2IMN20 - Real-Time Systems

Real-Time Communication

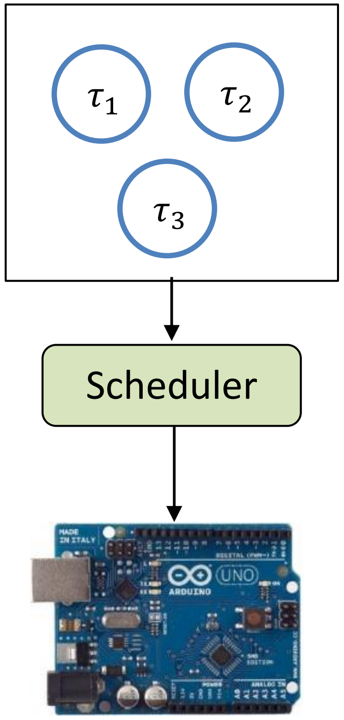
Geoffrey Nelissen

2023-2024

Agenda

- Introduction to real-time communication
- Describe the CAN protocol
- Analyzing the response time of messages on CAN

What you have learned so far



How to **model** real-time tasks

What can go wrong when tasks **access shared resources**

How to **protect** to access **shared resources**

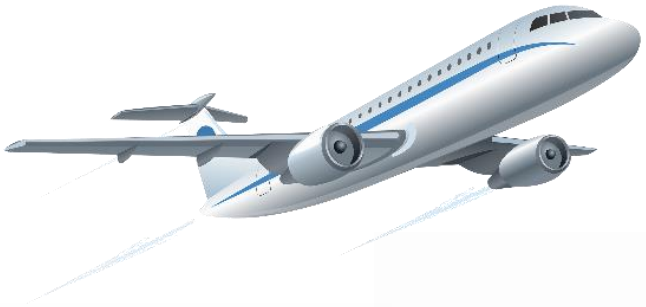
How to **isolate applications using servers and hierarchical scheduling**

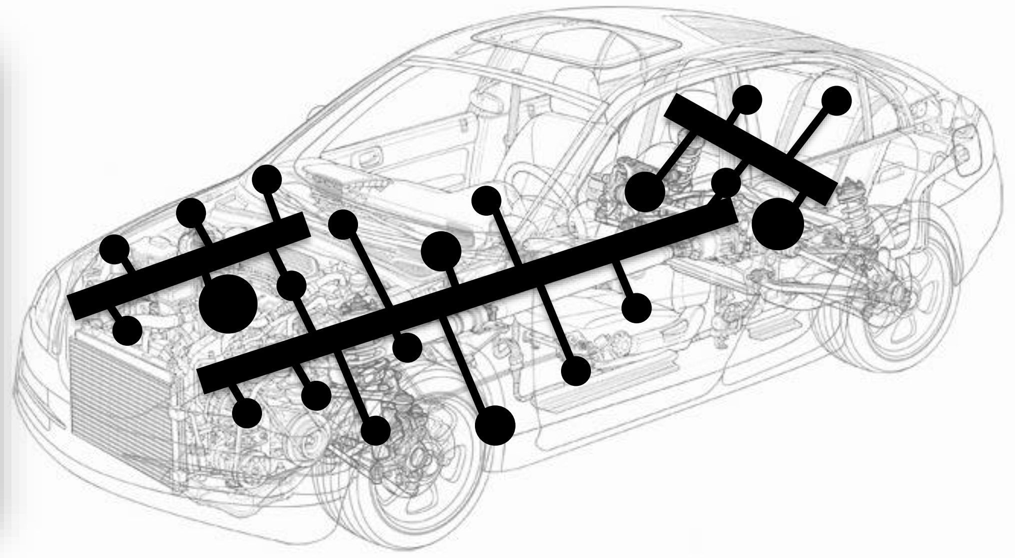
How the OS selects tasks to execute using a **scheduling policy**

How analyze the **schedulability/response time** of tasks for a given a scheduler and execution platform

Main assumption so far:
all tasks execute on a single chip

Many modern systems are distributed systems

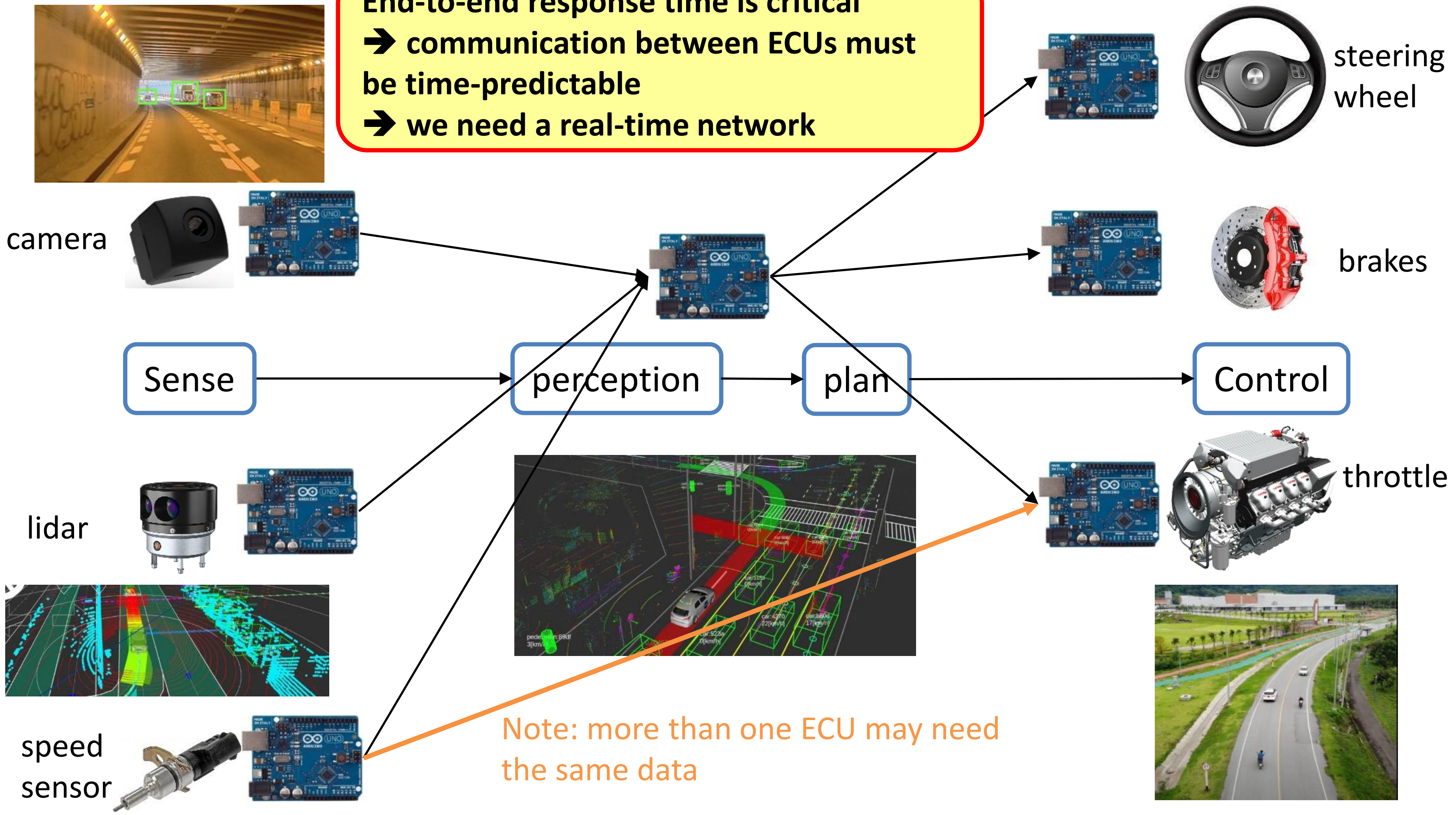




Example: A modern car contains 70 to 100 execution nodes that need to communicate and exchange data

Many modern systems are distributed systems

Example: autonomous/assisted driving



End-to-end response time is critical
➔ communication between ECUs must be time-predictable
➔ we need a real-time network

Note: more than one ECU may need the same data

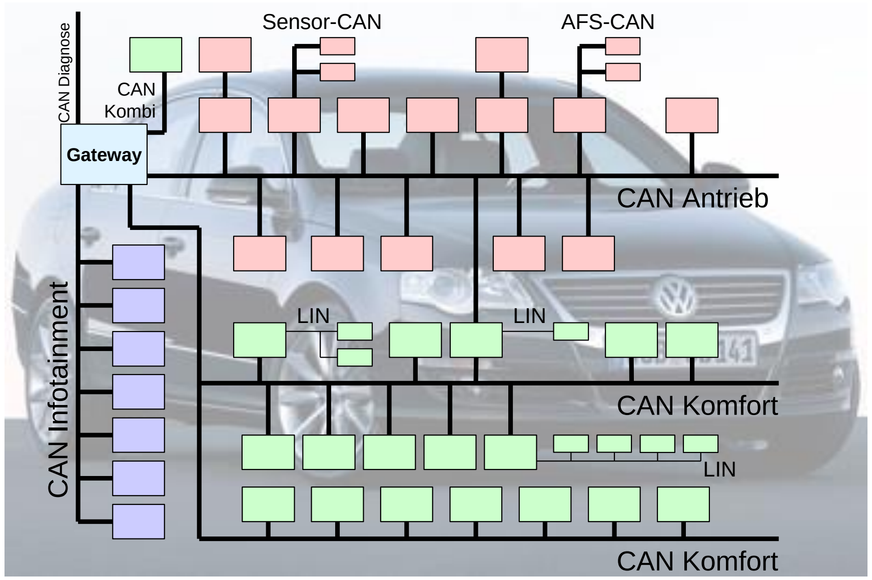
Old approach: point-to-point communication

As the number of electronic devices grew

- the wiring gets more “messy”
- the weight of the car increases
- maintenance and updates are complex

Modern approach: networked architecture

- Computing nodes are connected through **buses** (e.g., CAN, FlexRay, LIN) and **networks** (e.g., Ethernet AVB, Ethernet TSN)
- **Multiple networks** for different types of
  - **applications** (e.g., drive train, infotainment, comfort such as windows, AC, internal lights),
  - **data** (e.g., large data such as camera images and lidar point clouds, or small data such as speed, distance, control command, …)
  - **zones** in the system



- Different networks may be connected through gateways
- **Benefits**
  - Cost reduction (less wires, reduced labor, lower weight)
  - Flexibility (adding/ removing an ECU is mostly plug-and-play)

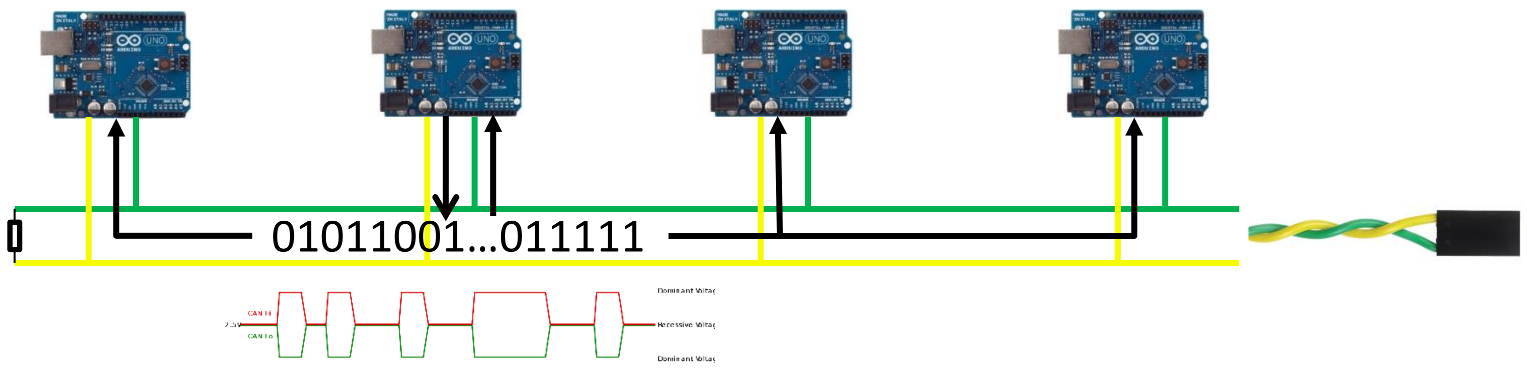
A widely used real-time communication protocol:
CAN

CAN – Control Area Network

- Originally developed for the automotive industry needs
  - 1983: BOSCH starts CAN development (Intel joins 1985)
  - 1987: First CAN chip
  - 1990: First car with CAN (Mercedes S-class)
  - 1993: ISO standard
  - 2012: release of CAN FD by BOSCH (larger data, 8 times faster than CAN)
  - Since 2018: definition of CAN XL (larger data, 20 times faster than CAN)

- Used in automotive, robotics, factory automation, trains, …

CAN – Physical layer



- Two twisted electric wires (improved robustness to electromagnetic noise)
- ECUs are connected to the wires
- Messages are sent as a sequence of 0 and 1
  (voltage difference between the two wires)
- All ECUs can read what is written on the CAN bus
- Transmission at maximum 1Mbps (CAN FD: max 8 Mbps, CAN XL: max 20 Mbps)

CAN protocol

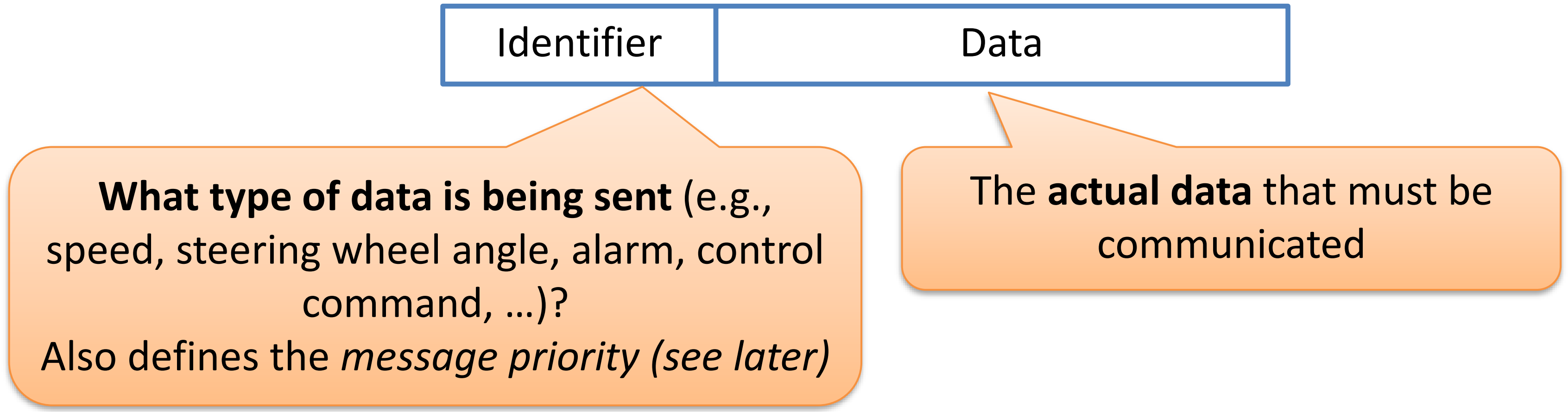
Content of a message (frame)?

What should a message contain? Any idea?

Which ECU can send a message when?

CAN frame format

- Two main fields are required

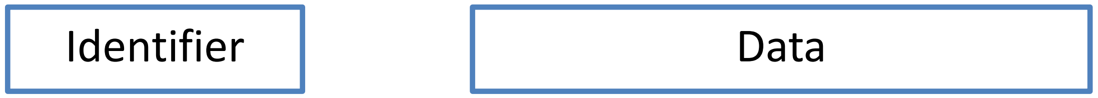

Identifier
Data
What type of data is being sent (e.g., speed, steering wheel angle, alarm, control command, …)?
Also defines the *message priority (see later)*
The **actual data** that must be communicated

- **Note 1**: In CAN, the data can be between 0 and 8 bytes long

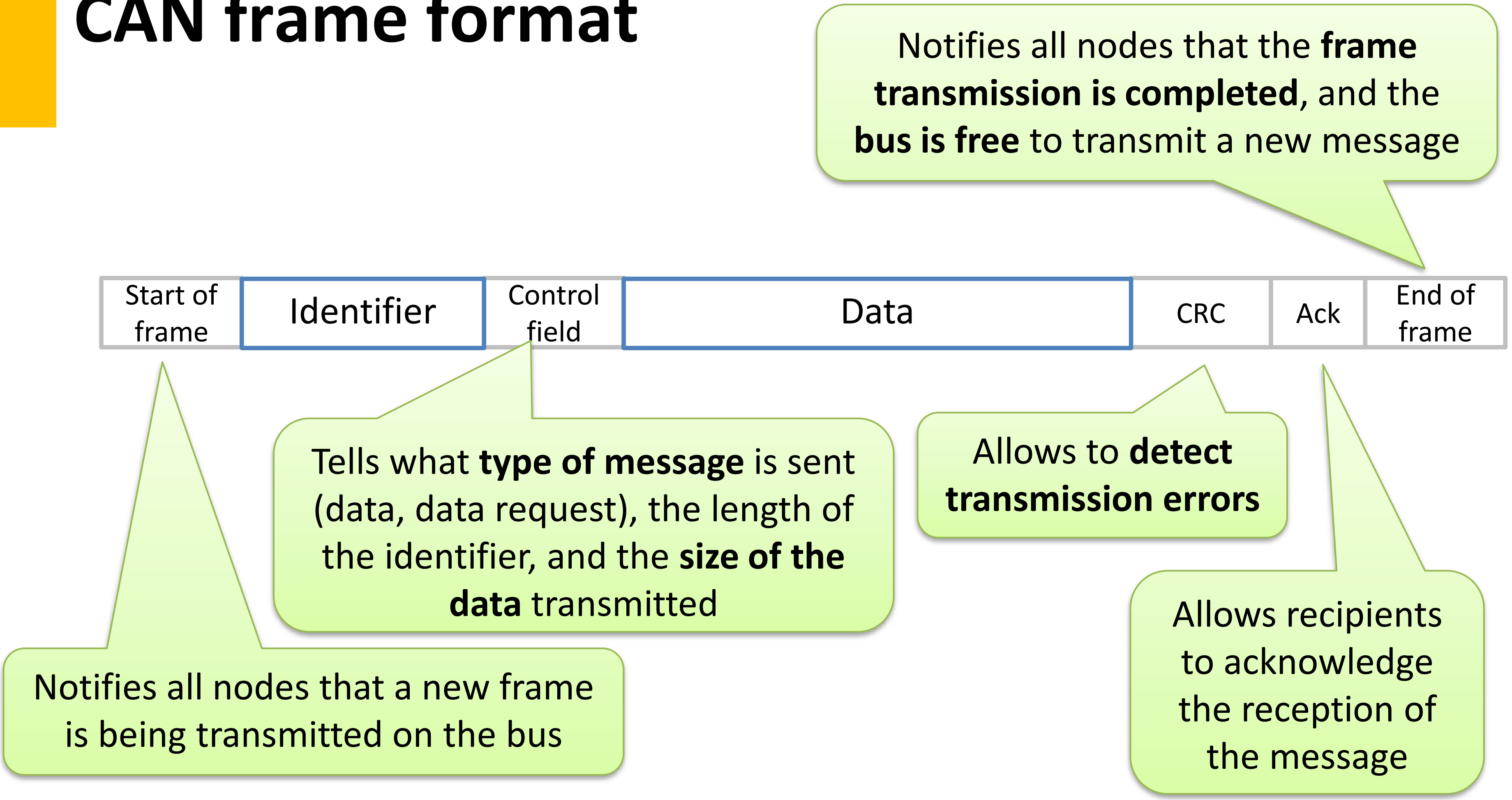
  (extended to 64 bytes with CAN FD and 2kB with CAN XL)

- **Note 2**: the message identifier can be 11 (at most 2048 different message types) or 29 (>536,000,000 message types) bits long

CAN frame format



CAN frame format



Notifies all nodes that the **frame transmission is completed**, and the **bus is free** to transmit a new message

Notifies all nodes that a new frame is being transmitted on the bus

Tells what **type of message** is sent (data, data request), the length of the identifier, and the **size of the data** transmitted

Allows to **detect transmission errors**

Allows recipients to acknowledge the reception of the message

CAN data frame

SOF	ID	RTR	Control	Data	CRC	CRC DEL	ACK	ACK DEL	EOF	IFS
1 bit	11 bits	1 bit	6 bits	0-8 bytes	15 bits	1 bit	1 bit	1 bit	7 bits	min 3 bits

SOF	- *Start of Frame*, start bit (always 0), used for signaling that a frame will be sent (the bus must be free)
ID	- ***Identifier*, identity for the frame and its priority**
RTR	- *Remote Transmission Request* (0=data frame, 1=data request frame)
Control	- indicates the length of the data field (value between 0 and 8)
Data	- **message data**
CRC	- *Cyclic Redundancy Check*,
CRC DEL	- *CRC delimiter* (always ‘1’)
ACK	- *Acknowledgement*
ACK DEL	- *ACK delimiter* (always ‘1’)
EOF	- *End of Frame* (always seven successive ‘1’)
IFS	- *Inter Frame Space*, minimum time between two frames sent on the bus (bus idle during the equivalent of three bits minimum

CAN protocol

Content of a message (frame)?

Which ECU can send a message when?

Why does that matter?

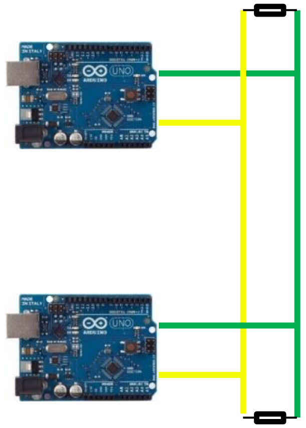
CAN bus access arbitration

- CAN uses a **CSMA/CR protocol**
  - **CSMA**: Carrier Sense Multiple Access
    - **All nodes can access the bus** at the same time
    - **All nodes can read from the bus** at the same time
  - **CR**: Collision Resolution
    - If **multiple nodes try to write** at the same time on the bus, the **collision is detected and only one will be allowed to continue**

- The arbitration is **priority driven**
  - **The identifier of a message is its priority**
  - The message with the lowest identifier always wins access first

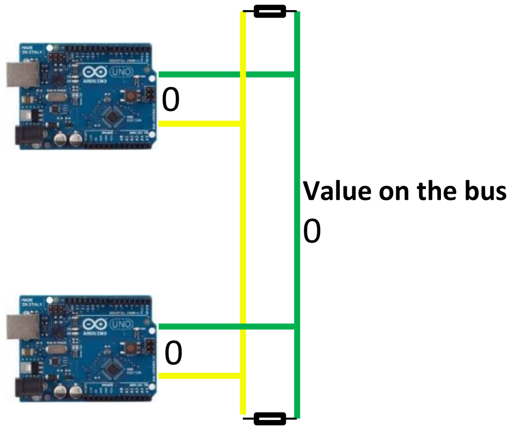
CAN bus access arbitration

- How does it work?
  - Assume two nodes write at the same time on the bus



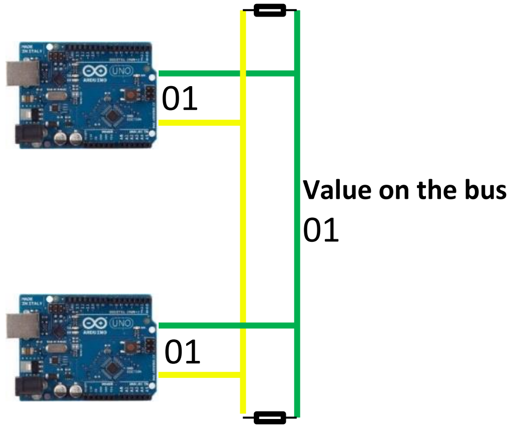
CAN bus access arbitration

- How does it work?
  - Assume two nodes try to access the CAN bus at the same time
    - If the two nodes write a ‘0’, the value on the bus is ‘0’



CAN bus access arbitration

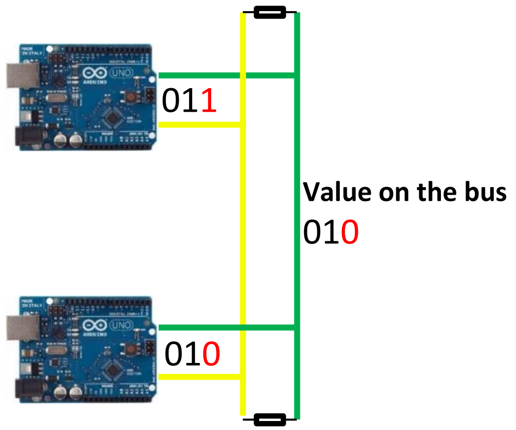
- How does it work?
  - Assume two nodes try to access the CAN bus at the same time
    - If the two nodes write a ‘0’, the value on the bus is ‘0’
    - If the two nodes write a ‘1’, the value on the bus is ‘1’



CAN bus access arbitration

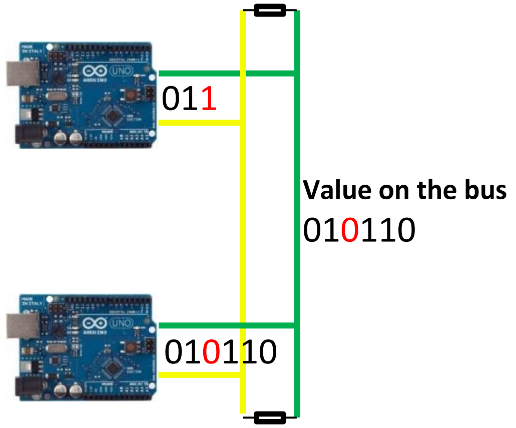
- How does it work?
  - Assume two nodes try to access the CAN bus at the same time
    - If the two nodes write a ‘0’, the value on the bus is ‘0’
    - If the two nodes write a ‘1’, the value on the bus is ‘1’
    - If the one node write a ‘0’ and the other a ‘1’, the value on the bus is ‘0’

➔ in case of conflict, ‘0’ wins over ‘1’



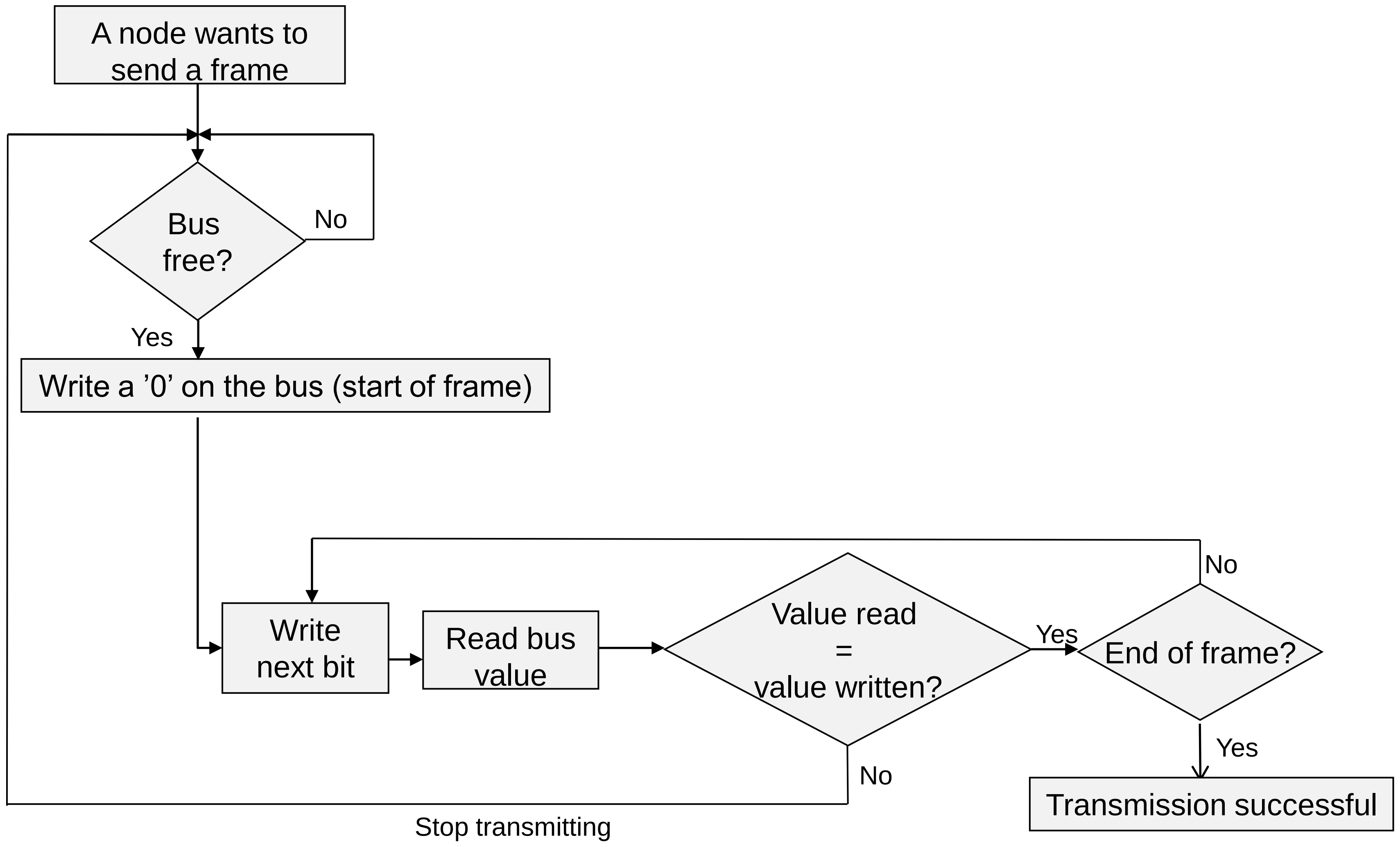
CAN bus access arbitration

- How does it work?
  - Assume two nodes try to access the CAN bus at the same time
    - If the two nodes write a ‘0’, the value on the bus is ‘0’
    - If the two nodes write a ‘1’, the value on the bus is ‘1’
    - If the one node write a ‘0’ and the other a ‘1’, the value on the bus is ‘0’
    - If a node reads a different value on the bus than what it sends, it stops sending and waits for the message of the other node to be completed



➔ Since the first field sent is the identifier, the message with the lowest identifier (first one to have a ‘0’) is the one that is transmitted on the bus

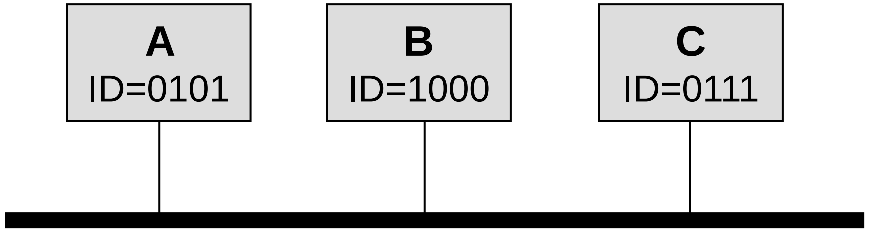
CAN bus access arbitration



CAN bus access arbitration (example)

Three ECUs try to send a message at the same time

- ECU A sends a message with ID 0101
- ECU B sends a message with ID 1000
- ECU C sends a message with ID 0111



Note: $0101_2 = 5_{10}$
$1000_2 = 8_{10}$
$0111_2 = 7_{10}$

➔ The message sent by ECU A has the highest priority

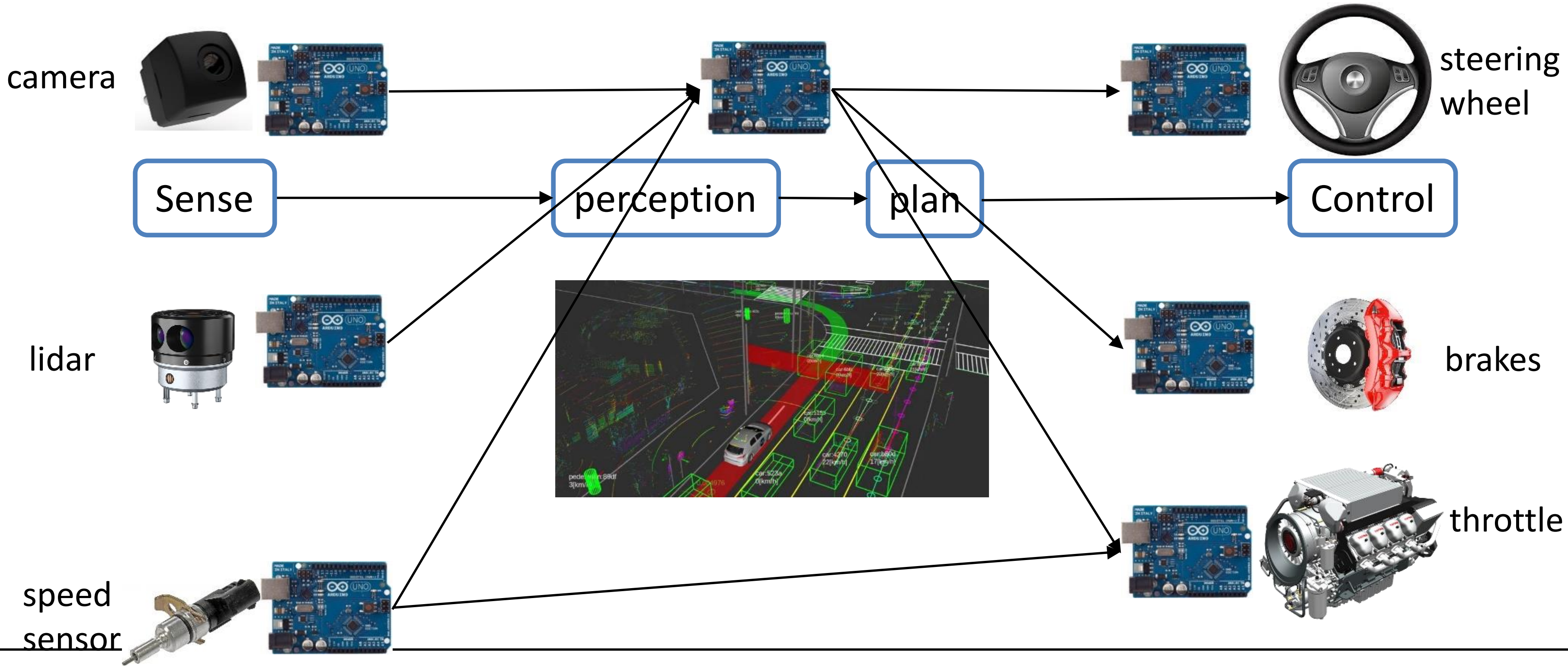
How does the arbitration look like if the nodes are sending simultaneously?

Node	ID	Bit 0	Bit 1	Bit 2	Bit 3	Bit 4	
A	0101	0	0	1	0	1	→ Sends the rest of the frame
B	1000	0	1	→ abort! (bit 1 ≠ bus value)			
C	0111	0	0	1	1	→ abort! (bit 3 ≠ bus value)	
Bus value:		0	0	1	0	1	

➔ All nodes start with writing a '0' to notify other nodes that they start transmitting a frame

A note on message identifiers

Important: identifiers are assigned to message types. They are not the same as the node ID. A single CAN node (i.e., ECU) may send messages with several different IDs. For example, the ECU in charge of planning may send commands to the steering wheel, the brakes and engine. Each will have a different message ID.
However, only one node should send messages with a given identifier, to ensure proper bus access arbitration (only one node can send a message).



Practical consideration: Bit stuffing

- **When transmitting a frame, we must avoid two bit-patterns**
  - 000000: used for error signaling
  - 111111: used for end of frame signaling

- ***Bit stuffing***:
  - the sender node automatically puts extra bits of opposite value in the message to prevent forbidden bit-patterns (i.e., **add a ‘1’ after five ‘0’s, and add a ‘0’ after five ‘1’s**) **except in the ack and end of frame fields**
  - The receiver node automatically reconstruct the original frame by removing the extra bits

Example:

Original frame:	…00101000000101…
Bits sent on the bus by the sender:	…001010000010101…

➔ a ‘1’ is added after five ‘0’s to avoid sending six successive ‘0’s

Bit stuffing

- **When transmitting a frame, we must avoid two bit-patterns**
  - 000000: used for error signaling
  - 111111: used for end of frame signaling

- ***Bit stuffing***:
  - the sender node automatically puts extra bits of opposite value in the message to prevent forbidden bit-patterns (i.e., **add a ‘1’ after five ‘0’s, and add a ‘0’ after five ‘1’s**) **except in the ack and end of frame fields**
  - The receiver node automatically reconstruct the original frame by removing the extra bits

Example:

Original frame:	…00101000000101…
Bits sent on the bus by the sender:	…001010000**1**0101…
Receiver removes extra bits:	…00101000000101…

Bit stuffing

What sequence of 0s and 1s is sent on the network after bit stuffing?

- **Example 2:**

Original frame: 01111 1000 0111 1000 0111 1

First attempt to bitstuffing: 01111 10000 0111 1000 0111 1

Added bit created a sequence of five ‘0’s

Full bitstuffing: 01111 10000 01111 10000 01111 1

Extra bit after **5** original bits	Extra bit after **4** original bits	Extra bit after **4** original bits	Extra bit after **4** original bits	..etc…

To avoid forbidden bit patterns we may need to insert one extra bit after the first five bits and one extra bit after each four next original bits

Message acknowledgement

SOF	ID	RTR	Control	Data	CRC	CRC DEL	ACK	ACK DEL	EOF	IFS
1 bit	11 bits	1 bit	6 bits	0-8 bytes	15 bits	1 bit	1 bit	1 bit	7 bits	min 3 bits

- After sending the data and CRC, the sender node writes two successive ‘1’s on the bus (the CRC delimiter and the Ack field)
- If any node listening to the bus received the data correctly (i.e., the CRC check on the received data is successful), the listening node writes a ‘0’ on the bus during the Ack field.
- If the sender reads a ‘0’ on the bus during the Ack field, it then knows the frame was successfully transmitted and received. Otherwise, it may decide to retransmit the frame.

Agenda

- Introduction to communication in automotive real-time systems
- CAN protocol
- **Schedulability analysis**

CAN response time analysis

Properties of CAN

- At most **one message** may be **sent at a time**
- **Highest priority** messages are **sent first**
- Messages are **non-preemptive** (once a message starts transmitting and wins the arbitration, they are sent to completion)

It is similar to…
single-core non-preemptive FP scheduling

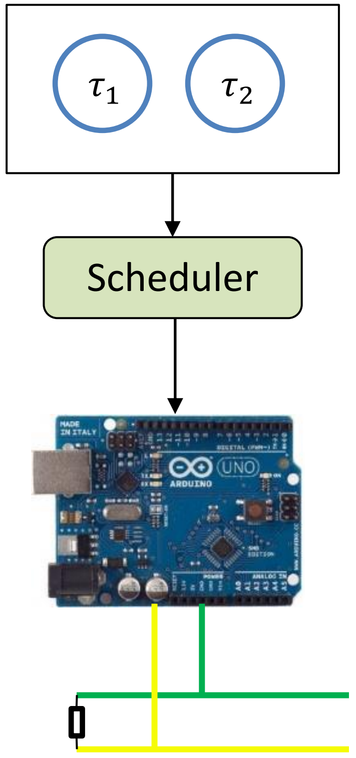
We can use the RTA for single-core non-preemptive FP scheduling to analyze the response time of messages in CAN

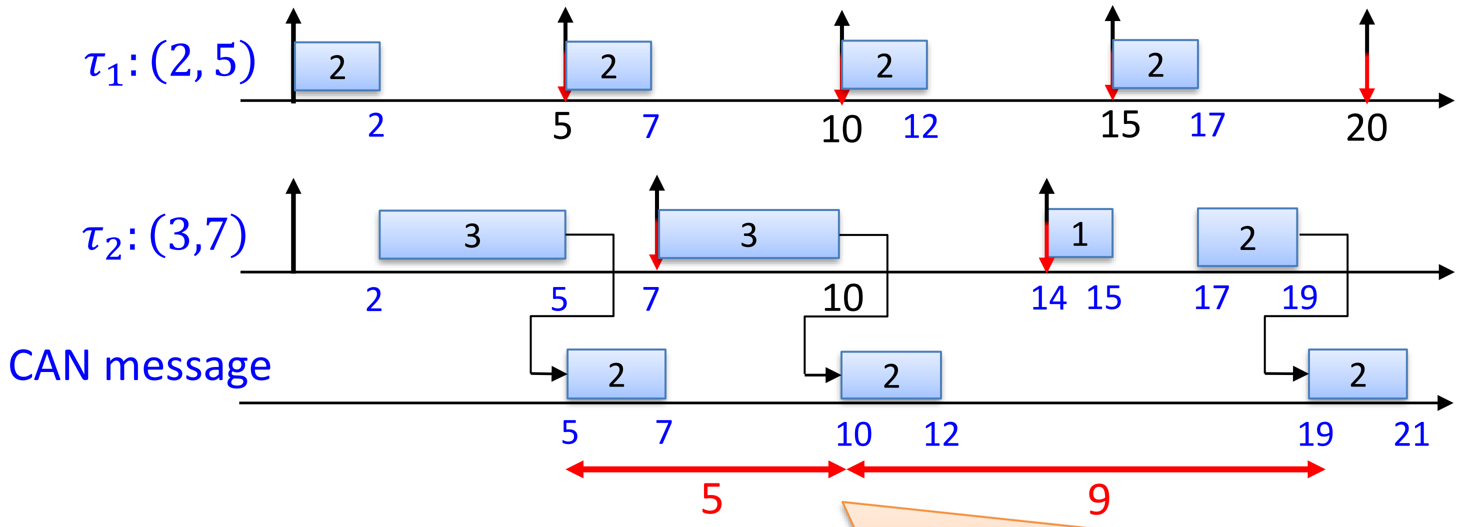
Important note:

- **Messages may not be sent periodically.** They may experience **jitter**.

Example of release jitter

- Two periodic tasks τ_1 and τ_2 execute on an ECU.
- τ_2 sends messages on the CAN bus.
- The scheduler used on the ECU is FP preemptive.
- τ_1 has higher priority than τ_2

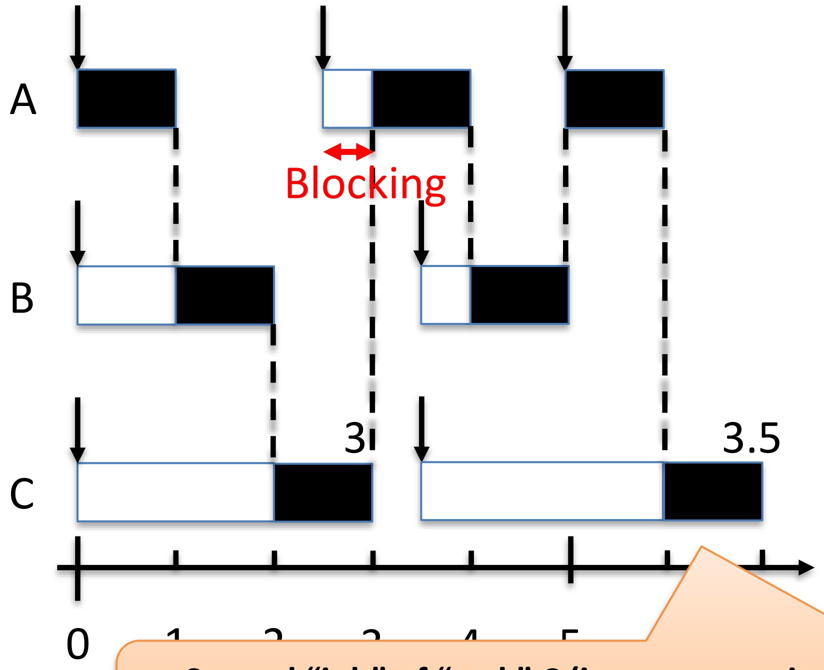




Not periodically released.
However, can be modeled by a periodic task with
period T=7 and release jitter $\sigma = 2 = WCRT - BCRT$ (see lectures 5 and 11)

Impact of non-preemptive execution

Message type	Identifier (=priority)	*T*	*C*
A	1	2.5 ms	1 ms
B	2	3.5 ms	1 ms
C	3	3.5 ms	1 ms



- **Higher-priority messages may experience blocking caused by lower-priority messages** due to non-preemptivity (i.e., we cannot access the bus if the bus is not idle whatever our priority)

- First message instance may not be the one experiencing the worst-case response time

Second "job" of "task" C (i.e., message instance of message type C) experiences a higher WCRT

Worst-case response time analysis for CAN messages

Level-i Active Period:

$$L_i = B_i + \sum_{M_j \in hep_i} \left\lceil \frac{L_i + \sigma_j}{T_j} \right\rceil C_j$$

$$B_i = \max_{M_k \in lp_i} \{C_k\}$$

Number of jobs to be analysed:

$$K_i = \left\lceil \frac{L_i + \sigma_i}{T_i} \right\rceil$$

C_i = max number of bits in message M_i divided by the baudrate of the CAN bus (in bps)

hep_i = set of messages with equal or higher priority than M_i (including M_i)

hp_i = set of messages with higher priority than M_i

lp_i = set of messages with lower priority M_i

Start time of the kth job of message M_i :

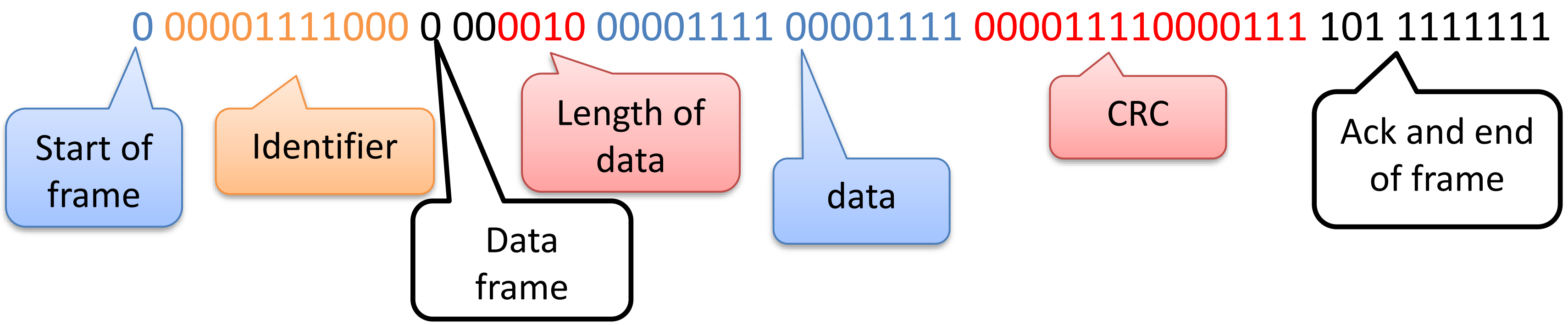
$$s_{i,k} = B_i + (k-1)C_i + \sum_{M_j \in hp_i} \left(\left\lfloor \frac{s_i + \sigma_j}{T_j} \right\rfloor + 1 \right) \cdot C_j$$

Worst-Case Response Time:

$$R_i = \max_{k \in [1, K_i]} \{ s_{i,k} + C_i - (k-1)T_i \} + \sigma_i$$

Exercise

- Consider the following CAN message sent on a 1Mbps CAN bus



0 00001111000 0 000010 00001111 00001111 000011110000111 101 1111111

What is the transmission time (C_i) of the message on the bus?

Exercise

- Consider the following CAN message sent on a 1Mbps CAN bus

  0 00001111000 0 000010 00001111 00001111 000011110000111 101 1111111

- Message after bit stuffing

  0 0000111110000 01 000010 000011111 0 000011111 0 00001111100000 1111 101 1111111

Note: no bit stuffing in the ack and end of frame fields

- The message is 70 bits long after bit stuffing. The bus has a baudrate of 1Mbps. We should also account for 3 bits for inter-packet gap.

  ➔ the transmission time is $\frac{70+3}{1Mbps} = 73\ \mu s$

CAN - Example

Assume a CAN system with three nodes. Two tasks on each node. One task on each node sends a message.

Node 1

Task	T (µs)	C (µs)	Msg
A1	10000	3000	$M1$ (id = 011)
A2	7000	1000	-

Node 2

Task	T (µs)	C (µs)	Msg
B1	5000	1000	$M2$ (id = 001)
B2	4000	1000	-

Node 3

Task	T (µs)	C (µs)	Msg
C1	4000	1000	$M3$ (id=000)
C2	10000	1000	-

Assumptions:

- Bus speed:1 Mbit/s
- Task instances send their messages at the end of the execution
- The size of each message is 135 bits
- Task priority assignment is according to Rate Monotonic

Questions:

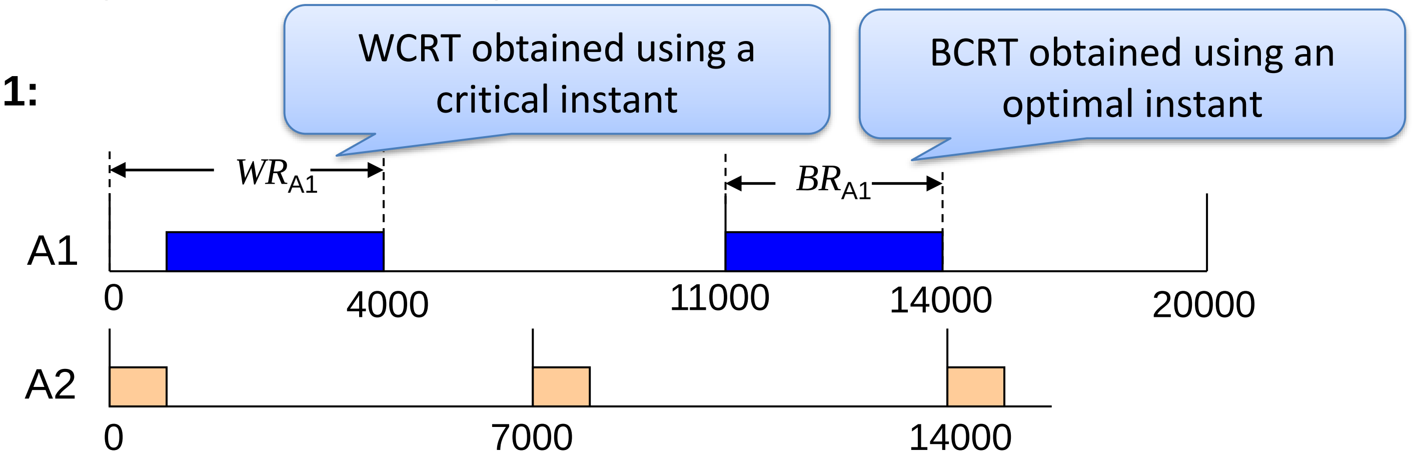
a) Calculate release jitter for the messages

b) Calculate response times for the messages

CAN - Example

a) Activation jitter of the messages

Node 1:


WCRT obtained using a critical instant
BCRT obtained using an optimal instant
WR_{A1}
BR_{A1}
A1
0
4000
11000
14000
20000
A2
0
7000
14000

$$\sigma_{M1} = WR_{A1} - BR_{A1} = 4000 - 3000 = \mathbf{1000}$$

Node 2:

$$\sigma_{M2} = WR_{B1} - BR_{B1} = 2000 - 1000 = \mathbf{1000}$$

Node 3:

$$\sigma_{M3} = WR_{C1} - BR_{C1} = \mathbf{0}$$

(Note: No jitter, because C1 has highest priority)

CAN – Example

b) Response times (priorities: $M3$ – high prio, $M2$ – middle, $M1$ – low)

$M3$: $lp(m3) = \{m1, m2\}$ → B_{m3} = max(C_{m1}, C_{m2}) = max(135µs, 135 µs) = 135 µs

K_{m3}: First determine L_{m3}

$$L_{m3}^{(0)} = B_{m3} + C_{m3} = 270\mu s$$

$$L_{m3}^{(1)} = B_{m3} + \left\lceil \frac{L_{m3}^{(0)} + \sigma_{m3}}{T_{m3}} \right\rceil C_{m3} = 135 + \left\lceil \frac{270+0}{4000} \right\rceil 135 = 270\mu s$$

$$\rightarrow L_{m3} = 270 \rightarrow \quad K_{m3} = \left\lceil \frac{L_{m3} + \sigma_{m3}}{T_{m3}} \right\rceil = \left\lceil \frac{270+0}{4000} \right\rceil = 1$$

Hence, we only need to consider a single ”job” of $m3$.

$$s_{m3,0} = B_{m3} + 0 = 135\mu s$$

$$R_{m3} = R_{m3,0} = \sigma_{m3} + s_{m3,0} + C_{m3} = 0 + 135 + 135 = 270\mu s$$

CAN – Example

b) Response times (priorities: $m3$ – high prio, $m2$ – middle, $m1$ – low)

M2: $lp(m2) = \{m1\} \rightarrow B_{m2} = C_{m1} = 135$

K_{m2}: First determine L_{m2}

$$L_{m2}^{(0)} = B_{m2} + C_{m2} + C_{m3} = 405$$

$$L_{m2}^{(1)} = B_{m2} + \left\lceil \frac{L_{m2}^{(0)} + \sigma_{m2}}{T_{m2}} \right\rceil C_{m2} + \left\lceil \frac{L_{m2}^{(0)} + \sigma_{m3}}{T_{m3}} \right\rceil C_{m3}$$

$$= 135 + \left\lceil \frac{405 + 1000}{5000} \right\rceil 135 + \left\lceil \frac{405 + 0}{4000} \right\rceil 135 = 405\mu s$$

$$L_{m2} = 405 \rightarrow K_{m2} = \left\lceil \frac{L_{m2} + \sigma_{m2}}{T_{m2}} \right\rceil = \left\lceil \frac{405 + 1000}{5000} \right\rceil = 1$$

Hence, we only need to consider a single ”job” of m.

$$s_{m2,0}^{(0)} = B_{m2} + C_{m3} = 135 + 135 = 270$$

$$s_{m2,0}^{(1)} = B_{m2} + \left(\left\lfloor \frac{s_{m2,0}^{(0)} + \sigma_{m3}}{T_{m3}} \right\rfloor + 1 \right) C_{m3} = 135 + \left(\left\lfloor \frac{270 + 0}{4000} \right\rfloor + 1 \right) 135 = 270\mu s$$

$$R_{m2} = R_{m2,0} = \sigma_{m2} + s_{m2,0} + C_{m2} = 1000 + 270 + 135 = 1405\mu s$$

CAN – Example

b) Response times (priorities: m3 – high prio, m2 – middle, m1 – low)

M1: $lp(m1) = \{\ \} \rightarrow B_{m1} = 0$

$L_{m1} = 405\mu s$

$$K_{m1} = \left\lceil \frac{405 + 1000}{10000} \right\rceil = 1$$

Hence, we only need to consider a single ”job” of *M1*.

$s_{m1} = 270\mu s$

$$R_{m1} = R_{m1,0} = \sigma_{m1} + s_{m1,0} + C_{m1} = 1000 + 270 + 135 = 1405\mu s$$

Real-time networks in modern systems

- Limitations of CAN
  - Limited baudrate (1Mps for CAN, up to 20Mbps with CAN XL)
  - Limited data sizes can be transmitted (8 bytes for CAN, up to 2kB per message with CAN XL)

- Modern systems use a combination of networks depending on their needs. This includes
  - CAN
  - FlexRay (time-triggered transmissions)
  - Ethernet TSN (allow priority driven and time-triggered transmissions, allows preemptions and hierarchical scheduling similar to using servers)
  - …