bit_period = 1 / out.bit_rate;
constraint: out.ns = 64;
constraint: out.ngt = 8;
constraint: out.ninf = 80;
constraint: out.hop_period = 100 * out.bit_period;
constraint: out.rise_time = 5 * out.bit_period;
constraint: out.fall_time = 5 * out.bit_period;
constraint: out.tuning_time = 10 * out.bit_period;
constraint: out.mod_index = 0.5;
```

6.22.4 Fm3tr.Phl.Tw1b



The Tw1b entity specifies the configuration parameters for TW#1b.

Ports

```
out:constant : Tw;
```

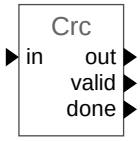
Configuration

```
constraint: out.rs1.Symbol.polynomial = Reverser([1,0,0,1,1]);
constraint: out.rs1.user_symbols = 8;
constraint: out.rs1.parity_symbols = 10-8;
constraint: out.rs2.Symbol.polynomial
                     = Reverser([1,0,0,0,1,1,1,0,1]);
constraint: out.rs2.user_symbols = 189;
constraint: out.rs2.parity_symbols = 252-189;
constraint: out.code_word1 = 1;
constraint: out.code_word2 = 0;
constraint: out.bit_rate = 25‘kHz;
constraint: out.bit_period = 1 / out.bit_rate;
constraint: out.ns = 32;
constraint: out.ngt = 4;
constraint: out.ninf = 40;
constraint: out.hop_period = 50 * out.bit_period;
constraint: out.rise_time = 2.5 * out.bit_period;
constraint: out.fall_time = 2.5 * out.bit_period;
constraint: out.tuning_time = 5 * out.bit_period;
constraint: out.mod_index = 0.5;
```

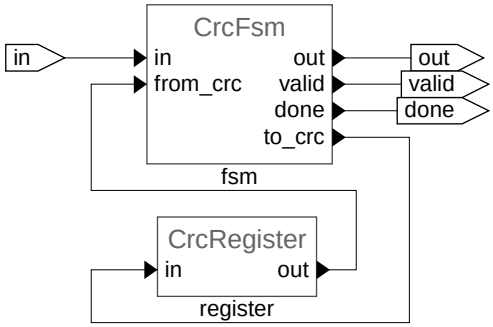
7 Library support

Descriptions of some entities that should become general purpose library contributions are provided in this section.

7.1 Crc



The CRC entity comprises a state machine to sequence CRC generation, emission and optional padding, and a shift register that is persistent across the states.



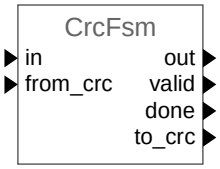
Ports

```
in:value crc:Polynomial;
in:token in : boolean[*];
out:token out boolean;
out:token valid:boolean;
out:event done:void;
```

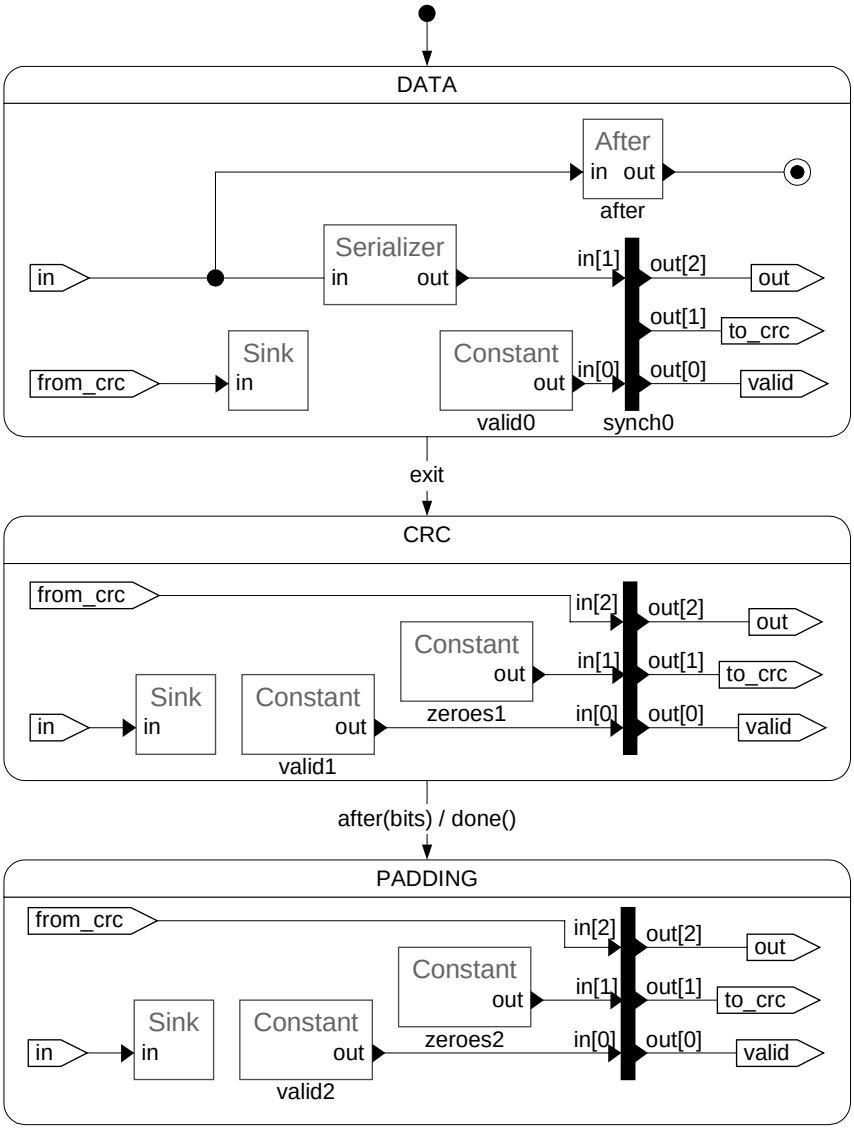
Configuration

```
constraint: fsm.bits = crc.order;
constraint: register.crc = crc;
```

7.1.1 CrcFsm



The CrcFsm entity specifies the sequencing through the sucessive phases of CRC generation. The input data is first passed through to the output, while calculating the CRC and signalling that the data is valid. Then the CRC is passed to the output while again signalling that data is valid, and on completion of the CRC emission generating a done output. Any further output is generated by the padding state that appends zeroes while signalling invalid.



Ports

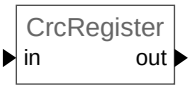
```
in:value bits:natural;
in:token in : boolean[*];
```

```
in:token from_crc : boolean;
out:token out boolean;
out:token valid:boolean;
out:event done:void;
out:token to_crc:boolean;
```

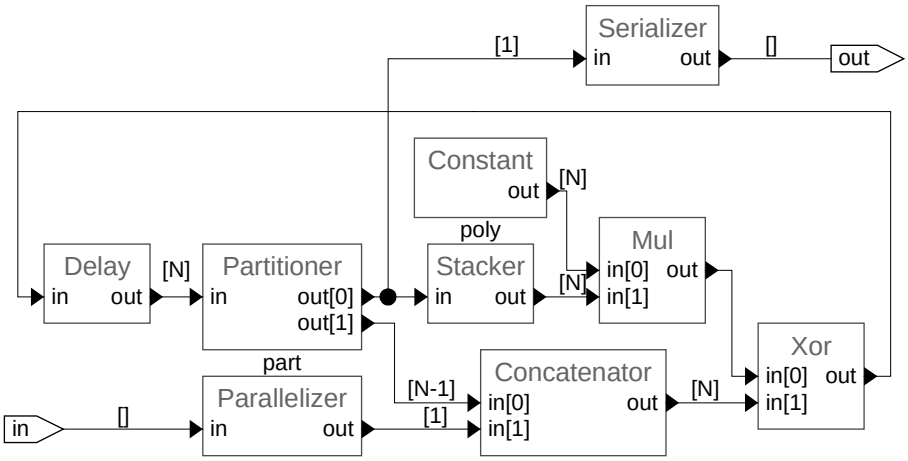
Configuration

```
constraint: valid0.value = true;
constraint: valid1.value = true;
constraint: valid2.value = false;
constraint: zeroes1.value = 0;
constraint: zeroes2.value = 0;
constraint: synch0.out[2] = synch0.in[1];
constraint: after.value = in._shape[0];
```

7.1.2 CrcRegister



The `CrcRegister` entity specifies the accumulation of a CRC bit.



This is a very tentative specification: the polynomial is probably back to front. It may be more appropriate to use a more abstract polynomial division.

In practice a much more efficient algorithm can be implemented that operates byte-wide.

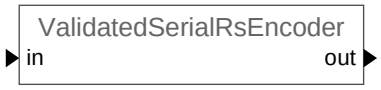
Ports

```
in:value polynomial : Polynomial;
in:token in : boolean;
out:token out boolean;
```

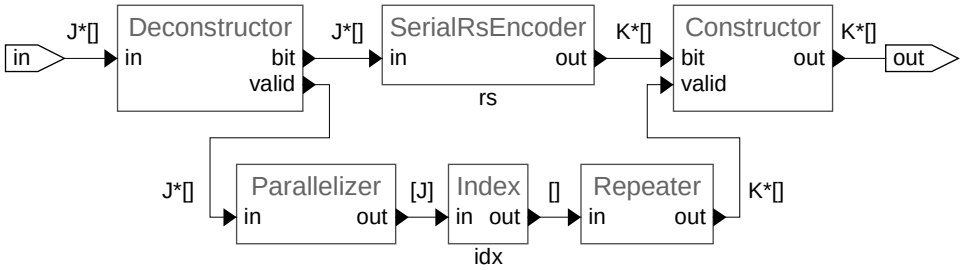
Configuration

```
constraint: polynomial._shape[0] > 0;
let n = polynomial._shape[0]-1;
constraint: poly.value = Index(polynomial, Ramp(0,n));
constraint: part.out[1]._shape[0] = n-1;
```

7.2 ValidatedSerialRsEncoder



The ValidatedSerialRsEncoder specifies a Reed-Solomon encoder with serial interfaces, that maintains a validity for each outgoing bit according to the validity of the first bit in an encoded frame.



The annotation `J*[]` indicates a sequence of J scalar values.

The `Deconstructor` and `Constructor` remove and add the data validity indication. The side path uses a `Parallelizer` to expose the vector of validity bits for the RS frame and then picks the first (0th) element and repeats it to create the temporal vector of output validity bits. A more costly but general purpose implementation could perform a multi-input `Or` rather than assuming that the first describes all.

Ports

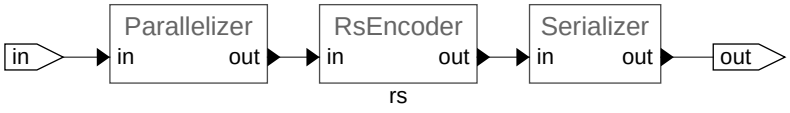
```
in:value params : RsParams;
in:token in : boolean;
out:token out : boolean;
```

Configuration

```
constraint: rs.params = params;
constraint: idx.indexes = 0;
constraint: idx.in._shape[0] = params.user_symbols;
```

7.3 SerialRsEncoder

The `SerialRsEncoder` specifies a Reed Solomon encoder with serial interfaces.



Ports

```
in:value params : RsParams;
in:token in : boolean;
out:token out : boolean;
```

Configuration

```
constraint: rs.params = params;
```

Index