

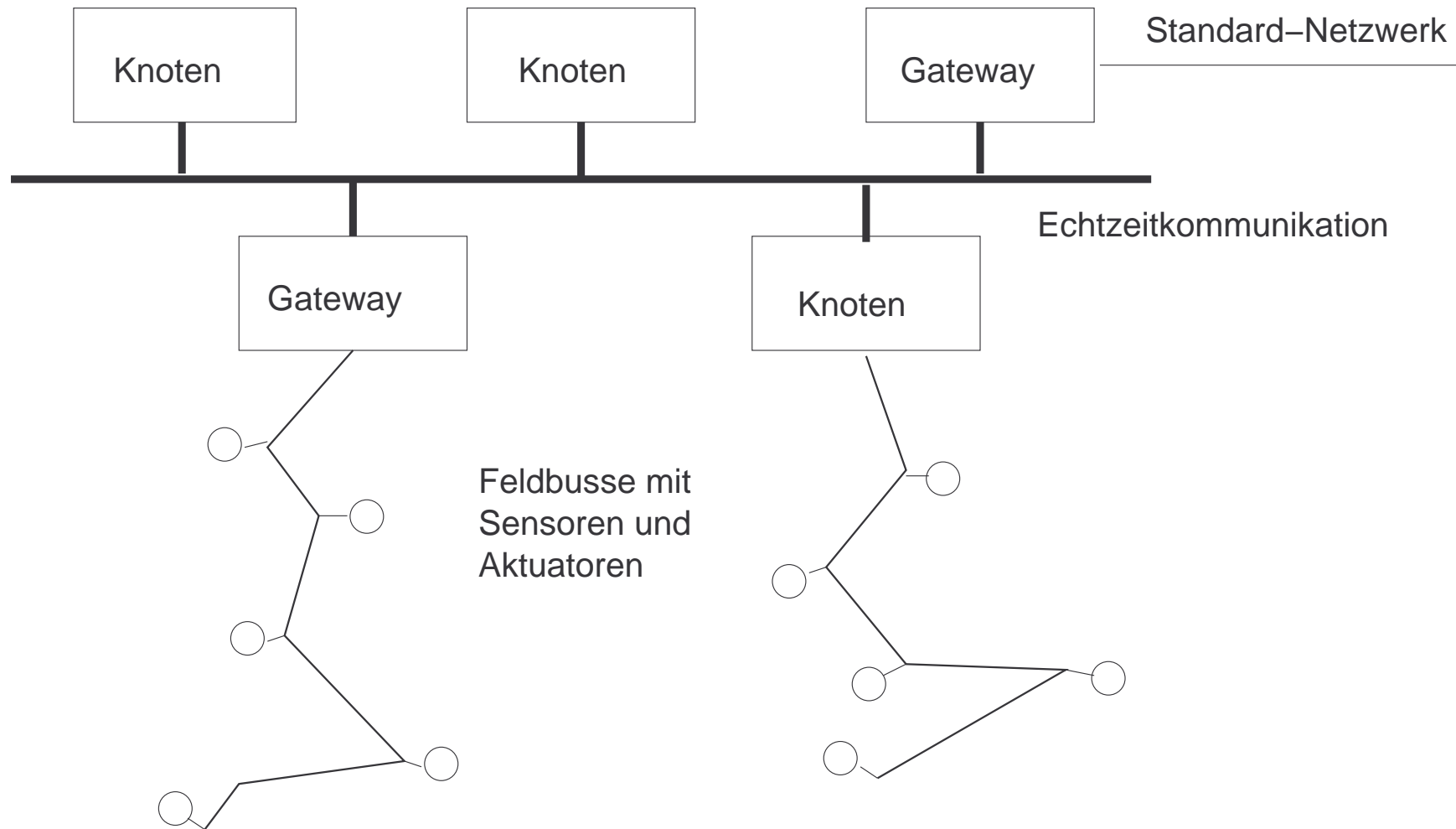




# Überblick

- Architektur eines Echtzeitsystems
- Was sind Feldbusse?
- Beispiel: CAN
- Beispiel: TTP/A
- Beispiel: ByteFlight

# Architektur eines Echtzeitsystems





# Feldbusse

- Anbindung von Sensoren und Aktuatoren an das Echtzeitsystem (oder einen Knoten eines verteilten Systems)
- In großen Systemen spezieller Gatewayknoten
- Architektur durch physikalische Begebenheiten beeinflusst
- Kurze Daten von und zu Sensoren
- Oft periodische Nutzung
- Hohe Datenrate selten erforderlich
- Fehlertoleranz nicht bedeutsam auf Feldbus-Ebene, da Sensoren/Aktuatoren Fehlerschwerpunkt sind
- Redundanz besser durch zweiten Feldbus mit zweitem Sensorsystem
- Kostenminimierung ist Ziel für Knoten und Kabel
- Oft ungeschirmte Zweidrahtleitung



# Beispiel: CAN

## Controller Area Network

- Entwickelt von Bosch Ende der 80er für Automobil-Industrie
- Benutzt in vielen europäischen Autos
  - VAG-Fahrzeuge (VW, Audi, Seat, Skoda)
  - Volvo, Saab
  - Ford
  - in-House-Standard bei GM
- Broadcast-Bus
- Datenrate: 1MBit/s (maximal)
- Kleine Nachrichten (0...8 Byte)
- Relativ hoher Overhead (47 Bits + Bitstuffing-Bits)

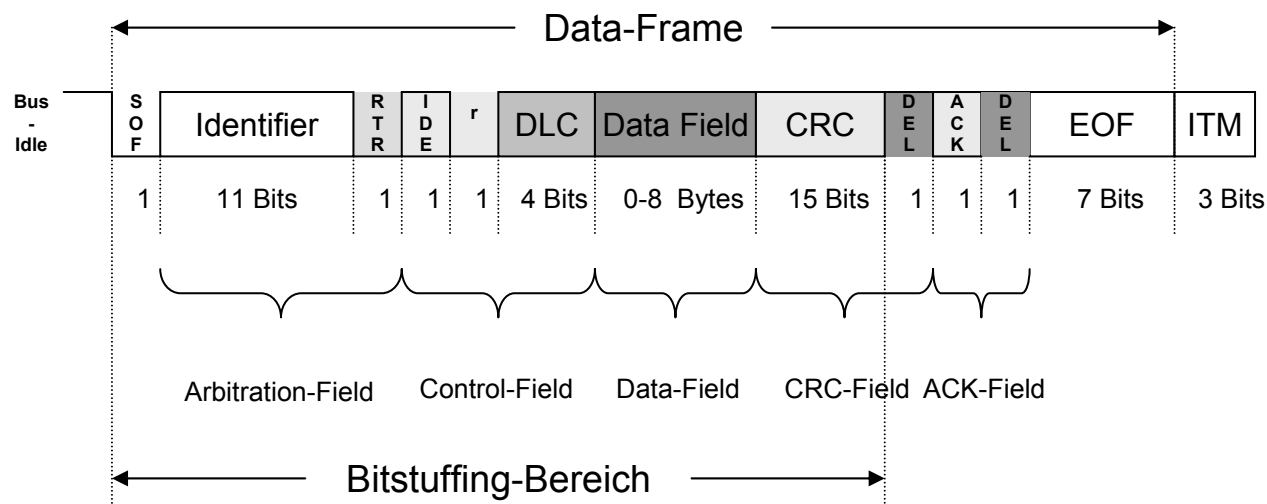


# CAN: Funktion

- CAN ist wie Ethernet:
  - Stationen mit Sendewunsch warten, bis der Bus frei ist, und senden dann.
  - Wenn eine Kollision festgestellt wird, wird Sendung abgebrochen und später wiederholt
- ... aber systematischer:
  - Bus hat elektrische Eigenschaften, die die Kollisionsbehandlung verbessern
  - Kollisionen werden nicht nur erkannt, sondern so vermieden, daß eine Station trotz Kollision senden kann

# CAN: Protokoll I

- Nachrichten werden “frames” genannt
- Nachrichten sind durch einen Nachrichtentyp gekennzeichnet
  - bedeutet Inhalt der Nachricht (benutzt für Adressierung)
  - Benutzt in der Arbitrierung (niedrigster Typ bedeutet höchste Priorität bei Konflikten)
- Physikalische Schicht von CAN verhält sich wie ein verdrahtetes AND (d.h. sobald einer Null sendet, empfangen alle Null)



# CAN: Protokoll II

- Alle Knoten empfangen alle Frames
- Spezieller CAN-Controller (Chip) realisiert CAN-Kommunikation für einen Knoten
- CAN-Controller filtert ankommende Nachrichten für Knoten anhand der Nachrichtentypen
- Zu sendende Nachrichten werden vom Controller nach Prioritäten sortiert gepuffert (lokale Behandlung der Prioritäten)
- Semantik der Kommunikation: Publisher/Subscriber



# CAN: Arbitrierung I

- Frames beginnen mit der Sendung des höchstwertigsten Bits des Nachrichtentyps
- Während der Sendung des Typs findet die Arbitrierung statt:
  - Andere Knoten dürfen auch senden
  - Ziel: Feststellung des Frames mit der höchsten Priorität (= niedrigster Typ)
- Wenn ein Knoten eine 1 (rezessives Bit) sendet, aber eine 0 (dominantes Bit) empfängt, wird Sendung abgebrochen (und später neu versucht bei Ruhe auf dem Bus)

# CAN: Arbitrierung II

**Frame 1 1344**

**1**

**Frame 2 1306**

**1**

**Frame 3 1498**

**1**

**Bus**

**1**

# CAN: Arbitrierung III

**Frame 1 1344**

**10**

**Frame 2 1306**

**10**

**Frame 3 1498**

**10**

**Bus**

**10**

# CAN: Arbitrierung IV

**Frame 1 1344**

**101**

**Frame 2 1306**

**101**

**Frame 3 1498**

**101**

**Bus**

**101**

# CAN: Arbitrierung V

Frame 1 1344

1010

Frame 2 1306

1010

~~Frame 3 1498~~

~~1011~~

Bus

1010

# CAN: Arbitrierung VI

~~Frame 1 1344~~

~~10101~~

Frame 2 1306

10100011010

~~Frame 3 1498~~

~~1011~~

Bus

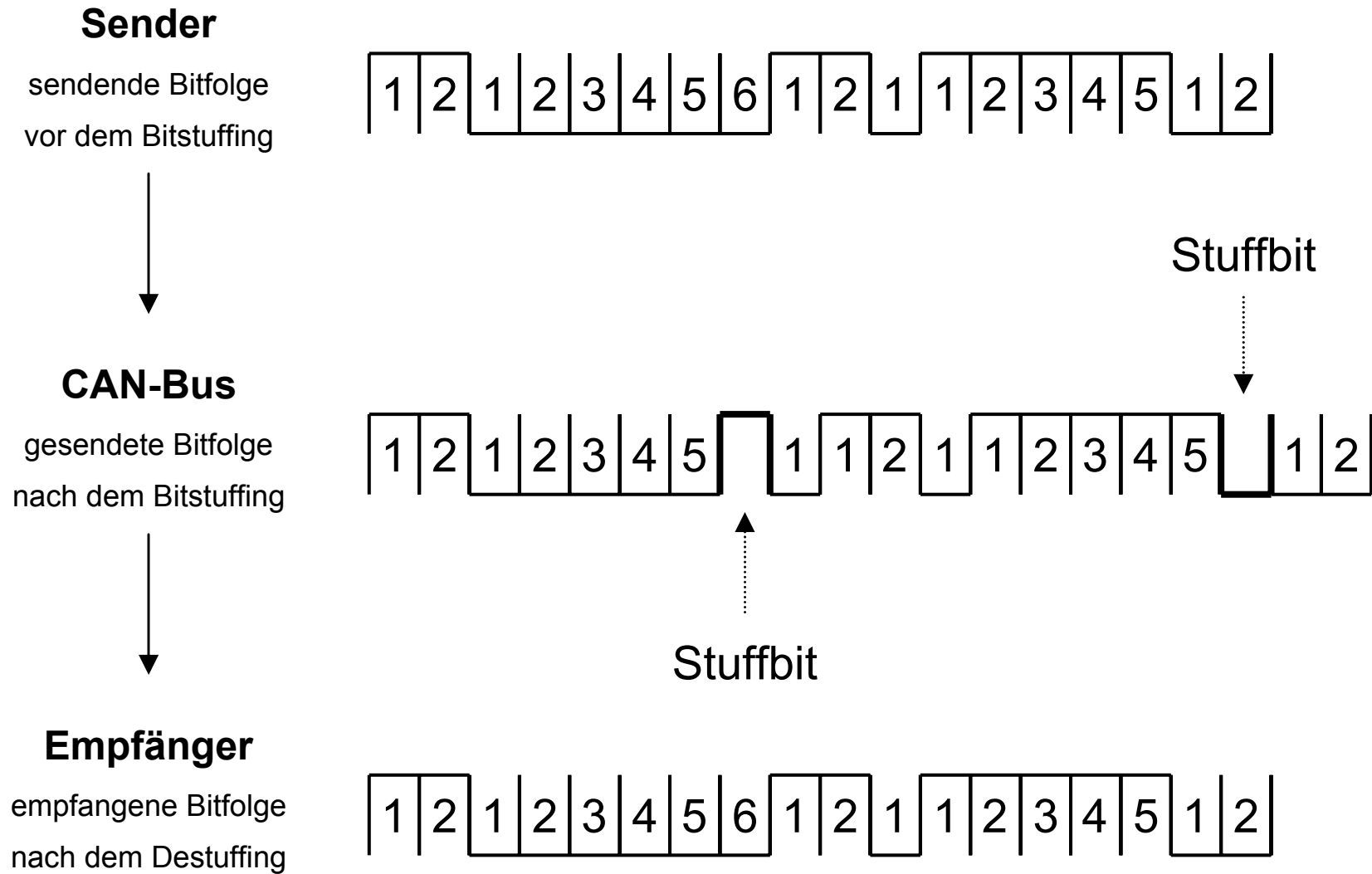
1306

10100011010

# CAN: Bitstuffing I

- Problem:
  - Stille auf dem Medium muß erkennbar sein (eine lange Folge von 1-Bits (elektrisch null) kann als Stille fehlinterpretiert werden)
  - Lange Folgen gleicher Bits erschweren Synchronisation der Stationen, da es keine Bitflanken gibt
  - Mehr als 5 mal 0 in Folge wird als Errorframe benutzt - Unterscheidung ist notwendig
- Lösung:
  - Bitstuffing
  - Nach 5 gleichen Bits fügt das Protokoll ein Bit des anderen Wertes ein
  - Dekodierung analog

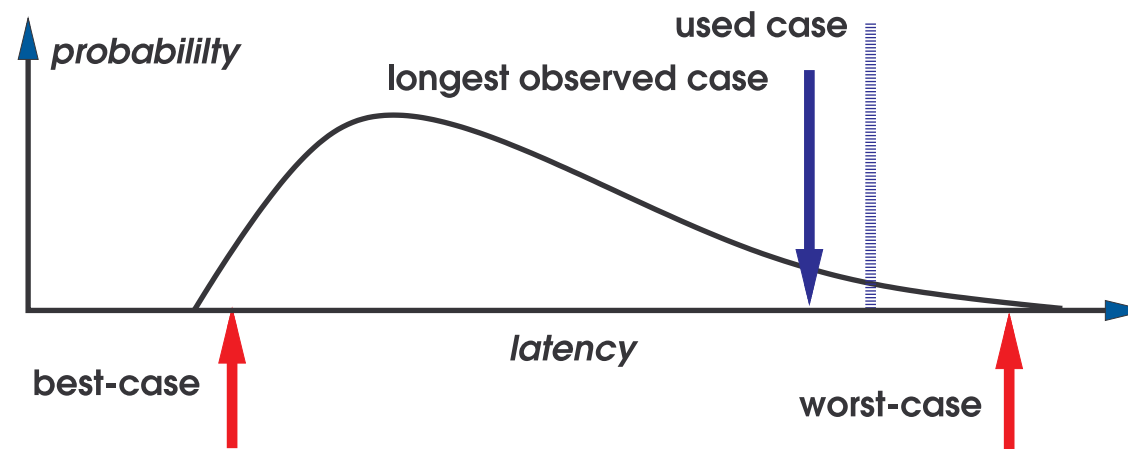
# CAN: Bitstuffing II





# CAN: Harte Echtzeit

- Erforderlich: Worst Case Zeiten für Frame Latency (end-to-end-delay)
- Durch Testen:



- Durch Analyse: Anwendung der Fixed Priority Scheduling Theorie
  - Bus = geteilte Ressource (entspricht CPU im klassischen Scheduling)
  - Frame = Job (entspricht Task-Invokation)

# Fixed Priority Scheduling

## Periodische Tasks

Berechnung der Response-Time:

$$R_i = C_i + B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

- $C_i$  — Ausführungszeit einer Task
- $B_i$  — Blocking Time (Zeit, die die Task durch niederpriorere Tasks blockiert werden kann)
- $\sum \dots$  — Zeit, die die Task durch höherpriorere Tasks unterbrochen werden kann

# CAN: Scheduling

- Berechnung von Worst Case Frame Latencies
- Deadlines werden relativ zur Zeit der Erzeugung eines Frames gemessen
- Periode ist der kürzeste Abstand zwischen der Erzeugung zweier Frames gleichen Typs
- Typ des Frames entspricht der Priorität
- Unterschied zu “normalem” Scheduling: Übertragung von Frames ist nonpreemptiv

# CAN: Längste Übertragungszeiten

- Übertragungszeit ist abhängig von der Größe des Frames (0..8 Datenbytes)
- Framegröße: 47 Bit Header + 8 Datenbytes + Stuffbits

$$C_i = \left( 47 + 8s_i + \left\lfloor \frac{8s_i + 34}{4} \right\rfloor \right) T_{bit}$$

- $T_{bit}$  — Zeit zum Übertragen eines Bits ( $1\mu s$  bei 1MBit/s)
- $s_i$  — Anzahl der Datenbytes

# CAN: Längste Wartezeit (Queuing Time)

Wie lange dauert es, bis ein Frame die Arbitrierung gewinnen kann?

Frame wird “behindert” durch:

- Gerade übertragenen Frame mit niedrigerer Priorität (Blockierung, da Senden nonpreemptiv ist)
- Alle Frames mit höherer Priorität (Preemption im Sinne des Scheduling-Modells)

# CAN: Blocking Time

- $B_i$  ist die Zeit, die der größtmögliche Frame niedrigerer Priorität zur Übertragung benötigt
- Nie mehr als 135 Bitzeiten

$$B_i = \max_{\forall k \in lp(i)} (C_k) \leq 135T_{bit}$$

# CAN: Längste Queuing Time und Latency

- Längste Queuing Time

$$q_i = B_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{q_j + T_{bit}}{T_j} \right\rceil C_j$$

$q_i$  wird durch Iteration berechnet

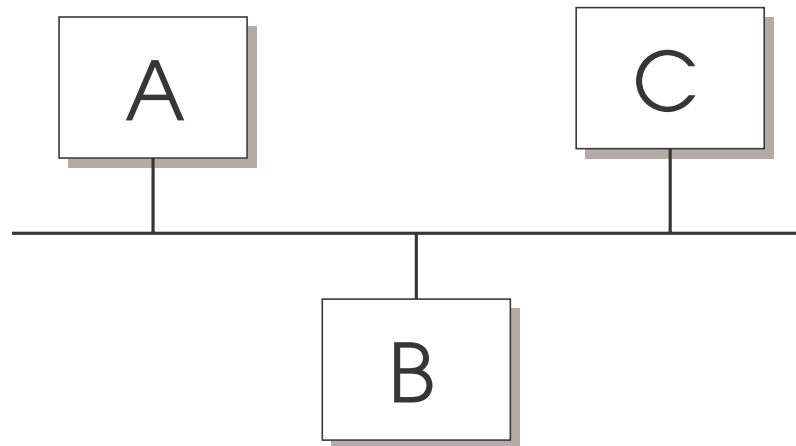
- Latency

$$R_i = q_i + C_i$$



# CAN: Beispiel I

Frame	ID	Period	Size	WCET
A	572	9.0 ms	8	1.350ms
B	347	5.0 ms	2	0.750ms
C	115	2.5 ms	8	1.350ms





## CAN: Beispiel II

$$q_B = C_A + \left\lceil \frac{q_B + T_{bit}}{T_C} \right\rceil C_C$$

$$q_B^0 = 0$$

$$q_B^1 = 1.35 + \left\lceil \frac{0 + T_{bit}}{2.5} \right\rceil 1.35 = 2.7$$

$$q_B^2 = 1.35 + \left\lceil \frac{2.7 + T_{bit}}{2.5} \right\rceil 1.35 = 4.05$$

$$q_B^3 = 1.35 + \left\lceil \frac{4.05 + T_{bit}}{2.5} \right\rceil 1.35 = 4.05$$

$$R_B = q_B + C_B = 4.05 + 0.75 = 4.8$$

# CAN: Fehlererkennung

## Erkennbare Fehler:

- Bit Error  
Empfangsteil liest anderes Bit als Sendeteil gesendet hat (Ausnahme: Arbitrierung oder Ack-Slot)
- Stuff Error  
Auftreten von 6 gleichen Bits in einem Datenfeld, in dem Bitstuffing benutzt wird
- CRC Error  
Berechneter CRC-Wert weicht von empfangenen Wert ab
- Form Error  
Ein Feld fester Form enthält ungültige Bits
- Acknowledgment Error  
Nachricht wurde nicht mit einem dominanten Bit im ACK-Slot bestätigt

# CAN: Fehlerbehandlung

- Erkennung eines Fehlers führt zum Senden eines Fehlerflags
- Wenn Station *error active* ist, dann aktives Errorflag (6 dominante Bits), sonst passives Errorflag (6 rezessive Bits)
- Aktives Errorflag führt zu Fehler in ALLEN Stationen
- Andere Stationen senden ebenfalls Errorflag
- Nach Errorflags startet Retransmission

# CAN: Fault Confinement I

- Problem: Fehlerhafte Station kann durch häufiges Senden von aktiven Errorflags den Bus stören
- Lösung: Einführung von drei Zuständen
  - *error active*  
Nimmt normal am Busverkehr teil
  - *error passive*  
Darf keine aktiven Errorflags senden, muß vor Senden von Nachrichten warten
  - *bus off*  
Darf keinen elektrischen Einfluß auf den Bus nehmen
- Gesteuert durch zwei Zähler:
  - Transmit error count (TEC)
  - Receive error count (REC)

# CAN: Fault Confinement II

Regeln (vereinfacht, Ausnahmen weggelassen):

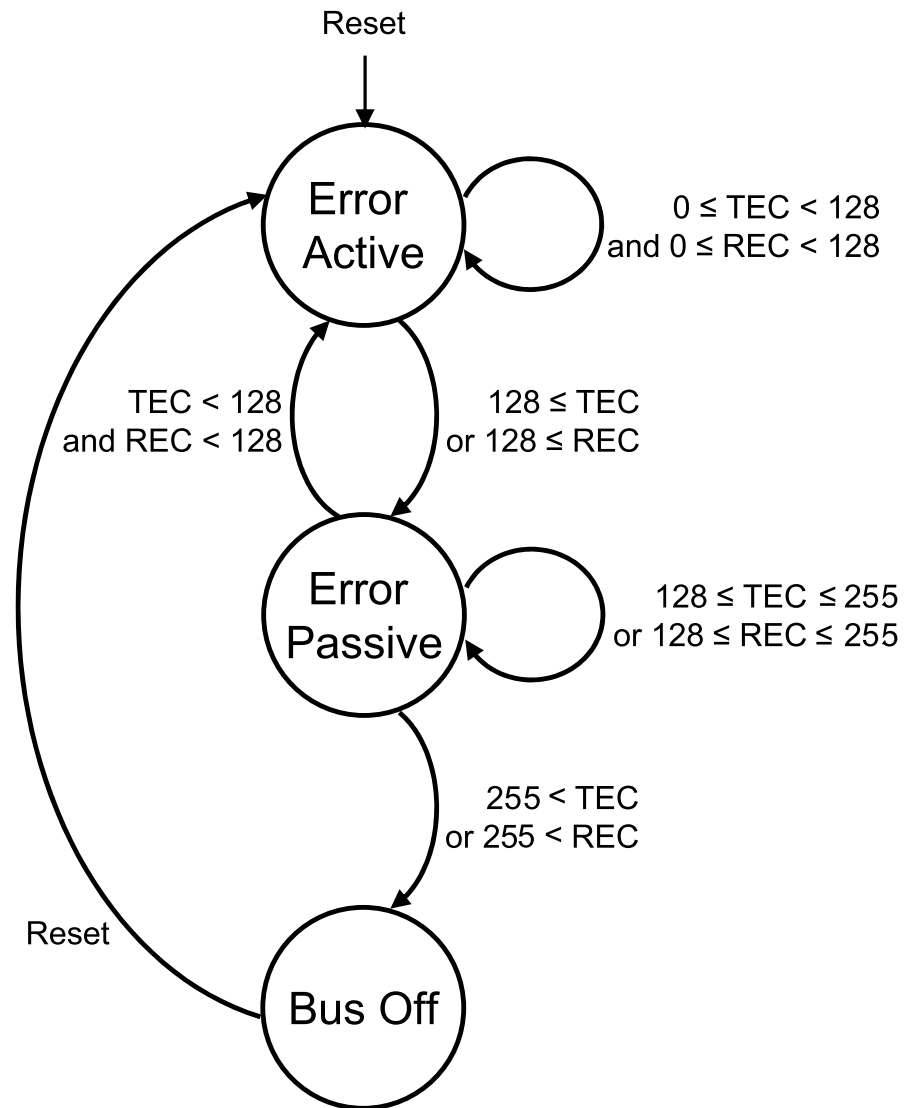
- Empfänger erkennt Fehler:  $REC=REC+1$
- Empfänger liest dominantes Bit NACH dem Errorflag:  $REC=REC+8$  (das betrifft die verursachende Station)
- Sender sendet Errorflag:  $TEC=TEC+8$  (Ausnahmen während Arbitrierung und ACK)
- Sender erkennt Bitfehler während Sendung eines aktiven Errorflags:  $TEC=TEC+8$
- Empfänger erkennt Bitfehler während Sendung eines aktiven Errorflags:  $REC=REC+8$
- Nach dem 14. dominanten Bit (8. nach einem Errorflag):  $TEC=TEC+8$  bzw.  $REC=REC+8$
- Nach Senden einer Nachricht und positiver Bestätigung:  $TEC=TEC-1$  (wenn nicht schon Null)

# CAN: Fault Confinement III

## Regeln (Fortsetzung):

- Nach erfolgreichem Empfang einer Nachricht:  $REC = REC - 1$ , falls  $0 < REC < 128$ , und  $REC = 120 \dots 126$ , falls  $REC > 127$
- Wenn  $REC > 127$  oder  $TEC > 127$ : Station wird *error passive*, Senden eines aktiven Errorflags beim Wechsel
- $TEC > 255$ : *bus off*
- *error passive* Stationen werden wieder aktiv, wenn  $TEC$  und  $REC$  unter 128 fallen
- *bus off* Stationen werden *error active* nach 128 Auftreten von 11 aufeinanderfolgenden rezessiven Bits

# CAN: Fault Confinement IV



# CAN: Fault Confinement V

## Anmerkungen:

- Fehlerzähler größer 96 weisen auf einen stark gestörten Bus hin
- Startup:
  - Einzelner Knoten sendet, aber es gibt keine Empfänger
  - Kein positives ACK
  - Knoten kann *error passive* werden, aber nicht *bus off*





# Beispiel TTP/A

## TTP: Time Triggered Protocol

- Grundidee: Benutzung von TDMA (Time Division Multiple Access)
- Zugriff auf das Medium in statisch zugewiesenen Zeitschlitz
- Sehr viel Vorabwissen
- Protokollvarianten
  - TTP/C: vollständiges Protokoll, Teil der TTA (Time Triggered Architecture) — wird behandelt bei Architekturen
  - TTP/A: abgerüstete Version für Feldbusse

# TTP/A: Protokoll

- 1 Byte Nachrichten (Statusnachrichten)
- Rundenbasierte Arbeitsweise
- Zentraler Master (Gateway zum Backbone)
- Für jede Runde ist die Folge der Nachrichten statisch in der MEDL (Message Descriptor List) festgelegt
- Master sendet zu Beginn “Firework-Byte” mit der Nummer der für diese Runde aktuellen MEDL, ungerade Parität
- Danach werden zu den in der MEDL definierten Zeiten die Nachrichten mit gerader Parität OHNE jeden Overhead gesendet (reine Daten)
- Firework-Byte synchronisiert Knoten
- Hardware: Mikrocontroller mit UART (Universal Asynchronous Receiver Transceiver)
- Bandbreite 10 KBit/s (auch 48 und 100)

# TTP/A: Fehlerbehandlung

- Zeit-Domäne
  - Jeder Verstoß gegen die MEDL führt zum Setzen eines Fehlerflags (beispielsweise Nachricht zur falschen Zeit)
  - Sehr kurze Fehlerpropagationszeit
  - Fehler führen zum Ende der Runde für betroffenen Knoten, Warten auf neues Firework-Byte
  - Master kann nach Timeout durch Backup-Master ersetzt werden
- Werte-Domäne
  - Protokoll sieht nur Paritätschecks vor
  - Weitere Mechanismen in den UARTs (z.B. Noise-Reduction)
  - Fehlererkennung darüber hinaus obliegt Anwendungslogik (wird vom Controller über erkannte Fehler informiert)
- Weitere Details:
  - Vorlesung über Architekturen in Zusammenhang mit TTA

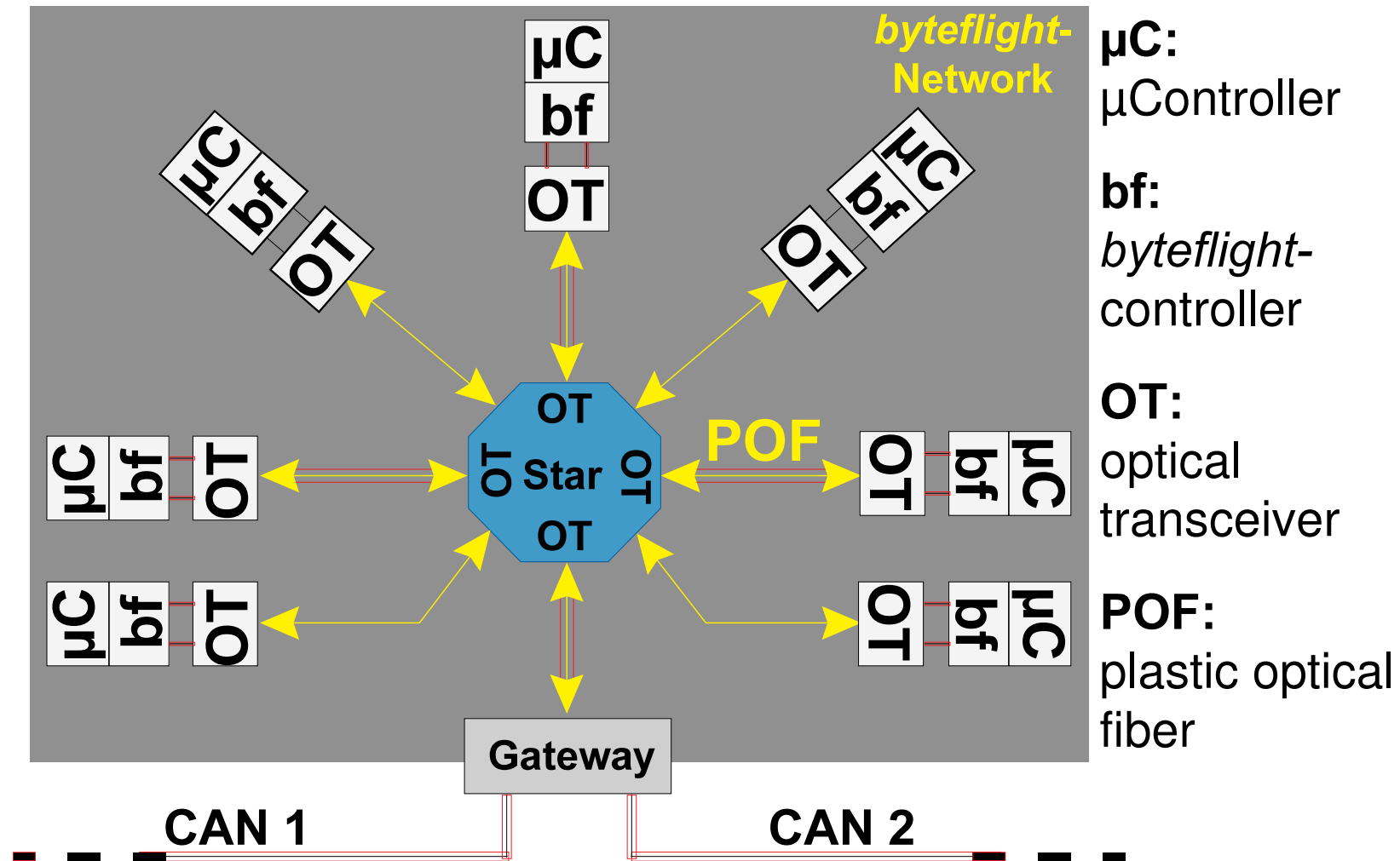
# Beispiel: ByteFlight

- Idee und Konzept: BMW
- Spezifikation: BMW und Motorola
- Entwicklung ab 1996
- Partner:
  - BMW Motorola, Elmos, Infineon, Siemens EC
  - Weise GmbH
  - Steinbeis Transferzentrum Prozessautomatisierung
  - TÜV Bayern, TÜV Rheinland
- Mittlerweile in Zusammenarbeit mit anderen Autoherstellern in Flex-Ray eingeflossen (Vereinigung u.a. mit TTP-Ideen)

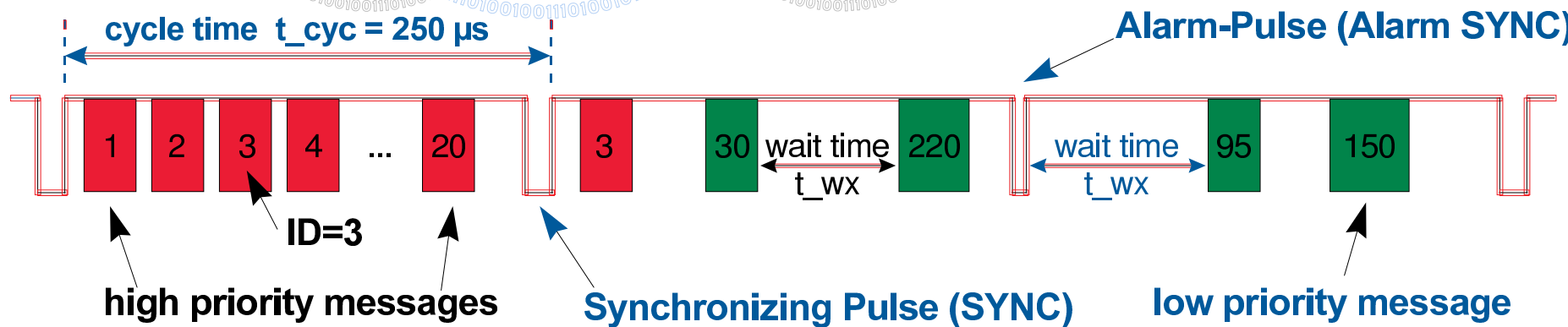
# Byteflight: Ideen

- Zentraler Master generiert Sync-Impulse
- Zwischen Sync-Impulsen können Nachrichten gesendet werden
- Zugriff im Flexible Time Division Multiple Access (FTDMA)
  - Ähnlich Minislotting (ARINC 620)
  - Wartezeit indirekt proportional Priorität
  - Garantierte Übertragung für eine bestimmte Anzahl (z.B. 10) hoch-priorer Nachrichten
- Physikalisch: optischer Sternkoppler (logischer Bus)
- Strategien für Fehlerbehandlung und Vermeidung von Babbling Idiot

# Byteflight: Systemüberblick

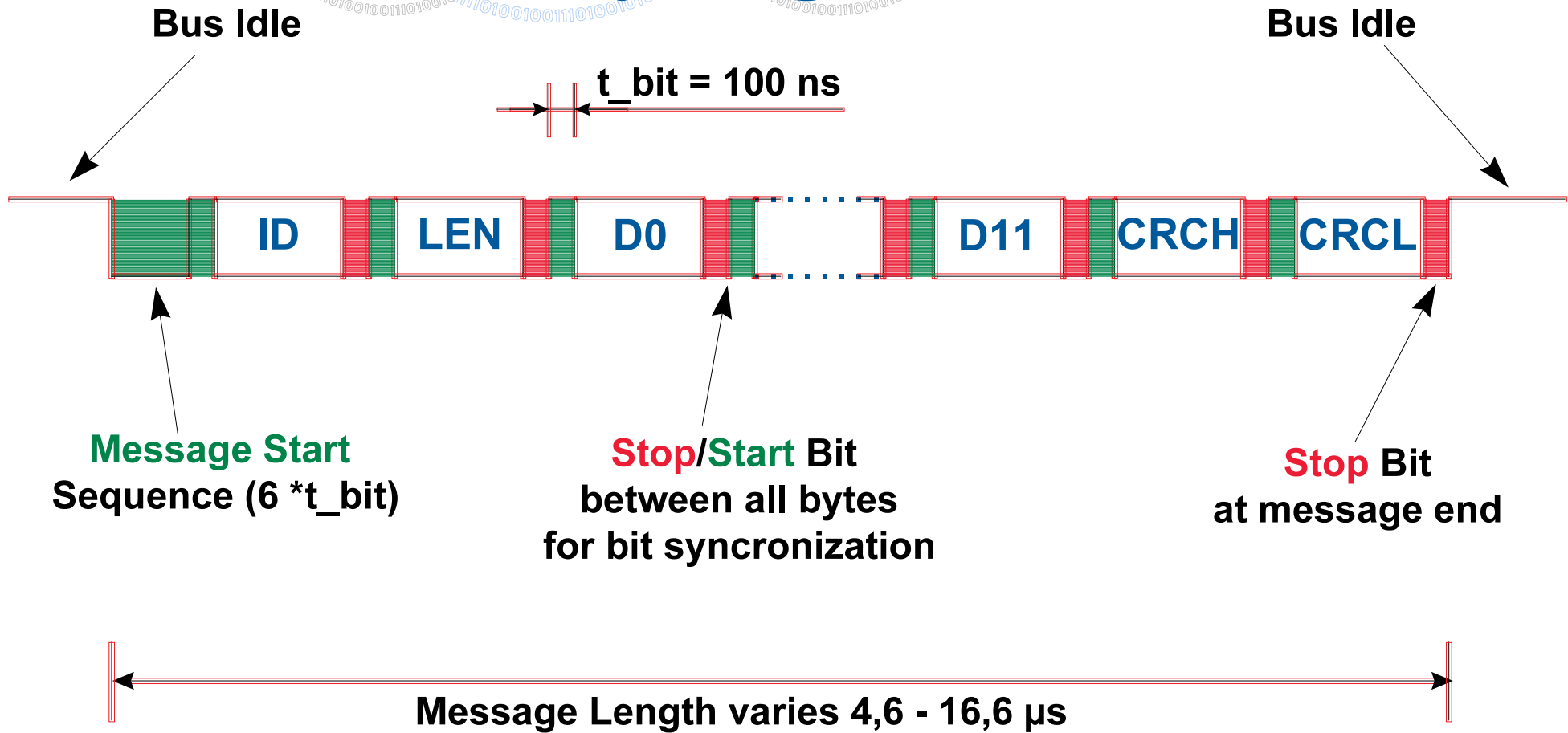


# Byteflight: Protokoll



- Buszugriff
  - Logisches Token wird von ID zu ID weitergegeben nach Wartezeit  $t_{wx}$
  - Wartezeit ist abhängig von der Priorität der Nachricht
- Kombination von Vorteilen synchroner und asynchroner Protokolle
  - Garantierte Latenz für definierte Anzahl hochpriorer Nachrichten
  - Flexibler Zugriff für alle übrigen Nachrichten (asynchron)
  - Separater logischer Kanal für Alarmmeldungen

# Byteflight: Nachrichtenformat

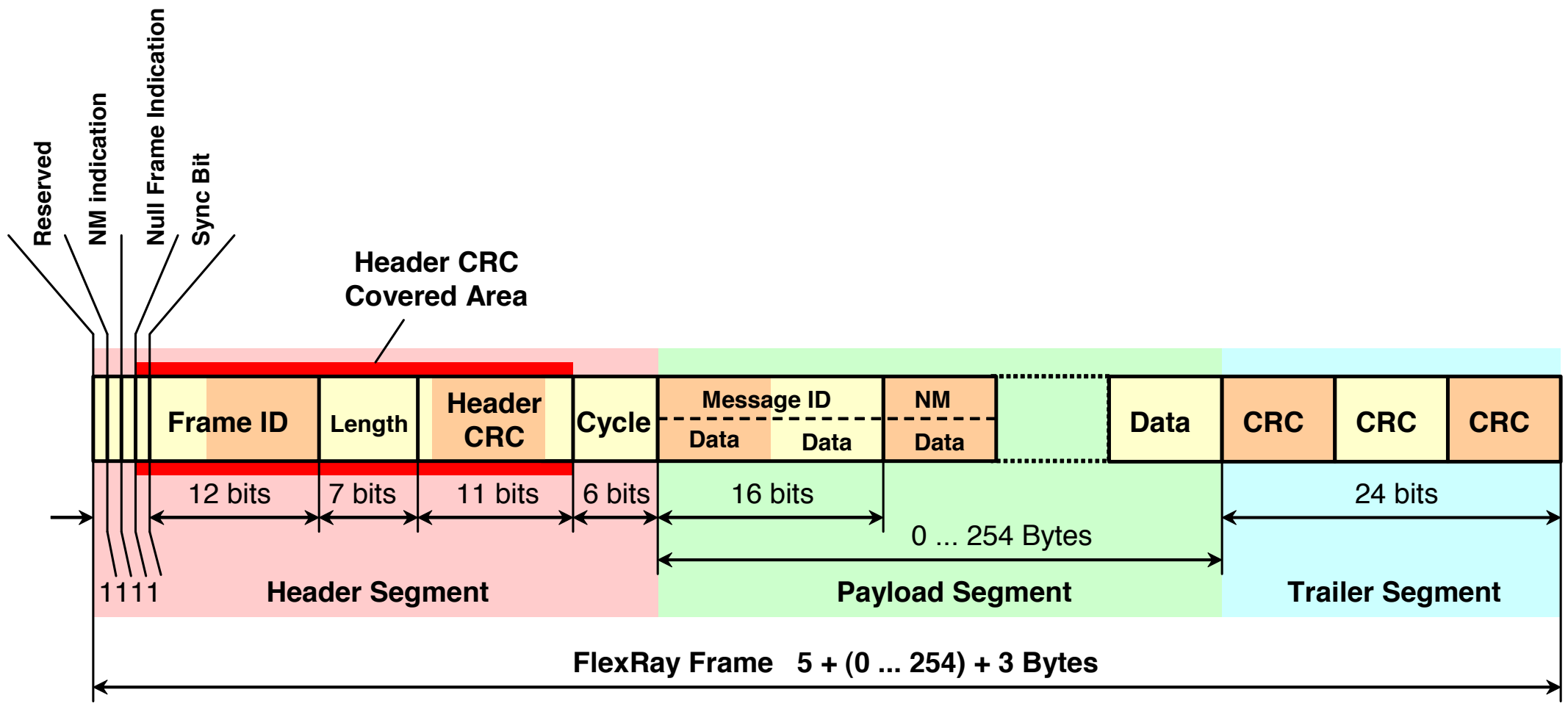




- FlexRay-Konsortium 1999 von BMW, DaimlerChrysler, Motorola und Philips gegründet.
- (Redundante) Bus-, Stern- oder Multi-Stern-Topologie
- Design-Ziel: Flexibilität
- Kombination aus TTP/C und ByteFlight
- Fehlertolerante Uhrensynchronisation (ähnlich TTP)
- Entkopplung des Kommunikationskontrollers (ähnlich TTP)
- Kommunikationszyklus besteht aus zwei Teilen:
  - Statischer Teil: TDMA
  - Dynamischer Teil: ByteFlight FTDMA



# Flexray: Frame



# Flexray: Kommunikationszyklus

