CAN Bus

Design for SummerSchool 2

Pascal Sartoretti

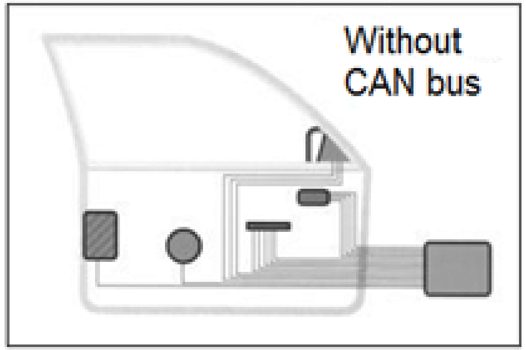
What’s CAN?

- CAN: Controller Area Network
  - Bus for a network of (micro-)controllers
  - Originally developed by R. BOSCH for use in automobiles at the end of the 80’s
  - Today, ISO (International Standard Organization) standard : ISO 11898
- Cheap, robust, medium speed, (hard) real-time bus to interconnect systems in a car
  - Doors, heating and cooling system, infra-red receiver, motor management…
  - Also used in industrial and building automation, medical equipment, robotics…

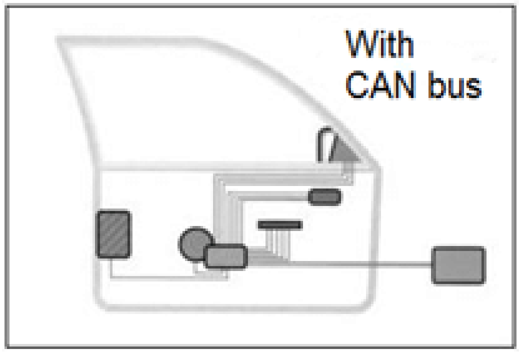
CAN and the automobile

- Example: A car door with / and without CAN

Without CAN



With CAN

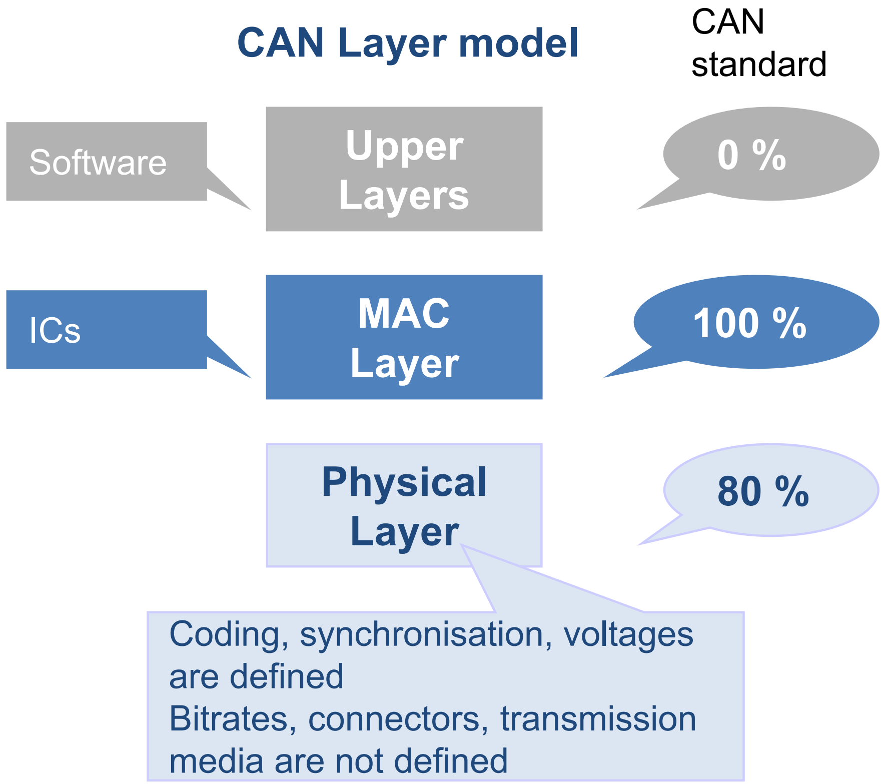


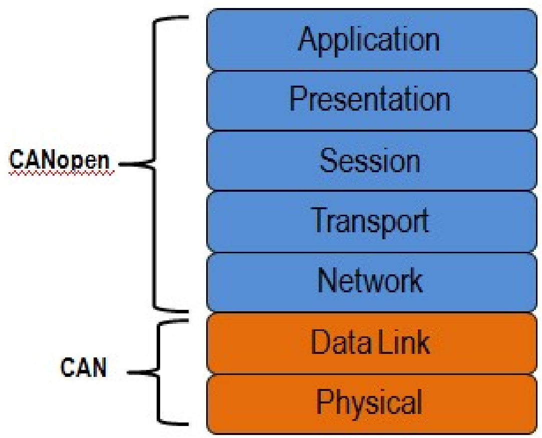
The automobile: the problem

- Too heavy, too big
  - Before CAN, high-end cars had several km of wires
- Too complicated to build
- Too difficult to test
  - No centralised data logging and diagnostic capability
- Not interoperable enough
  - Close all windows when locking the doors
  - Stop air conditioning when the sliding roof is open

The CAN bus standard

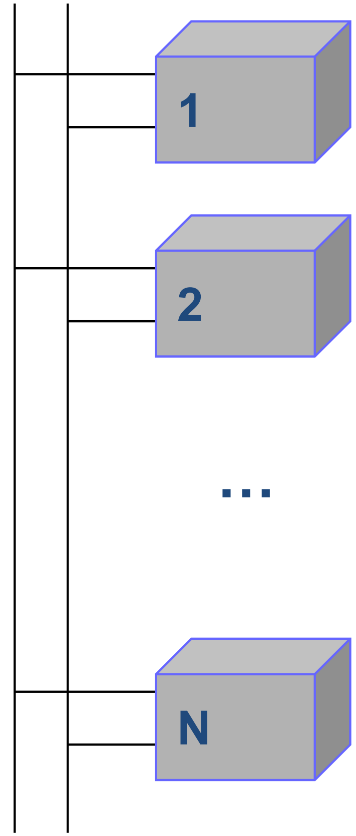
- Proprietary systems
  - All nodes are designed under a unique responsibility
  - Opposite of an open system
- CAN is a basic block used for proprietary applications
- Why standardizing proprietary applications?
  - Promote the development of compatible integrated circuits (ICs)
  - Standard is far less complete than standards for open busses
- CANopen (www.can-cia.org) is an open specification defining a CAN based protocol stack for automation





The architecture

- Two-wire & half-duplex
  - A single electrical circuit for all data transfers
- Master less
  - From a CAN point of view, all nodes are equivalent
- Single segment bus
  - Gateway function not included in the standard



The typologies

- High-speed CAN network. ISO 11898-2 (2016)
  - Up to 1Mb/sec
- Low-speed CAN signaling. ISO 11898-3 (2006)
  - Up to 125kB/sec
- [2012] CAN FD ISO 11898-1 (2012)
  (Controller Area Network **F**lexible **D**ata-Rate)
  - Used in modern high-performance systems
  - Nodes can dynamically switch to different data-rate and with larger or smaller message sizes
  - Data 8 bytes (Low/High-speed CAN) Vs. 64 bytes (FD)

The length, speed, payload

- The maximum length of cables is
  - 500 meters (@ 125kbit/s)
  - 40 meters (@ 1Mbit/sec)
- The baud rate* is
  - Up to 125 kbit/s for low speed CAN
  - Up to 1 Mbit/s for classical CAN
  - Up to 5-8 Mbit/s for CAN FD (Flexible Data-rate)
- The payload is
  - (low/high speed CAN) 11 or 29 bits for identifier and 0 to 8 bytes for data
  - (CAN FD) 11 or 12 or 29 bits for identifier, 0 to 64 bytes

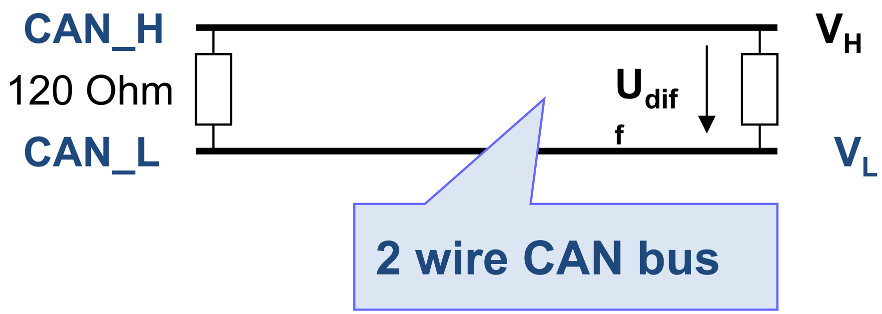
Approx. Speed (kbit/s)	Length (m)
1000	30
800	50
500	100
250	250
125	500
62.5	1000
20	2500
10	5000

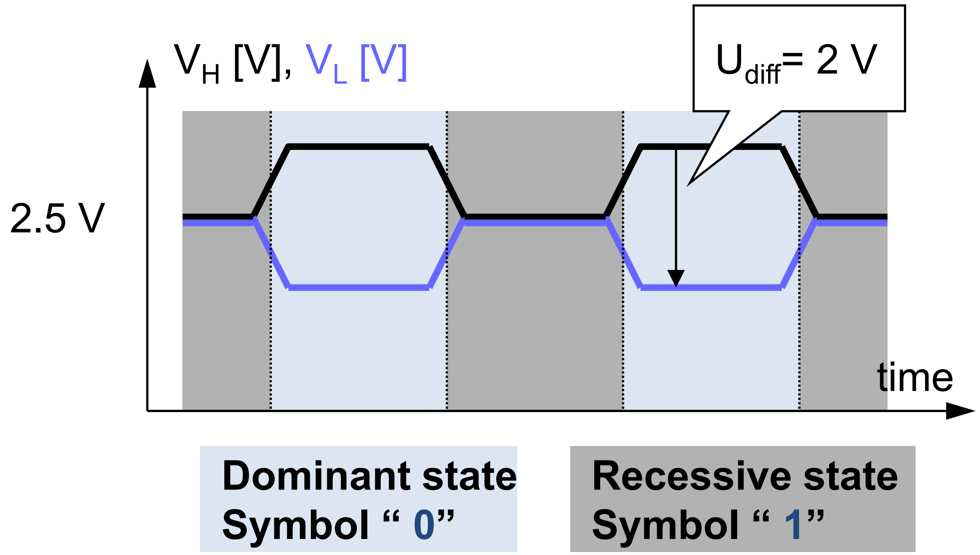
Source: https://fr.wikipedia.org/wiki/Bus_de_donn%C3%A9es_CAN#Support

* *Data rate depends on the topology for the bus network and the used Transceivers.*

Bus and physical encoding

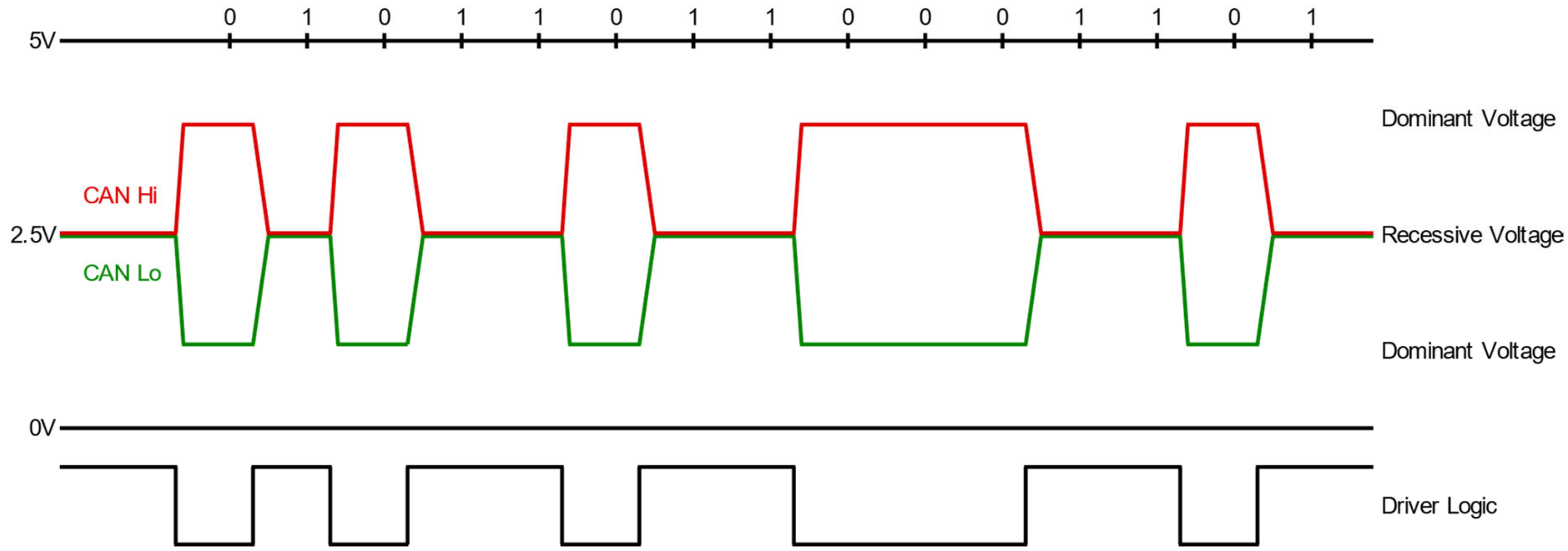
- NRZ (Non-Return to Zero) encoding
  - Symbol "**1**": recessive state
  - Symbol "**0**": dominant state
- 120 Ohm termination
  - Avoid reflection
    - ~ characteristic impedance of the line
- DC voltage: 2.5 V
- Differential data signal
  - +/- 1 V
- Nodes don't have to share the same ground signal



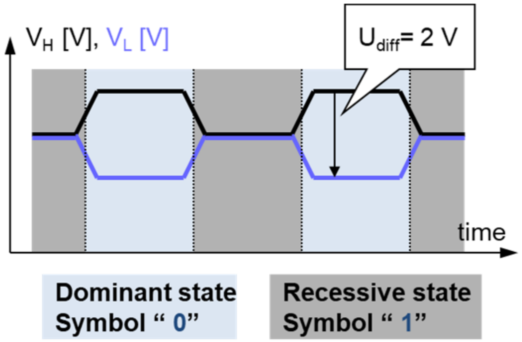


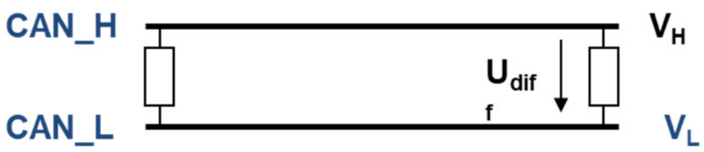
Bus and physical encoding

Physical layer



Source: https://en.wikipedia.org/wiki/CAN_bus#Physical_organization





Logic state	$V_{CANH-GND}$	$V_{CANL-GND}$	$V_{CANH-CANL}$
Recessive or « 1 »	2.5 V	2.5 V	between 0 - 0.5 V
Dominant or « 0 »	3.5 V	1.5 V	Between 0.9 - 2 V

About synchronisation

- Synchronous or asynchronous?
  - Asynchronous: periodic start & stop symbols to resynchronize the independent Tx and Rx clocks
  - Synchronous: receiver synchronizes its clock on sender's clock
- CAN is synchronous
  - All nodes have the same nominal bit rate
    - Real clock frequencies are different
    - The transmitter's clock control the Transmit Bit Rate **TBR**
  - A DPLL circuit adapts the receiver's clock to the transmitter's clock
    - DPLL: Digital Phase Lock Loop

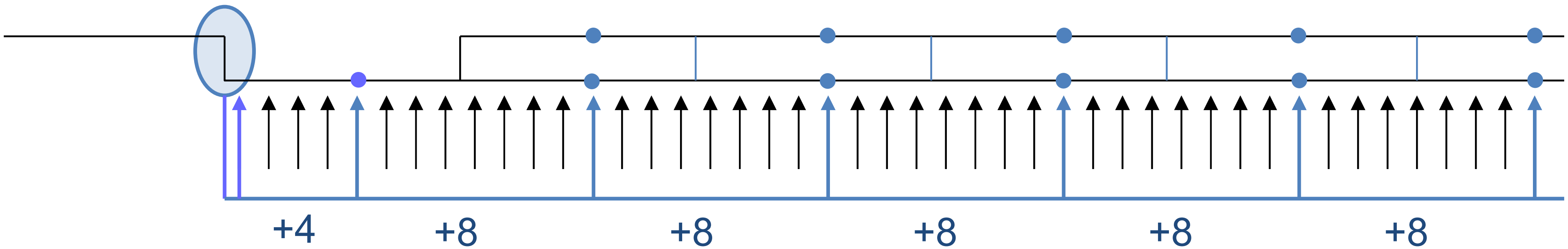
Synchronisation: DPLL

Physical layer

- The receiver oversamples the incoming signal
  - 8 to 25 samples per symbol
    - 8 samples/symbol in the example
  - The receiver's clock controls the Receiver Oversampling Frequency ROF

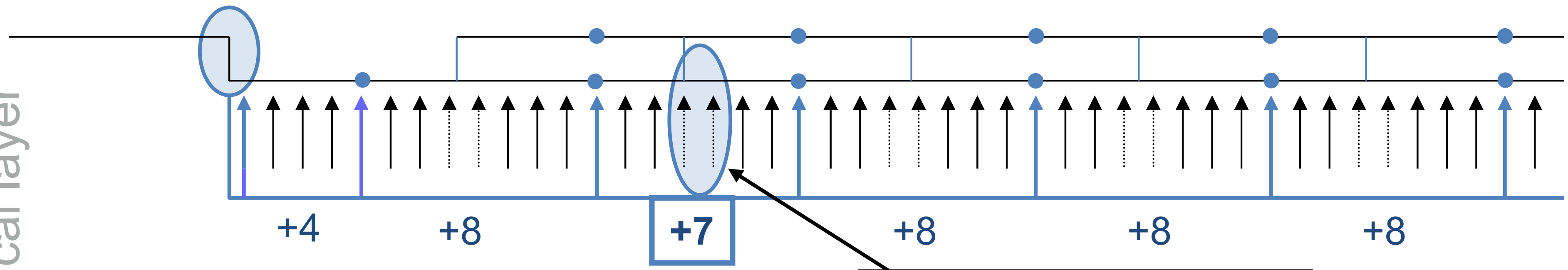
- The ideal case:

  Receiver's clock frequency = Transmitter's clock frequency



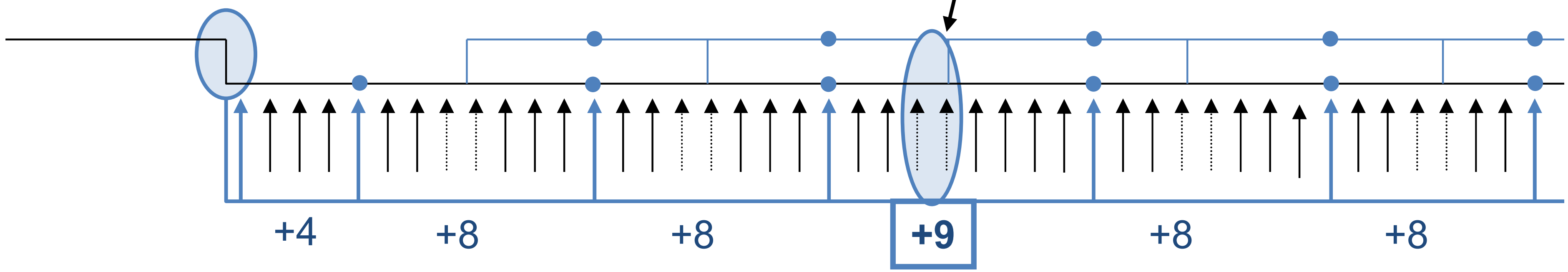
Synchronisation: DPLL

- The receiver is too slow
  - Receiver's clock frequency < Transmitter's clock frequency



Resynchronisation

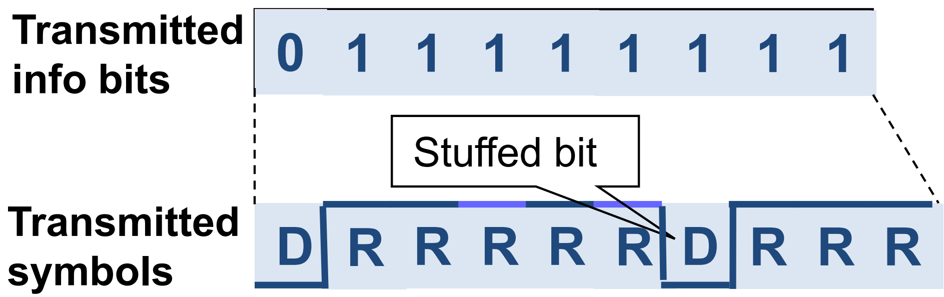
- The receiver is too fast
  - Receiver's clock frequency > Transmitter's clock frequency



Physical layer

Synchronisation: Bit stuffing (classical)

- To decide whether resynchronisation should occur, a receiver checks if a state transition occurs too early or too late:
  - State transition: from dominant to recessive state, or from recessive to dominant state
- This check can only be performed if there is a transition!
  - A minimal frequency of transitions is required
  - Bit stuffing imposes a minimal frequency of transitions

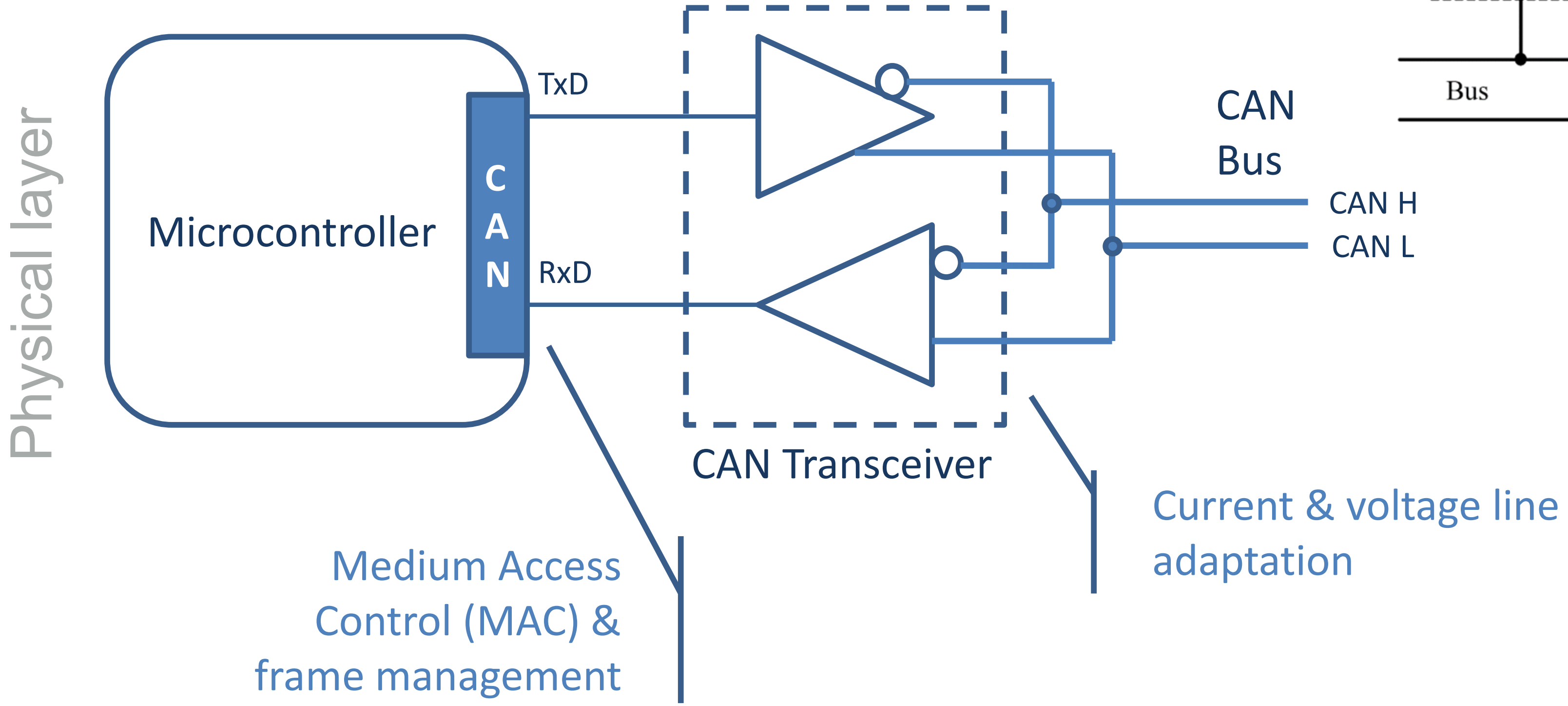


Bit stuffing:

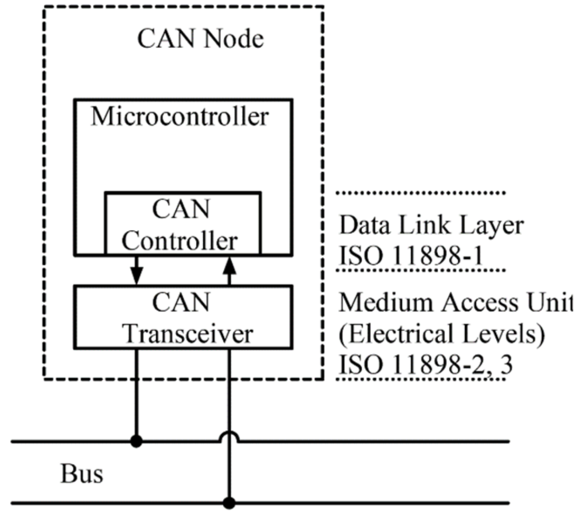
@ Transmitter: CAN hardware add an artificial dominant (recessive) symbol (called "stuffed bit") after 5 recessive (dominant) symbols

@ Receiver: CAN hardware removes stuffed bits (i.e. dominant symbols following 5 recessive symbols and recessive symbols following 5 dominant symbols

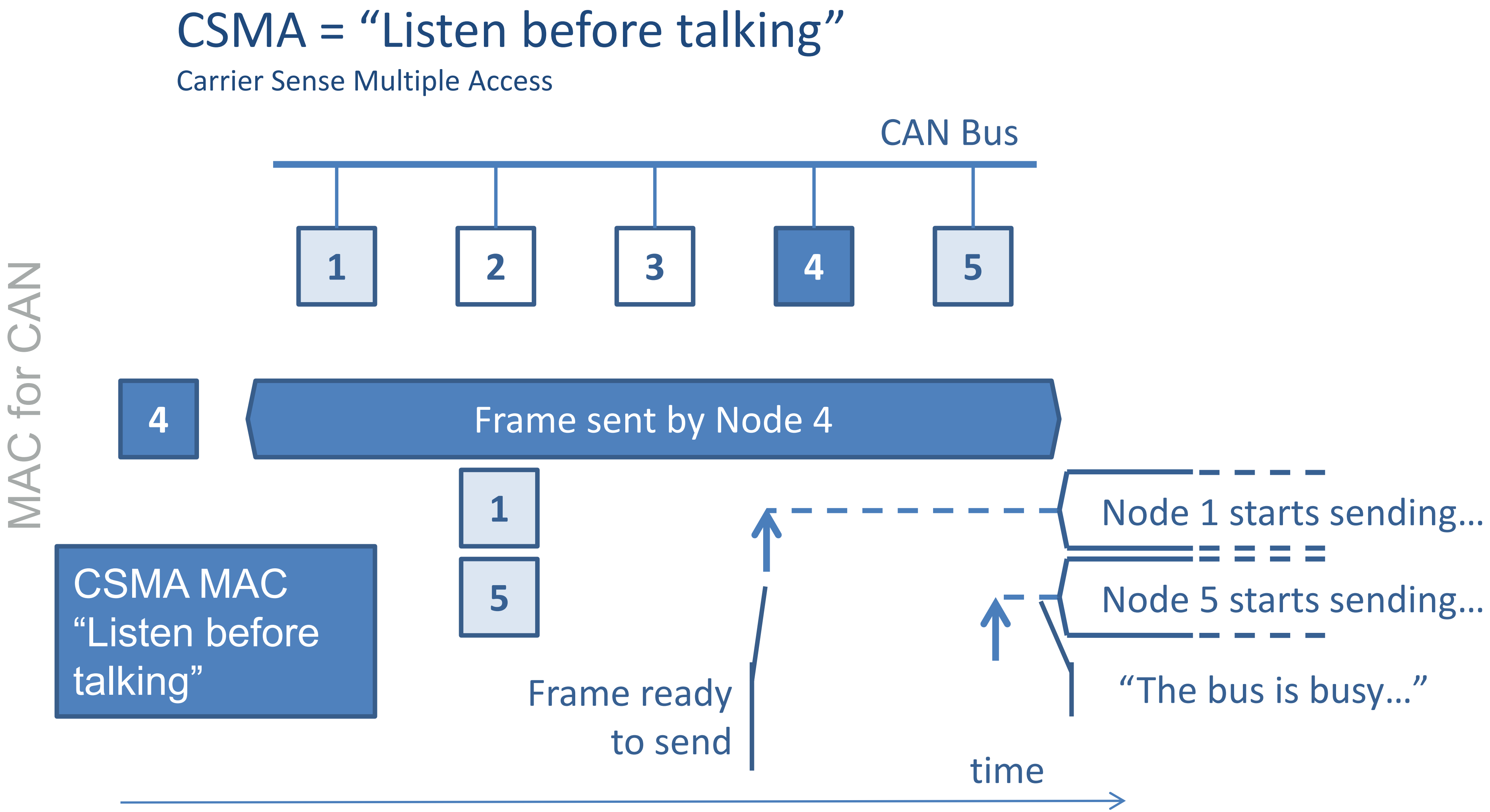
CAN & microcontrollers



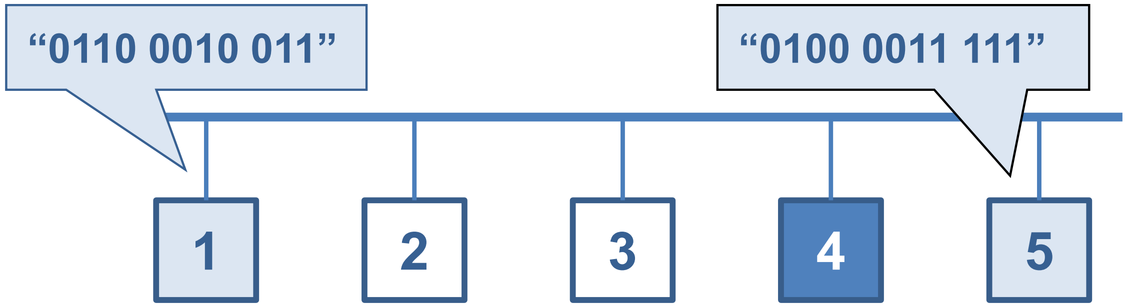
Source: https://en.wikipedia.org/wiki/CAN_bus#Nodes



CSMA based MAC for the CAN bus



CAN bus arbitration: Example



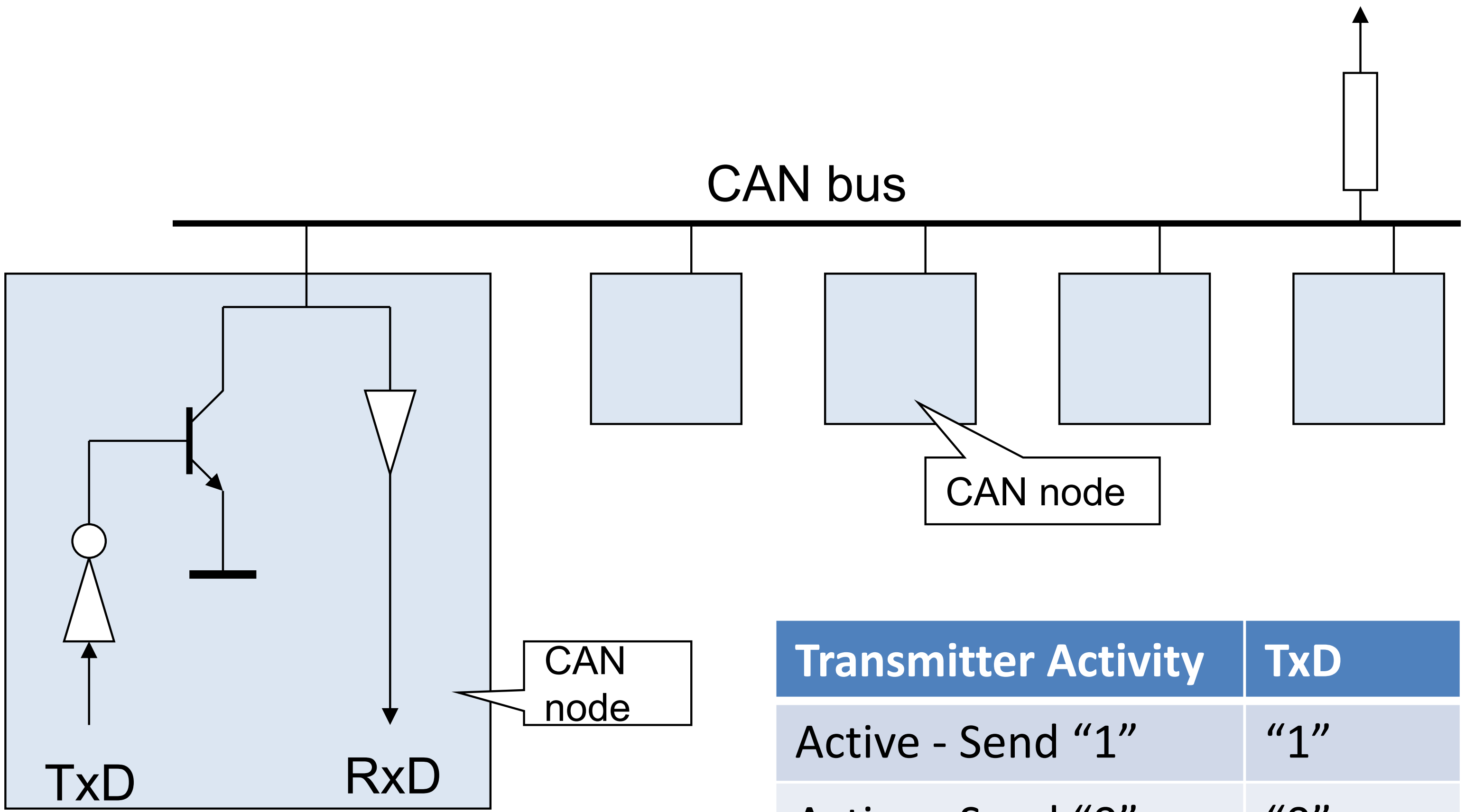
Dominant wins over **recessive**:
Node 1 looses arbitration and stops transmitting

Node — Arbitration phase data

Node	1	2	3	4	5	6	7	8	9	10	11
1	0	1	1								
5	0	1	0	0	0	0	1	1	1	1	1
Bus	D	R	D	D	D	D	R	R	R	R	R

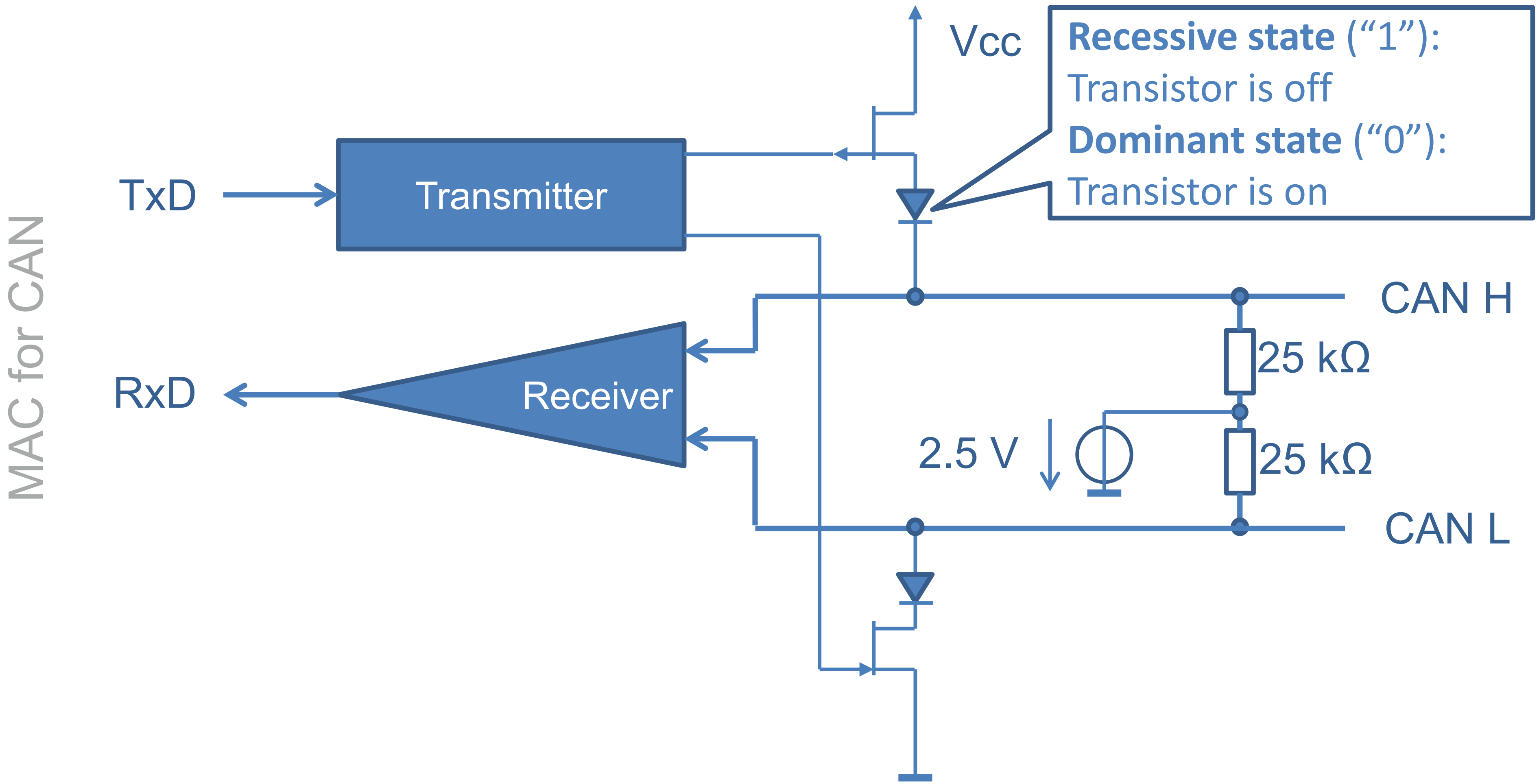
D: Dominant state R: Recessive state

Wired AND behaviour: Principle

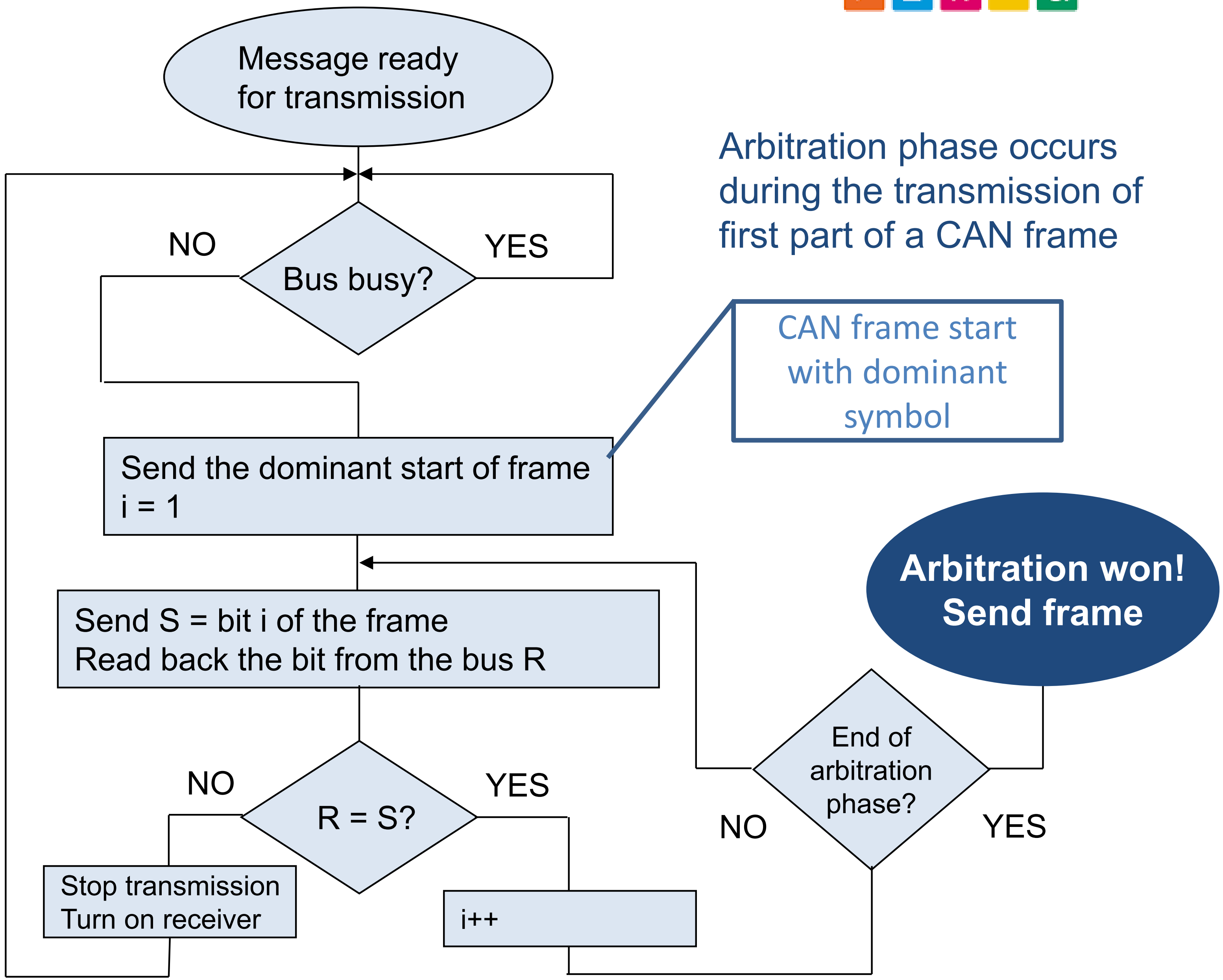


Transmitter Activity	TxD
Active - Send “1”	“1”
Active - Send “0”	“0”
Inactive	“1”

Wire AND behaviour: Implementation



Arbitration algorithm



Arbitration phase occurs during the transmission of first part of a CAN frame

Non-destructive collisions:
- The frame with the lowest bit pattern gets transmitted
- Transmission of other frames is stopped

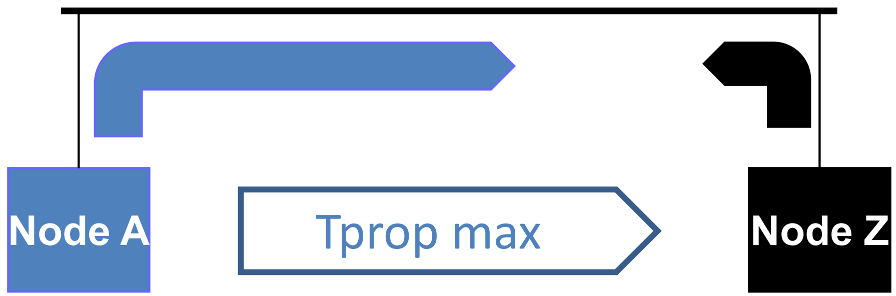
Arbitration: Influence of propagation delay

- On CAN bus, propagation delays have the same order of magnitude as bit duration
  - Example:
    - Propagation delay for a 100 m bus: 0.5 μs
    - Bit duration for 2 Mbit/s: 0.5 μs
- At a given time, nodes see different bus states:
  - What is the influence on bus arbitration?

Arbitration: Worst case

- Assume
  - Nodes A and Z are located at both ends of the CAN bus
    - **Tprop max** is the propagation time between Node A and Node Z
  - At time Ta, node A senses the bus as free and starts transmission
  - At time Tz (Ta < Tz < Ta + Tprop max), node Z senses the bus as free and starts transmission

- What will happen?



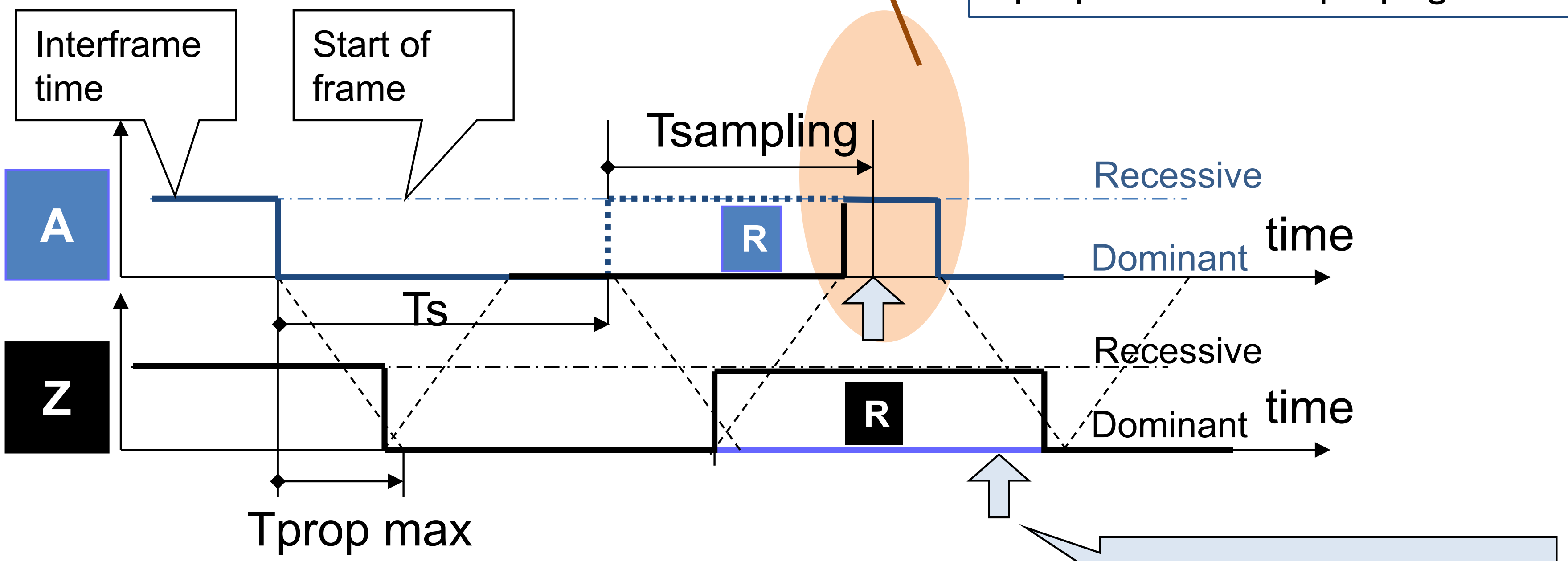
T prop = Delay on transmission lines + Delay in transceivers

200 ns to 300 ns, i.e. 40 m to 60 m line

Arbitration: Worst case

Sampling (read back) must produce "**Recessive**"

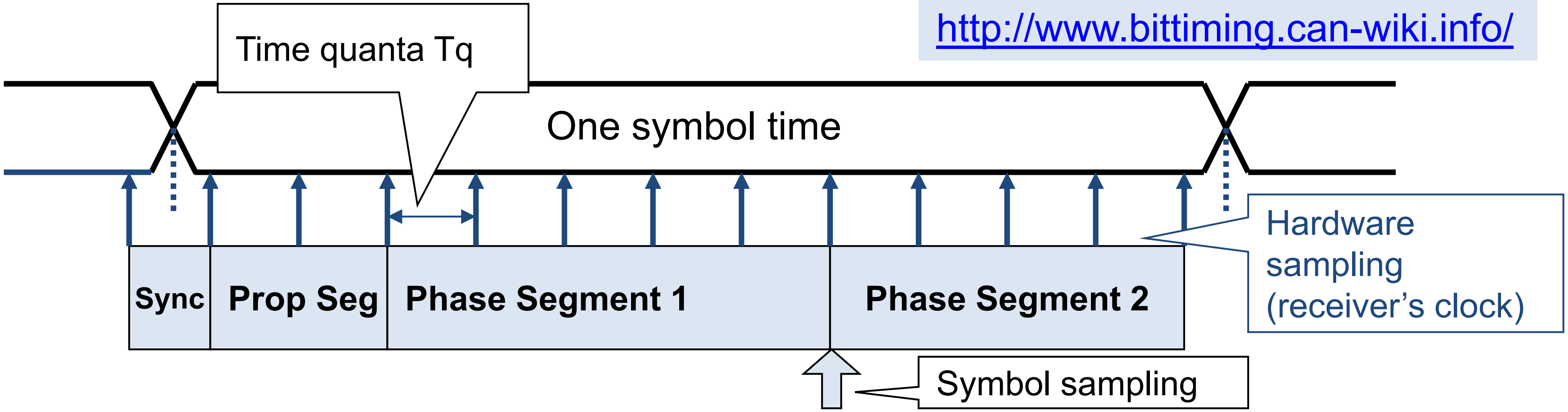
Ts: Symbol (bit) duration
Tsampling: Delay for read back since start of symbol
Tprop max: A – Z propagation time



Condition:
Tsampling > 2 Tprop max

Symbol sampling at receiver

See details at
http://www.bittiming.can-wiki.info/

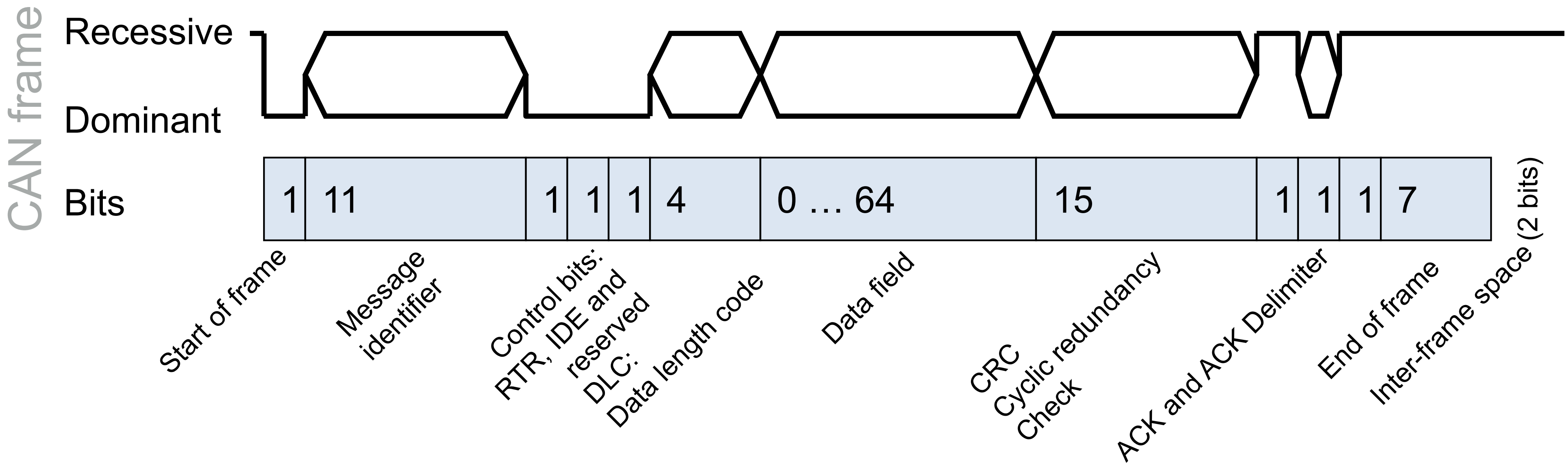


- The time quanta duration Tq and the number of Tq per symbol are configurable values
- Symbol duration is divided in several phases:
  - Sync segment: 1 Tq Used to synchronise node
  - Propagation segment: xpr* Tq Used to compensate for signal delays across the bus
  - Phase segment 1: x1* Tq Determines the position of sampling point, together
  - Phase segment 2: x2* Tq with Phase segment 2

*: Configurable values

Frame format

- Fields used for arbitration: Message identifier and RTR bit
  - Arbitration must always be resolved after these fields
- Message identifiers are either 11 bit long or 29 bit long



Frame fields

- "Start of frame"
  - Single symbol – dominant state
- "Message identifier"
  - 11 bit parameter freely managed by the application designers
  - Most common use of a Message identifier: Source addresses
    - Message identifiers can also be used to encode a source node address and a destination node address
    - Mostly "write" operation using Producer Consumer model
- "Control bits"
  - RTR (Remote Transmission Request): (not in CAN FD)
    - RTR = 1 -> Client "read" operation in Client Server model
  - IDE (Identifier Extension):
    - Enable 29-bit message identifier instead of 11-bit if 1
      (in this case, the 18 remaining identifier bits comes after this bit)

Frame fields

- "Data Length Codes" (DLC)
  - 4 bits -> 16 values, but only 9 (0…8) permitted values
  - Specifies the length of the Data field
    - DLC = 0 => Length = 0 bits; 1 => 8; 2 => 16; 3 => 24; 4 => 32; 5 => 40; 6 => 48; 7 => 56; 8 => 64
  - **Why so "short" messages?** Data encodes typically a single measurement values or a group of related measurement values
- "Data"
  - Content can be freely managed by software
  - Contains mostly only a value
  - All actively receiving nodes must be capable to interpret its content
- "Cyclic Redundancy Check" (CRC)
    - Hamming distance: 5
    - Rather long CRC (15 bits) for rather short frames
    - All receiving nodes perform the error detection

Frame fields

- "First delimiter"
  - Time to let the receiver(s) check the CRC
  - Always recessive
- "Acknowledge" (ACK)
  - Set by each receiver
  - No error detected:
    - Dominant state
  - Error detected:
    - Recessive state
  - Transmitter receives an explicit acknowledge as long as at least one receiver had a positive CRC check
    - Whether you like it or not
  - Behaviour of the transmitter upon error is not defined
    - Most probably retry
- "ACK delimiter"
  - Always recessive
- "End of frame"
  - 7 symbols with recessive states
- Inter-frame space
  - Recessive state during at least 2 symbol periods

Filtering of incoming frames

Frame filtering

Goal:
Reduce load on microcontroller

- Hardware filter:
  - Only “interesting” frames pass through a *software programmable* hardware filter

“Interesting” IDs	1	1	1	0	1	X	X	X	X	0	X
Mask	1	1	1	1	1	0	0	0	0	1	0
Pattern	1	1	1	0	1	0	0	0	0	0	0

  - Possibly several hardware filters in parallel

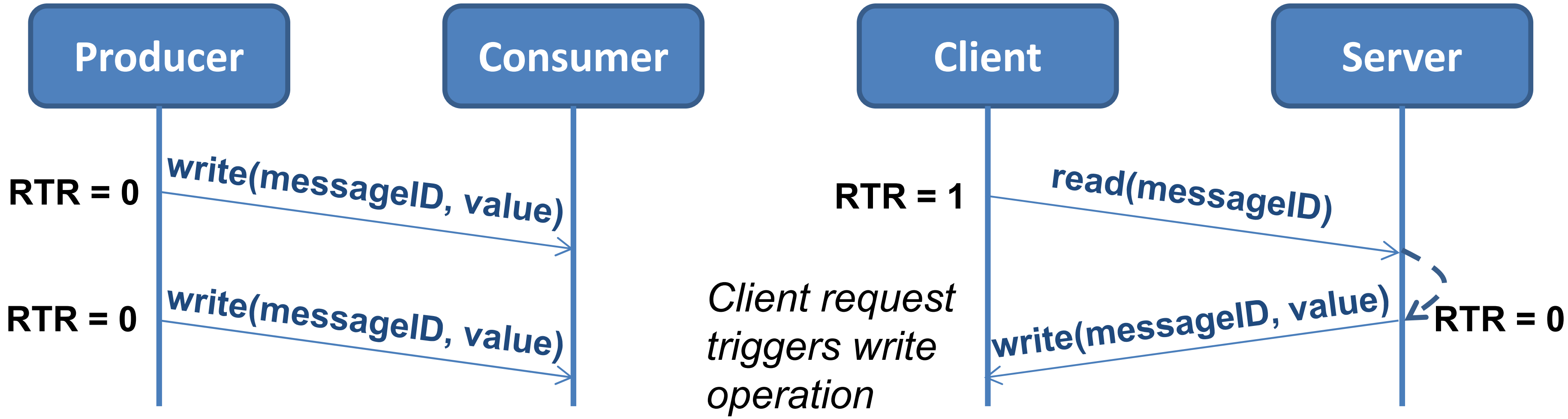
- Software filter:

```
currentId = readId(&inMessage) ;
switch (currentId) {
 case(ID1): {
   …
      break ;
 }
 case(ID2): {
   …
   break ;
 }
}
```

Uninteresting IDs are simply not handled in the `switch` instruction

Producer Consumer and Client Server models

Programming model



"write" operation

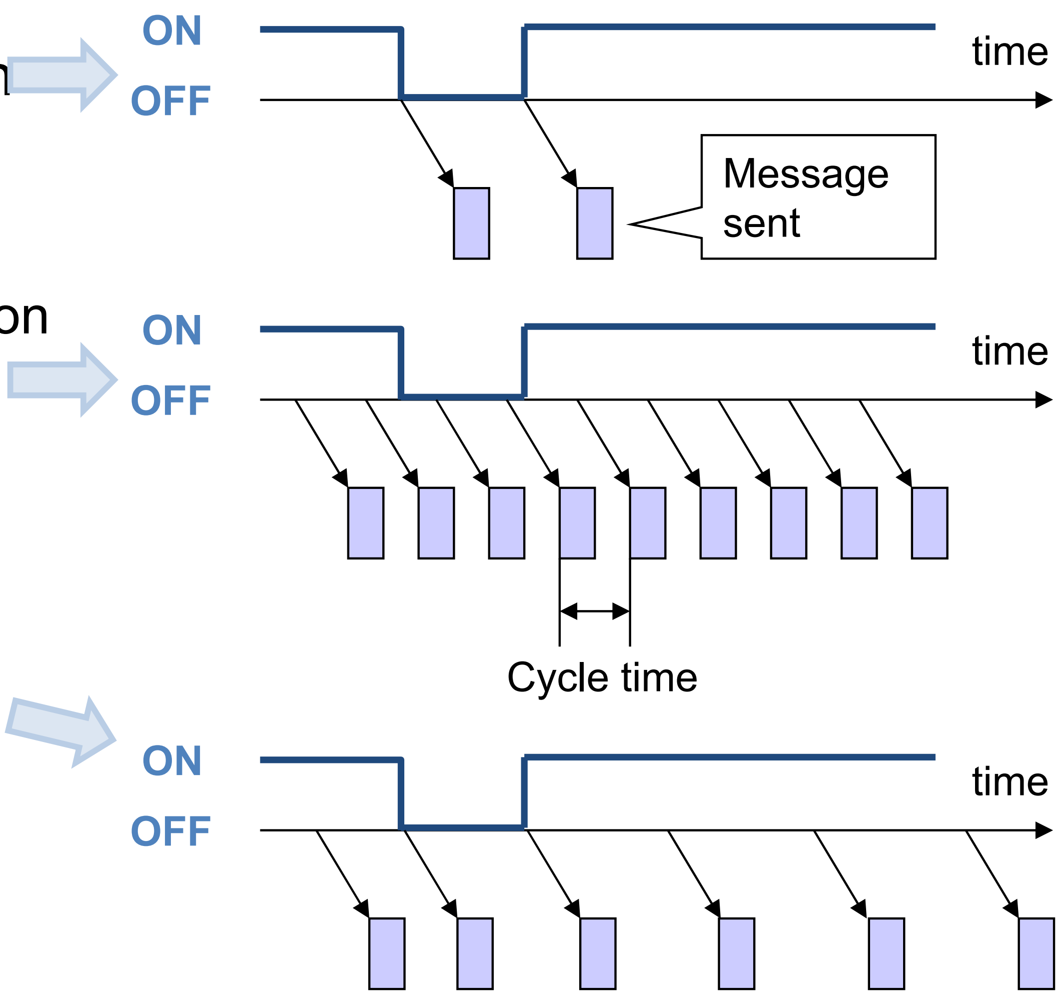
"read" operation

No arbitration problem between a Client read and a Producer write, as the RTR bit is part of the arbitration field

Triggering communication in Producer Consumer model

Programming model

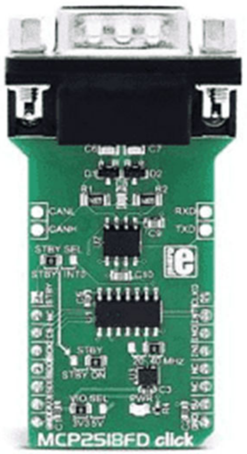
- Event triggered transmission
  - Event: Input state change
- Time triggered transmission
  - Periodic sample transmission
  - Period may be adapted to each input
- Event & time triggered transmission
  - "Watchdog" between (presumably rare) events
- Choice of transmission trigger is left open by CAN
  - Designer choice!

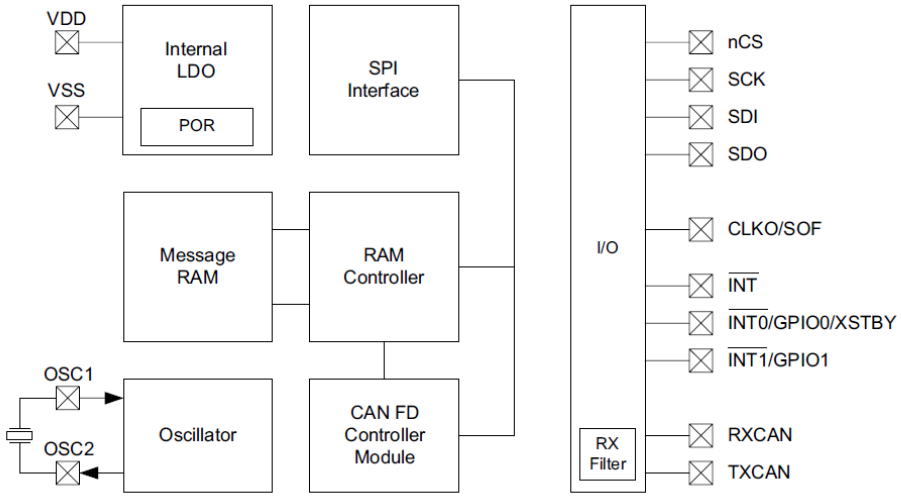


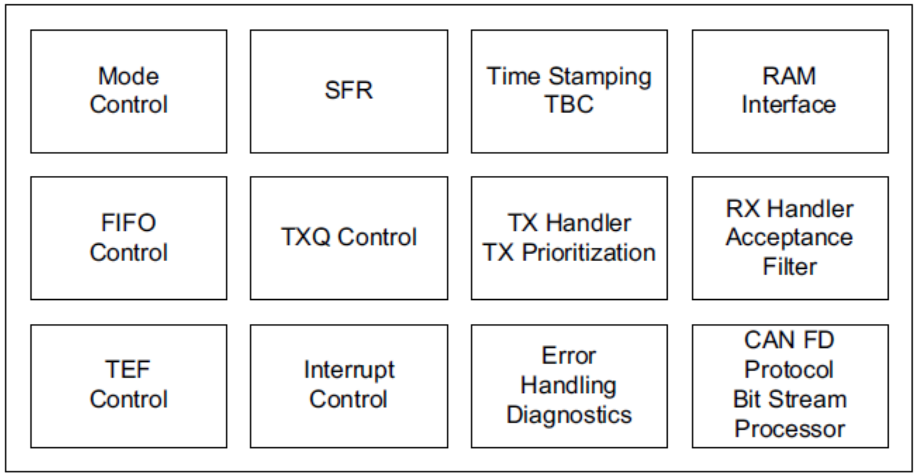
CAN identifiers assignment

- CAN identifiers are managed freely by the development team
  - Only restriction: two nodes may not send CAN messages with the same CAN Identifier
    - arbitration
- To be considered:
  - Capability to support hardware filtering:
    - Identifiers assignment should enable / simplify filtering
  - Priority
    - Urgent messages should have lower CAN identifiers
  - Ad hoc networking:
    - Support of an a priori unknown number of devices of several types
    - CAN Identifiers could be divided in three parts
      A. Device type
      B. Device identifier
      C. Variable source address
    - Centralised or distributed strategy to assign device identifiers

PIC CAN interface : MCP2518 FD







CAN FD
Controller module

PICEBS3 CAN library

CanInit(**CAN_BITTIME_SETUP** bitTime)

- Needs to be called at start
- bitTime: speed selection in *enum* CAN_BITTIME_SETUP

- Initialise CAN interface in classic mode (no FD)
- Creates an TX fifo of 5 messages
- Creates an RX fifo of 16 messages
- All messages are “timestamped”
- Interrupts are not implemented

Caution: use the slot 1 or Mikrobus interfaces !

PICEBS3 CAN library

CanSend(**CAN_TX_MSGOBJ** * txObj, **uint8_t** * txd)

- Used to send a message on the CAN bus
- txObj: *pointer* to message object
- txd: *pointer* to data to send
- returns ‘0’ if placed in tx buffer

Code example:

```
CAN_TX_MSGOBJ txObj;
uint8_t txd[8] = {0,1,2,3,4,5,6,7};

txObj.bF.id.ID = 0x300;            // standard identifier example
txObj.bF.ctrl.DLC = CAN_DLC_8;     // 8 bytes to send
txObj.bF.ctrl.RTR = 0;             // no remote frame
txObj.bF.id.SID11 = 0;             // only used in FD mode
txObj.bF.ctrl.FDF = 0;             // no CAN FD mode
txObj.bF.ctrl.IDE = 0;             // standard identifier format
txObj.bF.ctrl.BRS = 0;             // no data bitrate switch (FD mode)
txObj.bF.ctrl.ESI = 0;             // transmitting node error control
CanSend(&txObj, txd);
```

PICEBS3 CAN library

CanReceive(**CAN_RX_MSGOBJ** * rxObj, **uint8_t** * rxd)

- Used to get a message from the CAN bus
- rxObj: *pointer* to message object
- rxd: *pointer* to data to get
- returns ‘0’ if a message **was read** from RX fifo

Code example:

```
CAN_RX_MSGOBJ rxObj;
uint8_t rxd[8];

if(CanReceive(&rxObj, rxd) == 0)        // read a message if any
{
  if(rxObj.bF.id.ID == 0x300)           // check ID of messages
  {
    data = rxd[0];                      // get one data …
  }
}
```

PICEBS3 CAN library

CanSetFilter(**CAN_FILTER** filter, **CAN_FILTEROBJ_ID** * fObj, **CAN_MASKOBJ_ID** * mObj)

- Used to set a filter (mandatory for CAN reception)
- filter: one of 32 available filters
- fObj: pointer to filter identifier
- mObj: pointer to mask selection

Code example:

```
CAN_FILTEROBJ_ID fObj;
fObj.ID = 0x123;          // standard filter 11 bits value example (i.e., the value
                          // we are looking for)
fObj.SID11 = 0;           // 12 bits only used in FD mode
fObj.EXIDE = 0;           // assign to standard identifiers

CAN_MASKOBJ_ID mObj;
mObj.MID = 0x7FF;         // check all the 11 bits in standard ID
mObj.MSID11 = 0;          // 12 bits only used in FD mode
mObj.MIDE = 1;            // match identifier size in filter
CanSetFilter(CAN_FILTER0, &fObj, &mObj);
```

Exercice

Setup filters to receive only the 11 bits messages IDs below:

0x13x, 0x7FF

Send the message ID 0x33 below:

‘H’, ‘e’, ‘l’, ‘l’, ‘o’

PICEBS3 CAN library

- A lot a special function not explained (loopback, errors, …)
- FD mode not used in laboratories (more than 8 bytes, dynamic speed)
- Interruptions not used