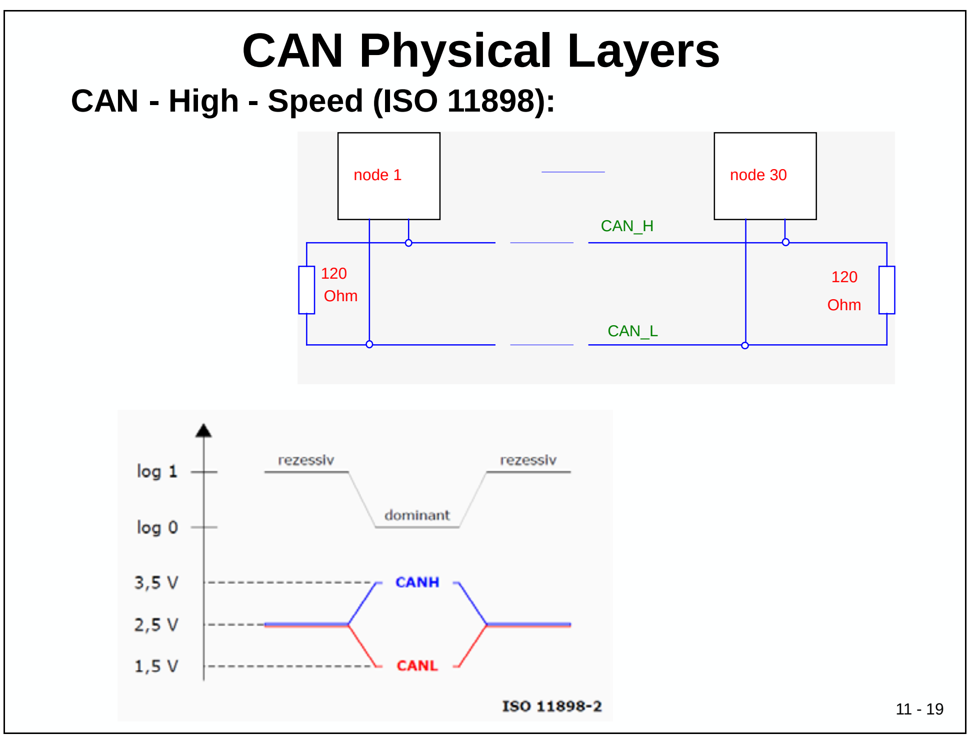
F2833x Controller Area Network

Introduction

One of the most successful stories of the developments in automotive electronics in the last decade of the 20th century has been the introduction of distributed electronic control units in passenger cars. Customer demands, the dramatic decline in costs of electronic devices and the amazing increase in the computing power of microcontrollers has led to more and more electronic applications in a car. Consequently, there is a strong need for all those devices to communicate with each other, to share information or to co-ordinate their interactions.

The “Controller Area Network” was introduced and patented by Robert Bosch GmbH, Germany. After short and heavy competition, CAN was accepted by almost all manufacturers. Nowadays, it is the basic network system in nearly all automotive manufacturers’ shiny new cars. Latest products use CAN accompanied by other network systems such as LIN (a low-cost serial net for body electronics), MOST (used for in-car entertainment) or Flexray (used for safety critical communication) to tailor the different needs for communication with dedicated net structures.

Because CAN has high and reliable data rates, built-in failure detection and cost-effective prices for controllers, nowadays it is also widely used outside automotive electronics. It is a standard for industrial applications such as a “Field Bus” used in process control. A large number of distributed control systems for mechanical devices use CAN as their “backbone”.



Module Topics

Basic CAN Features

CAN is a serial communication network, the information is transmitted over 1 (“fault tolerant low speed”) or 2 (“high speed” differential) physical signal lines. Although there is no explicit clock information in form of an additional clock line, the receivers are able to re-synchronize themselves based on a “non return to zero” (NRZ) modulation technique and an additional “stuff” bit rule, which forces the transmitter to include a stuff bit after 5 consecutive bits of ‘0’ or ‘1’.

CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions - it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received by CAN modules and determines the priority of the message when two or more nodes want to transmit at the same time.

Controller Area Network (CAN)

- **developed by Robert Bosch GmbH, Germany in 1987**
- **Products available from all microcontroller manufacturers**
- **International Standards: ISO11898 (Europe), SAE J2284 (US) for “high – speed” CAN; ISO 11519-2 for “fault-tolerant low speed” CAN**
- **backbone serial bus system for automotive applications, but also used in industrial automation & control**
- **Event triggered Serial Bus System; Self-Synchronisation**

More Features :

- **multi master bus access**
- **random access with collision avoidance (CSMA / CA)**
- **short message length , at max. 8 Bytes per message**
- **data rates 100KBPS to 1MBPS**
- **short bus length, physical length depends on data rate**
- **self-synchronised bit coding technology**
- **Robust EMC - behaviour**
- **build in fault tolerance**

11 - 2

The bus access procedure is a multi-master principle, all nodes are allowed to use CAN as a master node. One of the basic differences to Ethernet is the adoption of non-destructive bus arbitration in case of collisions, called “Carrier Sense Multiple Access with Collision Avoidance“(CSMA/CA). This procedure ensures that in case of an access conflict, the message with higher priority will not be delayed by this collision.

The physical length of the CAN is limited, depending on the baud rate. The data frame consists of a few bytes only (maximum 8), which increases the ability of the net to respond to new transmit requests. On the other hand, this feature makes CAN unsuitable for very high data throughputs, for example, for real time video processing.

There are several physical implementations of CAN, such as differential twisted pair (automotive class: CAN high speed), single line (automotive class: CAN low speed) or fibre optic CAN, for use in harsh environments.

Automotive Network Systems

Electronic Control Units

Examples for Microcontrollers used in car:

Antilock Break System - ABS (1 + 4)
Keyless Entry System(1)
Active Wheel Drive Control (4)
Engine Control (2)
Airbag Sensor Systems (6+)
Automatic Gearbox(1)

Seat occupation sensors(4)
Electronic Park Brake(1)
diagnostic computer(1)
driver display unit(1)
air conditioning system(1)
adaptive cruise control(1)
radio / CD-player(2)
collision warning radar(2)
rain/ice/snow sensor systems (1)

dynamic drive control(4)
active damping system (4)
driver information system(1)
GPS navigation system(3)

11 - 3

Today a car is packed with electronic devices, sensors, actuators and control units. To name a few, Slide 11-3 shows some of the functional blocks and the number of microcontrollers in brackets. There is a lot of information to be shared by such electronic control units: a network is required.

Why a car network like CAN?

➔ Requirements of an in car network:

- low cost solution
- good and high performance with few overhead transmission
- high volume production
- high reliability and electromagnetic compatibility (EMC)
- data security due to a fail-safe data transmission protocol
- short message length, only a few bytes per message

➔Where in a car is CAN used?

- communication between electronic control units
- separated CAN – sections at different speed for:
  - "Auto - Body" electronic control units (chassis, light, central locking)
  - Engine control units and Power train modules
  - Comfort modules

11 - 4

As you can guess, there are some options to implement a communication network into a car. Depending on the application field, the bandwidth for data throughput, the safety level and the budget limitation, we can find different communication standards:

- Controller Area Network (CAN)
    - High - speed CAN (1 Mbit/s, 500 kbit/s)
    - Low - Speed CAN (100 kbit/s, 83.3 kbit/s)
- Local Interconnect Network (LIN)
    - 20 kbit/s
- Media Oriented Systems Transport (MOST)
    - 25Mbit/s, 50 Mbit/s, 150 Mbit/s
- FlexRay®
    - 10 Mbit/s

Automotive network systems

➔ Other automotive networks than CAN:

- LIN – "Local Interconnect Network"
    - Body Electronic; Door, Mirror, Seat, Dashboard, Roof
    - 20 Kbit/s
    - Master / Slave time triggered protocol
    - Single wire system; 12 V signal level
    - www.lin-subbus.org
- MOST – "Media Oriented Systems Transport"
    - Optical System for Multi – Media and infotainment
    - Audio, Video, Mobile Phone, GPS
    - Fibre optical circular system at 25 Mbit/s or 150 Mbit/s or
    - Electrical layer at 50 Mbit/s.
    - www.mostcooperation.com
- FlexRay
    - Time Triggered Protocol for fail safe applications;
    - 10 Mbit/s; dual channel redundancy
    - www.flexray.com

11 - 5

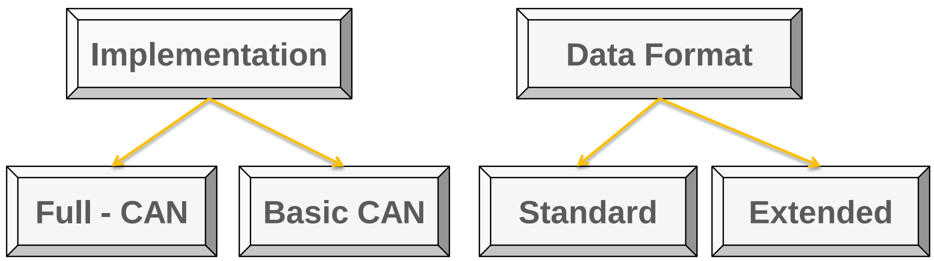
CAN Implementation / Data Format

Implementation / Classification of CAN

Implementation: amount of functionality in CAN- Silicon

Don't get confused !

Communication is standardized and identical for all implementations of CAN. However, there are two types of hardware implementation and two versions of data format:



11 - 6

There are two versions of how the CAN-module is implemented in silicon, called "Basic" and "Full" - CAN. Almost all new processors with a built-in CAN module offer both modes of operation. Basic-CAN as the only mode is normally used in cost sensitive applications.

Basic- and Full-CAN communication

Basic CAN

- **Close coupled MCU-core and CAN**
- **only one transmit buffer**
- **only two receive buffer**
- **only one filter for incoming messages**
- **Software routines are needed to select between incoming messages**

Full - CAN

- **provide a message server**
- **extensive acceptance filtering on incoming messages**
- **user configurable mailboxes**
- **mailbox memory area , size of mailbox areas depends on manufacturer**
- **advanced error recognition**

11 - 7

CAN Data Frame

The Data Format of CAN

Standard

- **CAN-Version 2.0A**
- **messages with 11-bit - identifiers**

Extended

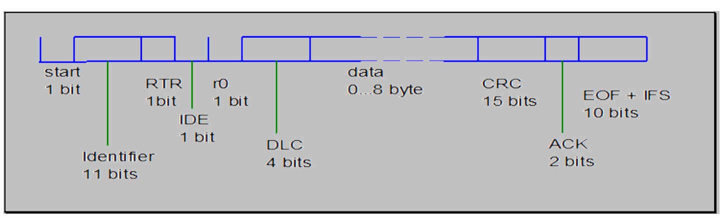
- **CAN-Version 2.0B**
- **messages with 29-bit-identifiers**

==> **Suitably configured, each implementation (BASIC or FULL) can handle both standard and extended data formats.**

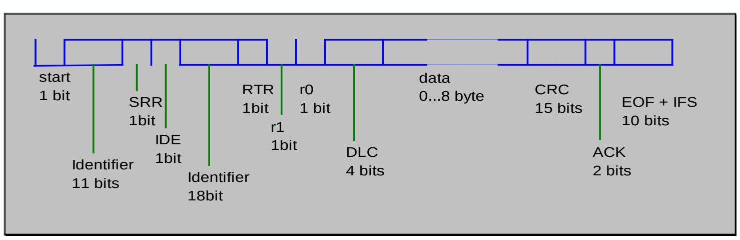
11 - 8

The two versions of the data frame format allow the reception and transmission of standard frames and extended frames in a mixed physical set up; provided the silicon is able to handle both types simultaneously (CAN version 2.0A and 2.0B respectively).

The CAN Data Frame



DATA-Frame CAN 2.0A (11-bit-identifier)



DATA-Frame CAN 2.0B (29-bit-identifier)

11 - 9

The CAN Data Frame

each data frame consists of four segments :

(1) arbitration-field :
- **denote the priority of the message**
- **logical address of the message (identifier)**
- **Standard frame, CAN 2.0A: 11 bit-identifier**
- **Extended frame, CAN 2.0B: 29 bit-identifier**

(2) data field :
- **up to 8 bytes per message ,**
- **a 0 byte message is also permitted**

(3) CRC field:
- **cyclic redundancy check ; contains a checksum generated by a CRC-polynomial**

(4) end of frame field:
- **contains acknowledgement, error-messages, end of message**

11 - 10

The arbitration field is used to denote both the priority and the type of the message. CAN uses a broadcast type of transmission, there are no node addresses. Instead of node addresses, CAN implements logical groups of message identifiers. The next slide explains all bit fields of a CAN data frame in detail.

The CAN Data Frame

start bit	**(1 bit - dominant): beginning of a message; after idle-time falling-edge to synchronize all transmitters**
identifier	**(11 bit): mark the name of the message and its priority ;the lower the value the higher the priority**
RTR	**(1 bit): remote transmission request; if RTR=1 (recessive) no valid data inside the frame - it is a request for receivers to send their messages**
IDE	**(1 bit): Identifier Extension; if IDE=1 then extended CAN-frame**
r0	**(1 bit): reserved**
CDL	**(4 bit): data length code in byte (0...8)**
data	**(0...8 byte): the data of the message**
CRC	**(15 bit): cyclic redundancy code for error detection, no correction; hamming-distance 6 (up to 6 single bit errors can be detected)**
ACK	**(2 bit): acknowledge; if a receiving node has received a valid message, it must transmit an dominant acknowledge – bit**
EOF	**(7 bit = 1, recessive): end of frame; intentional violation of the bit-stuff-rule ; normally after five recessive bits one stuff-bit follows automatically**
IFS	**(3 bit = 1, recessive): inter frame space; time space to copy a received message from bus-handler into buffer**

Extended Frame only :

SRR	**(1 bit = recessive): substitute remote request ; substitution of the RTR-bit in standard frames**
r1	**(1 bit): reserved**

11 - 11

Standardization ISO and SAE

The Standardisation of CAN

- CAN is an open system and has been standardized by ISO
- CAN follows the ISO - OSI seven layer model for open system interconnections
- CAN implements layer 1, 2 and 7 only
- However, Layer 7 is not standardised

Physical Layer Type	Europe www.iso.org	North America www.sae.org
Single – Wire CAN	n/a	SAE J2411 Single Wire CAN for Vehicle Applications
Low-Speed Fault Tolerant CAN	ISO 11519 - 2 ISO 11898 - 3	n/a.
High-Speed CAN	ISO 11898	SAEJ2284

11 - 12

As an open system, CAN today is standardized both by the European Standardization Organization (ISO) and the Society of Automotive Engineers (SAE). All CAN standards define layer 1 and 2 of the OSI - layer model only. For layer 7 some higher layer solutions exist.

ISO Reference Model

Open Systems Interconnection (OSI):

Layer 7 Application Layer	
Layer 6 Presentation Layer	void
Layer 5 Session Layer	void
Layer 4 Transport Layer	void
Layer 3 Network Layer	void
Layer 2 Data LInk Layer	
Layer 1 Physical Layer	

Layer 1: transmission line(s)
- differential two-wire-line, twisted pair with/without shield
- Transceiver Integrated Circuit
- Optional: fibre optical lines (passive coupled star, carbon)
- Optional: Coding as PWM, NRZ, Manchester Code
- ISO 11898

Layer 2: Data Link Layer
- message format and transmission protocol
- ISO 11898
- CSMA/CA access protocol

Layer 7: Application Layer
- different standards in industry, not standardized in automotive

11 - 13

CAN Application Layer

CAN Layer 7

1. CAN Application Layer (CAL):
   - European CAN user group "CAN in Automation (CiA)"
   - originated by Philips Medical Systems 1993
   - CiA DS-201 to DS-207
   - standardised communication objects, -services and -protocols (CAN-based Message Specification)
   - Services and protocols for dynamic attachment of identifiers (DBT)
   - Services and protocols for initialise, configure and obtain the net (NMT)
   - Services and protocols for parametric set-up of layer 2 &1 (LMT)
   - Automation, medicine, traffic-industry

2. OSEK/VDX
   - "**O**ffene **S**ysteme für **E**lektronik im **K**raftfahrzeug"
   - Standard of European automotive electronics industry
   - include services of a standardised real-time-operating system
   - Network Management Services
   - Communication Services

11 - 14

For OSI - layer 7, some user groups have defined specific layers, such as CAL, CANOpen or DeviceNet, which are tailored to certain application areas. These layers are not compatible with each other. In automotive applications, layer 7 is usually a proprietary (and confidential) in - house solution.

CAN Layer 7

3. CANopen
   - European Community funded project "ESPRIT"
   - 1995 : CANopen profile :CiA DS-301
   - 1996 : CANopen device profile for I/O : CiA DS-401
   - 1997 : CANopen drive profile
   - industrial control , numeric control in Europe

4. DeviceNet
   - Allen-Bradley, now ODVA-group (www. odva.org)
   - device profiles for drives, sensors and actuators
   - master-slave communication as well as peer to peer
   - industrial control , mostly USA

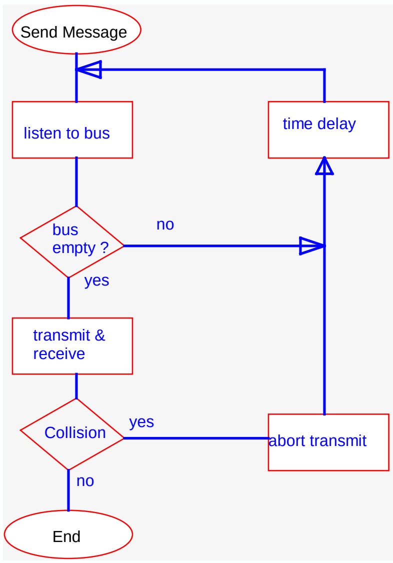
5. Smart Distributed Systems (SDS)
   - Honeywell , device profiles
   - only 4 communication functions , less hardware resources
   - industrial control and PC-based control

11 - 15

CAN Bus Arbitration - CSMA/CA

Bus Access Procedure

The "Ethernet": CSMA / CD



CSMA /CD:
Carrier
Sense
Multiple
Access with
Collision
Detection

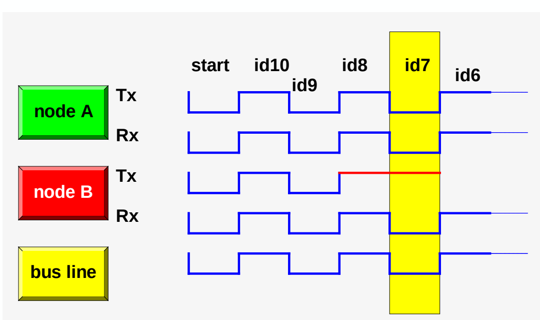
Note: This flowchart does NOT apply to CAN! See following page

11 - 16

CAN feature a modified CSMA/CD access control principle, where a message with the highest priority will continue its transmission regardless of the collision with other messages. Therefore the modification is called "collision avoidance" (/CA), sometimes "collision resolution" (/CR).

CAN Access Procedure: CSMA/CA

CSMA/ CA: "**C**arrier **S**ense **M**ultiple **A**ccess with **C**ollision **A**voidance"



- **access-control with non destructive bit-wide arbitration**
- **if there is a collision , the "winner" continues**
- **the message with higher priority is not delayed!**
- **real-time capability for high prioritised messages**
- **the lower the identifier, the higher the priority**

11 - 17

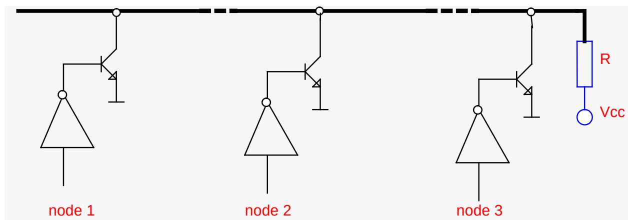
CSMA/CA (cont.)

CSMA / CA =

"bit - wide arbitration during transmission with simultaneous receiving and comparing of the transmitted message"

means :

- **if there is a collision within the arbitration-field, only the nodes with lower priorities cancel transmission.**
- **The node with the highest priority continues with the transmission of the message.**



high : reccessive

low : dominant

node 1	node 2	node 3	bus
high	high	high	high
high	low	high	low
low	low	high	low

11 - 18

As you can see from the previous slide the arbitration procedure at a physical level is quite simple: it is a “wired-AND” principle. Only if all 3 node voltages (node 1, node2 or node3) are equal to 1 (recessive), the bus voltage stays at V_{cc} (recessive). If only one node voltage is switched to 0 (dominant), the bus voltage is forced to the dominant state (0).

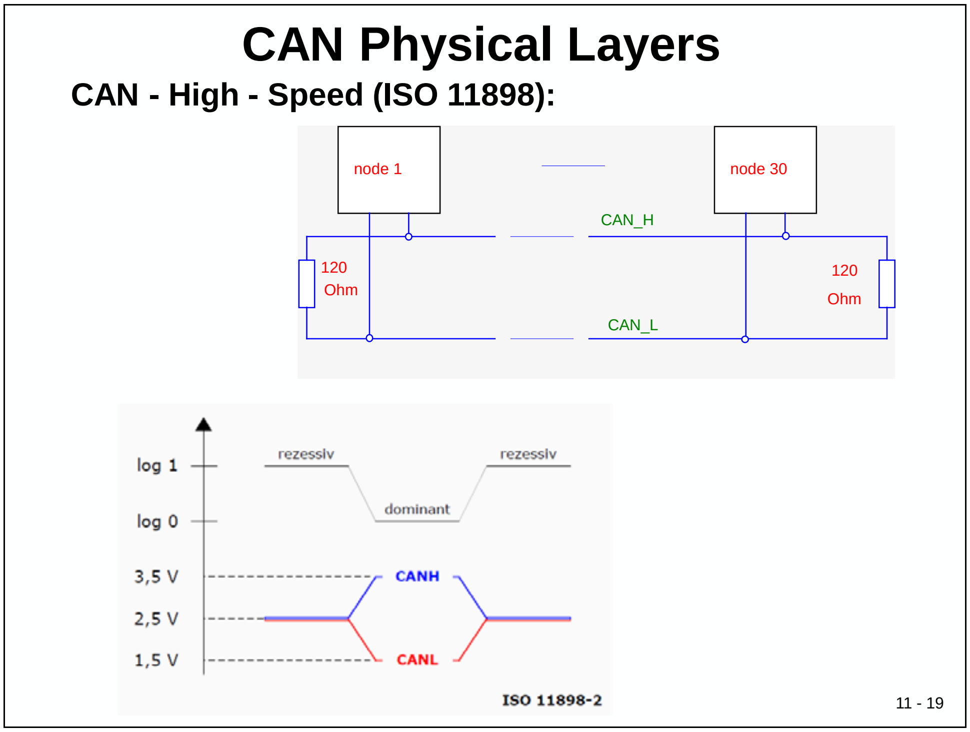
The beauty of CAN is that the message with highest priority is not delayed at all in case of a collision. For the message with highest priority, we can determine the worst-case response time for a data transmission. For messages with lower priorities, to calculate the worst-case response time is a little bit more complex task. It could be done by applying a so-called “time dilatation formula for non-interruptible systems”:

$$\boxed{R_i^{n+1} = C_i + B_{\max i} + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n - C_i}{T_j} \right\rceil * C_j}$$

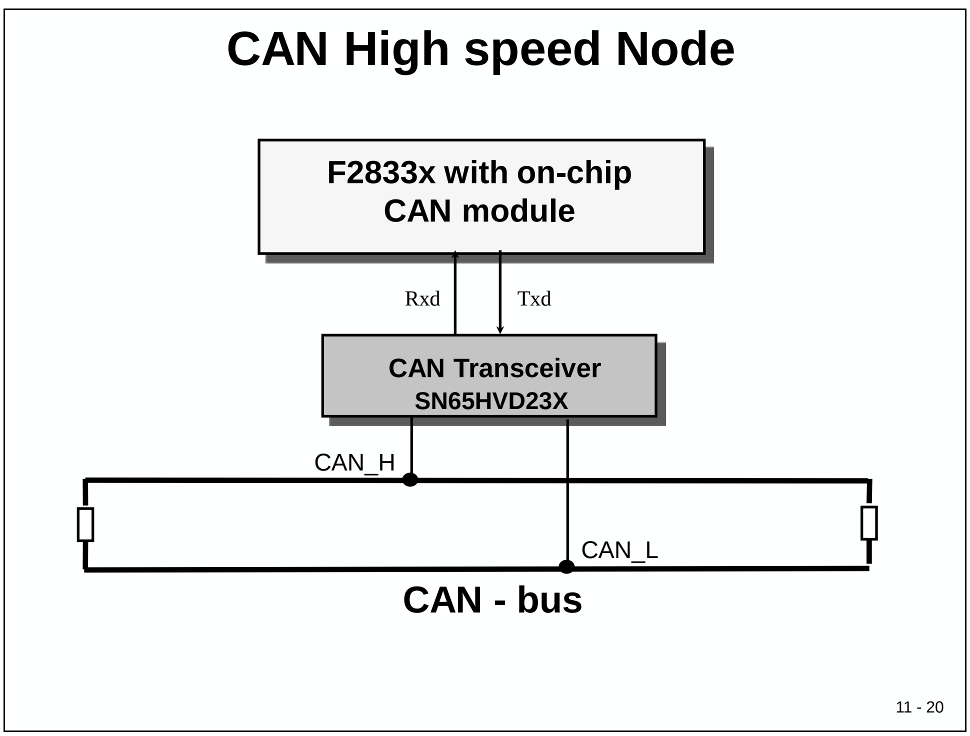
HARTER, P.K: “Response Times in level structured systems” Techn. Report, Univ. of Colorado, 1991

In detail, the hardware structure of a CAN-transceiver is more complex. Due to the principle of CAN-transmissions as a “broadcast” type of data communication, all CAN-modules are forced to “listen” to the bus all the time. This also includes the arbitration phase of a data frame. It is very likely that a CAN-module might lose the arbitration procedure. In this case, it is necessary for this particular module to switch into receive mode immediately. This requires every transceiver to provide the current bus voltage status permanently to the CAN-module.

High Speed CAN



To generate the voltage levels for the differential voltage transmission according to CAN High Speed, we need an additional transceiver device, e.g. the SN65HVD23x.



CAN Error Frames

Layer 2 of CAN also includes an enhanced strategy to detect transmission errors, which is based on error -levels and the exchange of error messages. Please note that the exchange of error messages is managed by the CAN communication controller in OSI layer 2; it is therefore totally independent of application layer 7.

CAN Error – Frame

- **any node that detects a bus error generates an error - frame**
- **an error frame is transmitted as soon as an error has been detected, e.g. inside a data frame**
- **consists of two fields: Error Flag Field; Error Delimiter Field**

- **Error Delimiter Field:**
  - **8 recessive bits**
  - **allow bus nodes to restart bus communication after an error**
- **Error Flag Field:**
  Type depends on the error-state of the node:
  - **error active: 6 consecutive dominant error bits; all other nodes will respond to this violation with their own error frames ➔ Error Flag Field = 6…12 dominant bits**
  - **error passive: 6 consecutive recessive bits plus 8 error delimiter bits = 14 recessive bits**
    - **receiver: does not corrupt the message**
    - **transmitter: other nodes may respond with active error frames**

11 - 21

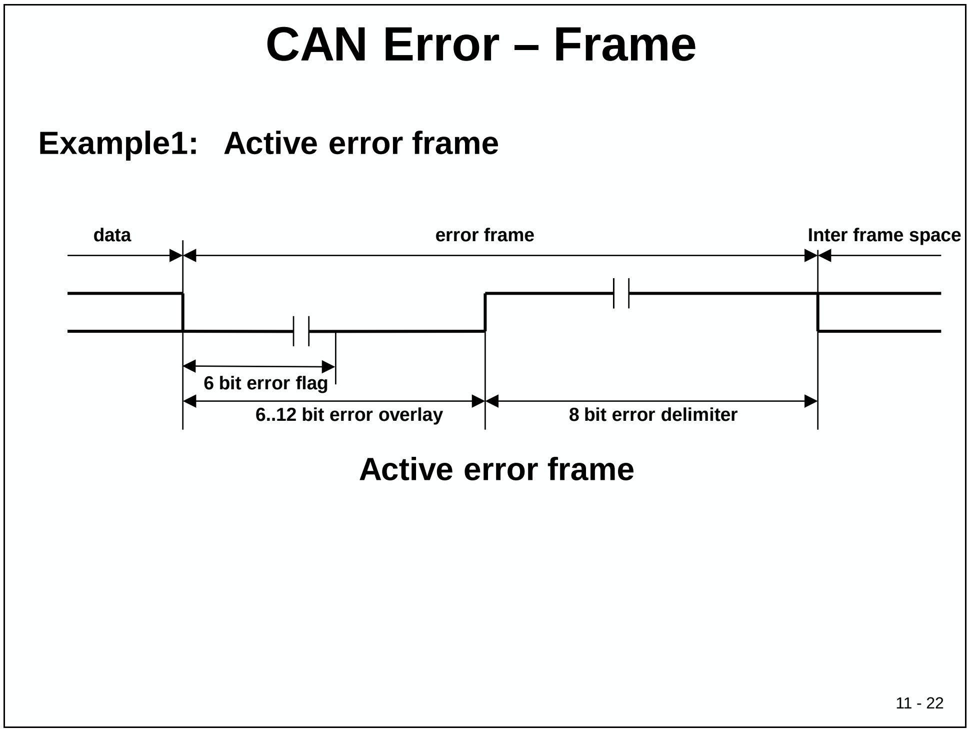
The error management of a node is based on one of 3 states, in which a node operates:

- Error Active State
- Error Passive State
- Bus OFF state

Depending on the state a node is able to transmit "Active Error" - frames, "Passive Error" - frames or no error frames at all.

The objective behind these 3 levels is to have the ability to identify a potential fault node, to isolate this node and to keep the remaining part of the bus running. This principle will be explained shortly. For now, let us concentrate on the characteristics of the different error frames.

Active Error Frame



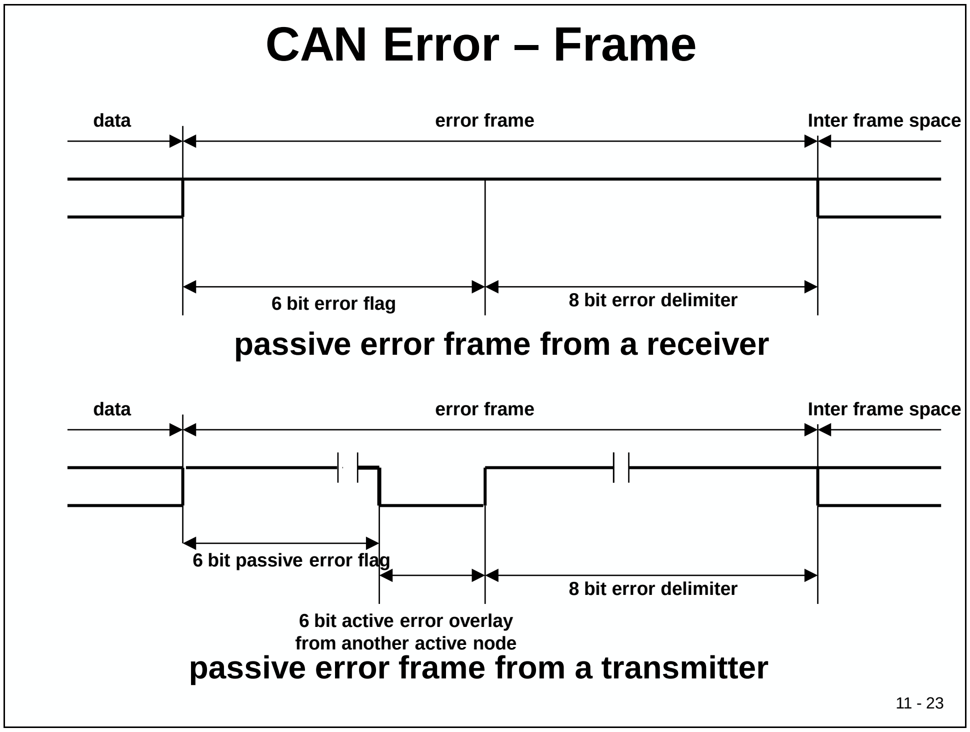
The first example in Slide 11-22 shows the timing diagram of an active error frame. As soon as a node detects faulty data, it will send such a frame to the bus. Since the error flag field contains 6 zero bits, which is (an intended) violation of the stuff bit rule, other nodes will respond with their own active error frames. Depending on how many bits of the last data group have been 0, the other nodes will start sooner or later with the transmission of their follow-up active error frames, leading to a 6...12 bit error overlay as shown in Slide 11-22.

If a receiving node receives an active error frame, it will mark the data contents of this message as faulty and cancel it. The message will not be forwarded to the mailbox server and to the application. Instead, the receiver mailbox will be cleared to be able to await a re-transmission of the message.

If a transmitting node receives an active error frame, it will immediately stop the current transmission. As soon as the bus is empty, it will try to re-transmit the message. As long as no successful transmission has happened, the application will not get the “Transmission Acknowledged” (TA) status flag.

Passive Error Frame

If a node has reached "error passive" level, is no longer able to generate active error frames. Instead, it will issue passive error frames in case of a detected data corruption.



Slide 11-23 shows what happen, if a node is in error passive mode.

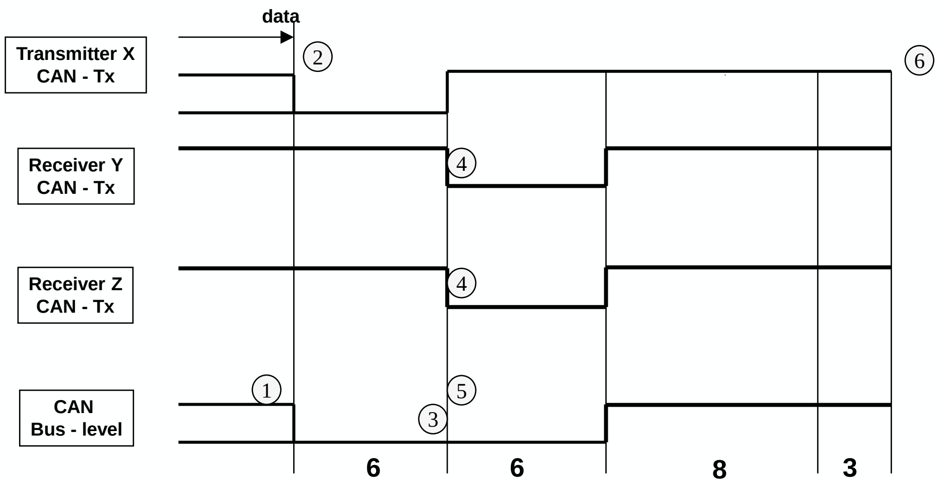
If a receiver spots faulty data, it will issue a passive error frame. The 6 recessive error bits can now be overwritten by dominant bits of the original transmitter data, which is still in active mode.

If a transmitter is in passive error mode and generates a passive error frame, this (intended) violation will be answered by receivers in error active mode with a 6 bit active error overlay, shown in the bottom half of Slide 11-23. Since the original transmitting node is the only transmitter at that time, the active error overlay ensures that all nodes will cancel the corrupted message, which has already been detected by the transmitter.

Using these two principles, it ensures that nodes in error active mode will always be able to overrule nodes in error passive state. Only if all nodes of a CAN subnet are in error passive mode, the recessive level of error passive frames from receivers will be treated as error messages.

The next slides will illustrate what happens in case of an error in a more realistic scenario.

CAN Error – Frame



Example: active error frame

11 - 24

The bullets 1 to 6 indicate events on the time line. At position 5, node X tries to generate 6 recessive bits for the error delimiter but the actual bus level is dominated by node Y and Z and their delayed active error frames. The time delay between bus and the Tx - line of node X is used to define the node, which has first spotted the error.

CAN Error – Frame

1. Node X detects a bit error
2. Node X generates an active error flag field
3. Nodes Y, Z realize a stuff bit error after bit 6 of the active error flag field (note: if the corrupted data frame had dominant bits, the stuff bit error is detected earlier)
4. Nodes Y,Z transmit their own active error flag field of 6 dominant bits
5. All nodes transmit the recessive error delimiter field. Node Y and Z see no difference @ bus level, but node X detects a delay of 6 bits between bus level and its own output ➔ First node to message error
6. After the last 8 recessive error delimiter bits @ CAN-bus and 3 bit of inter frame space a new arbitration is entered by node X, e.g. it has to compete again with other nodes

11 - 25

CAN Error Types

CAN Error Recognition

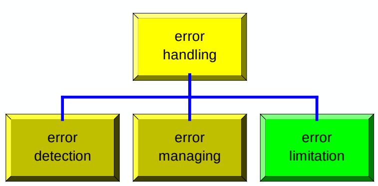
1. **Bit-Error**
   the transmitted bit doesn't read back with the same digital level (except arbitration and acknowledge- slot)
2. **Bit-Stuff-Error**
   more than 5 continuous bits read back with the same digital level (except 'end of frame'-part of the message)
3. **CRC-Error**
   the received CRC-sum doesn't match with the calculated sum
4. **Format-Error**
   Violation of the data-format of the message , e.g.: CRC-delimiter is not recessive or violation of the 'end -of-frame'-field
5. **Acknowledgement-Error**
   transmitter receives no dominant bit during the acknowledgement slot, i.e. the message was not received by another node.

11 - 26

CAN Error Status

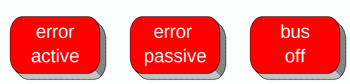
Here is a summary for the node's error states:

CAN Error Status



Purpose: avoid persistent disturbances of the CAN by switching off defective nodes

three Error States :



Error Active: normal mode, messages will be received and transmitted. In case of error an active error frame will be transmitted.

Error Passive: after detection of a certain number of errors, the node reaches this state. Messages will be received and transmitted but in case of an error the node sends a passive error frame.

Bus Off: the node is separated from CAN, neither transmission nor receive of messages is allowed and the node is no longer able to transmit error frames.

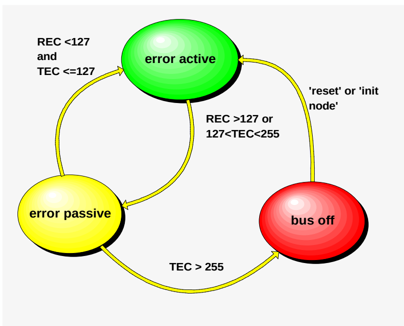
11 - 27

CAN - Error Counter

The transitions between error states of a node is based on the current value in two error counters, called Receive Error Counter (REC) and Transmit Error Counter (TEC).

CAN Error Counter

State - Diagram:



- **transitions will be carried out automatically by the CAN-chip**
- **states are managed by 2 Error Counters :**
  Receive Error Counter (REC)
  Transmit Error Counter (TEC)
- **Possible situations :**

a) a transmitter recognises an error:
TEC:=TEC + 8
b) a receiver sees an error : REC:=REC + 1
c) a receiver sees an error, after transmitting an error frame: REC:=REC + 8
d) if an 'error active'-node find's a bit-stuff-error during transmission of an error frame:
TEC:=TEC+ 1
e) successful transmission:
TEC:=TEC - 1
f) successful receive:
REC:=REC - 1

11 - 28

The current values both of REC and TEC are permanently available in two registers of the F2833x CAN Controller. For maintenance purposes it is a good idea to read the values from time to time to monitor the quality of the data transmission. Rising numbers in TEC and/or REC give an indication that something is going wrong with the communication and that this may be an appropriate time to take preventative action, e.g. switch into a local operating mode of the device.

The state diagram above shows the transitions between error active, error passive and bus off states. Successful communication is always represented by the number -1. Depending on the seriousness of a failure, the penalty is either +8 or +1 of the corresponding error counter.

After a RESET, the node is in error active mode. If REC or TEC is increased beyond 127, the node goes into error passive state. From this state the node can (a) go back to error active, if both REC and TEC are decreased below 127; or (b) will be forced into bus OFF state, if TEC is greater than 255.

The original CAN specification did not allow a recovery from bus OFF. The only option was to reset and re-initialize the device. This was really bad news as it meant that your car would lose full CAN communication and could grind to a halt.

However, newer microcontrollers, such as the F2833x, allow an automatic recovery, if a certain amount of idle time was applied to the bus. This additional feature can be enabled or disabled during the initialization of the CAN communication controller.

F2833x CAN Module

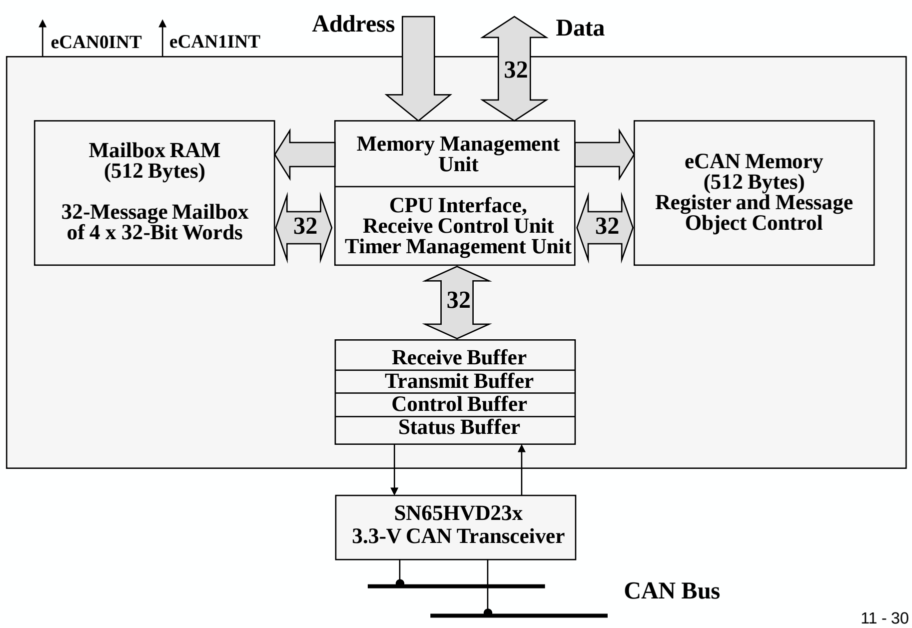
F2833x CAN Features

- Fully CAN protocol compliant, version 2.0B
- Supports data rates up to 1 Mbps
- Thirty-two mailboxes
  - Configurable as receive or transmit
  - Configurable with standard or extended identifier
  - Programmable receive mask
  - Supports data and remote frame
  - Composed of 0 to 8 bytes of data
  - Uses 32-bit time stamp on messages
  - Programmable interrupt scheme (two levels)
  - Programmable alarm time-out
- Programmable wake-up on bus activity
- Self-test mode

11 - 29

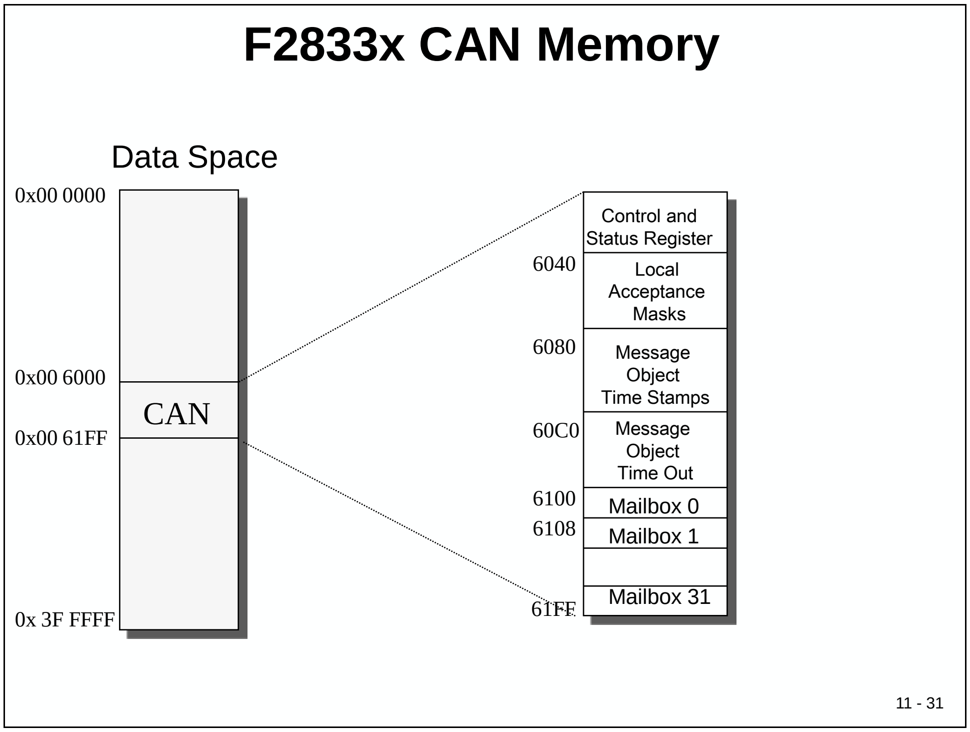
The F2833x CAN unit is a full CAN Controller. It contains a message handler for transmission, reception management and frame storage. The specification is CAN 2.0B Active - that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).

F2833x CAN Block Diagram



11 - 30

F2833x Programming Interface



The CAN controller module contains 32 mailboxes for objects of 0- to 8-byte data lengths:

- configurable transmit/receive mailboxes
- configurable with standard or extended identifier

The CAN module mailboxes comprise of the following components:

- MID - contains the identifier of the mailbox
- MCF (Message Control Field) - contains the length of the message (to transmit or receive) and the RTR bit (Remote Transmission Request - used to send remote frames)
- MDL and MDH - contain the data

The CAN module contains registers, which are divided into five groups. These registers are located in data memory from 0x006000 to 0x0061FF. The five register groups are:

- Control and Status Registers
- Local Acceptance Masks
- Message Object Time Stamps
- Message Object Timeout
- Mailboxes

It is the responsibility of the programmer to go through all those registers and set every single bit according to the designated operating mode of the CAN module. It is also a challenge for the student to exercise the skills required to debug. So let us start!

First, we will discuss the different CAN registers. If this chapter becomes too tedious, ask your teacher for some practical examples how to use the various options. Be patient!

CAN Register Map

CAN Control & Status Register

Address	31 … 0	Address	31 … 0
6000	CANME	6020	CANGIM
6002	CANMD	6022	CANGIF1
6004	CANTRS	6024	CANMIM
6006	CANTRR	6026	CANMIL
6008	CANTA	6028	CANOPC
600A	CANAA	602A	CANTIOC
600C	CANRMP	602C	CANRIOC
600E	CANRML	602E	CANLNT
6010	CANRFP	6030	CANTOC
6012	CANGAM	6032	CANTOS
6014	CANMC	6034	reserved
6016	CANBTC	6036	reserved
6018	CANES	6038	reserved
601A	CANTEC	603A	reserved
601C	CANREC	603C	reserved
601E	CANGIF0	603E	reserved

11 - 32

Mailbox Enable – CANME Mailbox Direction - CANMD

CAN Mailbox Enable Register (CANME) – 0x006000

31 … 16
CANME[31:16]

15 … 0
CANME[15:0]

Mailbox Enable Bits
0 = corresponding mailbox is disabled
1 = The corresponding mailbox is enabled. A mailbox must be disabled before writing to the contents of any mailbox identifier field.

CAN Mailbox Direction Register (CANMD) – 0x006002

31 … 16
CANMD[31:16]

15 … 0
CANMD[15:0]

Mailbox Direction Bits
0 = corresponding mailbox is defined as a transmit mailbox.
1 = corresponding mailbox is defined as a receive mailbox.

11 - 33

Transmit Request Set & Reset - CANTRS / CANTRR

CAN Transmission Request Set Register (CANTRS) – 0x006004

31	16
CANTRS[31:16]	

15	0
CANTRS[15:0]	

Mailbox Transmission Request Set Bits (TRS)
0 = no operation. NOTE: Bit will be cleared by CAN-Module logic after successful transmission.
1 = Start of transmission of corresponding mailbox. Set to 1 by user software;
OR by CAN –logic in case of a Remote Transmit Request.

CAN Transmission Request Reset Register (CANTRR) – 0x006006

31	16
CANTRR[31:16]	

15	0
CANTRR[15:0]	

Mailbox Transmission Reset Request Bits (TRR)
0 = no operation.
1 = setting TRRn cancels a transmission request, if not already in progress.

11 - 34

Transmit Acknowledge - CANTA

CAN Transmission Acknowledge Register (CANTA) – 0x006008

31	16
CANTA[31:16]	

15	0
CANTA[15:0]	

Mailbox Transmission Acknowledge Bits (TA)
0 = the message is not sent.
1 = if the message of mailbox n is sent successfully, the bit n of this register is set.
Note: To reset a TA bit by software: write a ‘1’ into it.

CAN Abort Acknowledge Request Register (CANAA) – 0x00600A

31	16
CANAA[31:16]	

15	0
CANAA[15:0]	

Mailbox Abort Acknowledge Bits (AA)
0 = The transmission is not aborted.
1 = The transmission of mailbox n is aborted.
Note: To reset a AA bit by software: write a ‘1’ into it.

11 - 35

Receive Message Pending - CANRMP

CAN Receive Message Pending Register (CANRMP) – 0x00600C

31	16
CANRMP[31:16]	

15	0
CANRMP[15:0]	

Mailbox Receive Message Pending Bits (RMP)
0 = the mailbox does not contain a message.
1 = the mailbox contains a valid message.
Note: To reset a RMP bit by software: write a '1' into it.

CAN Receive Message Lost Register (CANRML) – 0x00600E

31	16
CANRML[31:16]	

15	0
CANRML[15:0]	

Mailbox Receive Message Lost Bits (RML)
0 = no message was lost.
1 = an old unread message has been overwritten by a new one in that mailbox.
Note: To reset a RML bit by software: write a '1' into it.

11 - 36

Remote Frame Pending - CANRFP

CAN Remote Frame Pending Register (CANRFP) – 0x006010

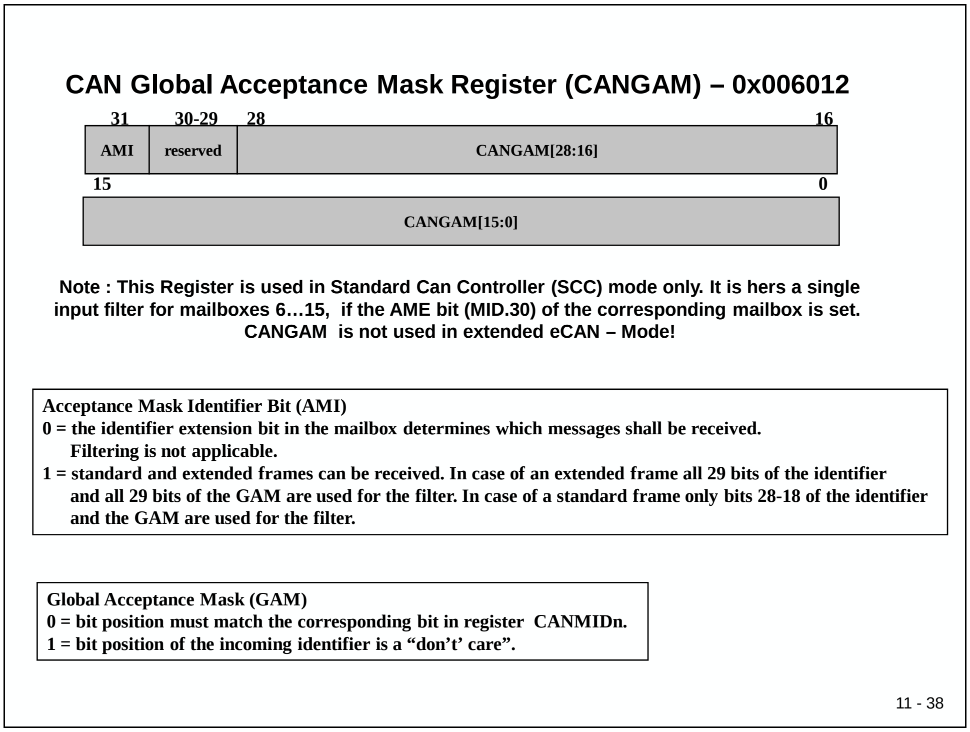
31	16
CANRFP[31:16]	

15	0
CANRFP[15:0]	

Mailbox Remote Frame Pending Bits (RFP)
0 = no remote frame request was received.
1 = a remote frame request was received by the CAN module.
Note: To reset a RFP bit by software: write a '1' into the corresponding TRR bit.

11 - 37

Global Acceptance Mask - CANGAM



The F2833x CAN module is able to operate in one of two operating modes:

- Standard CAN Controller Mode (SCC)
- Extended CAN Controller Mode, or “High End CAN Controller Mode (HECC)”.

The SCC is a legacy mode to keep the CAN communication controller software compatible to the 16-bit family TMS320F240x. In this mode there are 16 mailboxes only and the receiver system can use 3 common filters for incoming messages, LAM0, LAM1 and CANGAM. Register LAM0 is the mask register for mailboxes 0, 1 and 2; LAM1 for mailboxes 3, 4 and 5 and CANGAM for mailboxes 6...15. If you start a new design there is no advantage in using SCC mode.

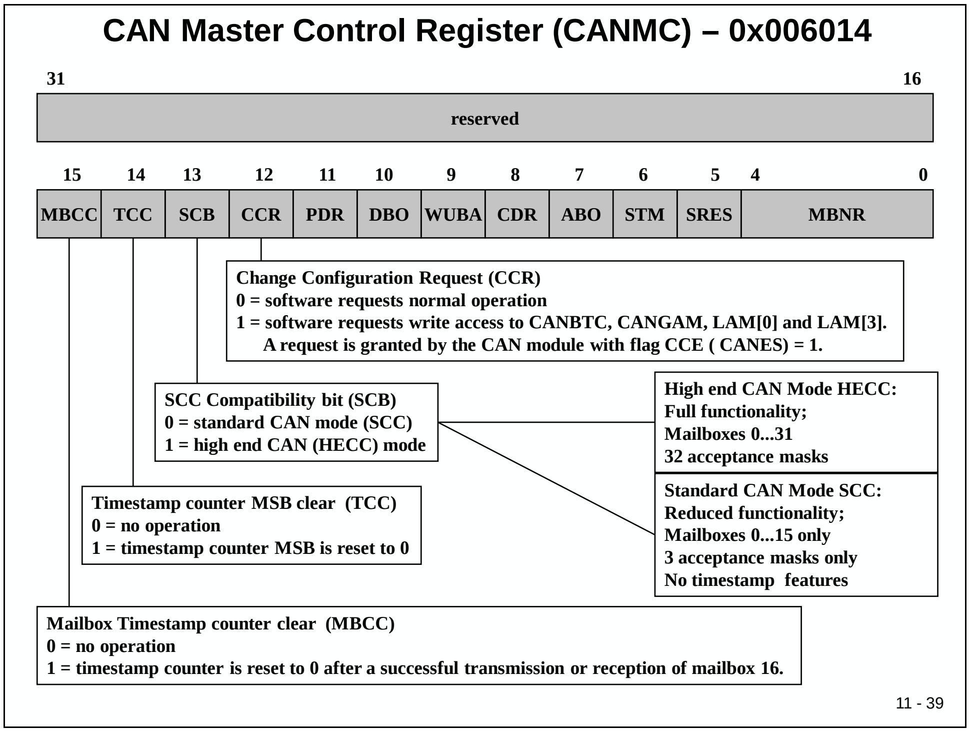
In HECC mode, each of the 32 mailboxes can be programmed to use an individual acceptance filter. Filter here means that we declare certain bits of the identifier combination of the incoming message to be “don’t cares”. This is done by setting the corresponding bits in register LAMx to ‘1’.

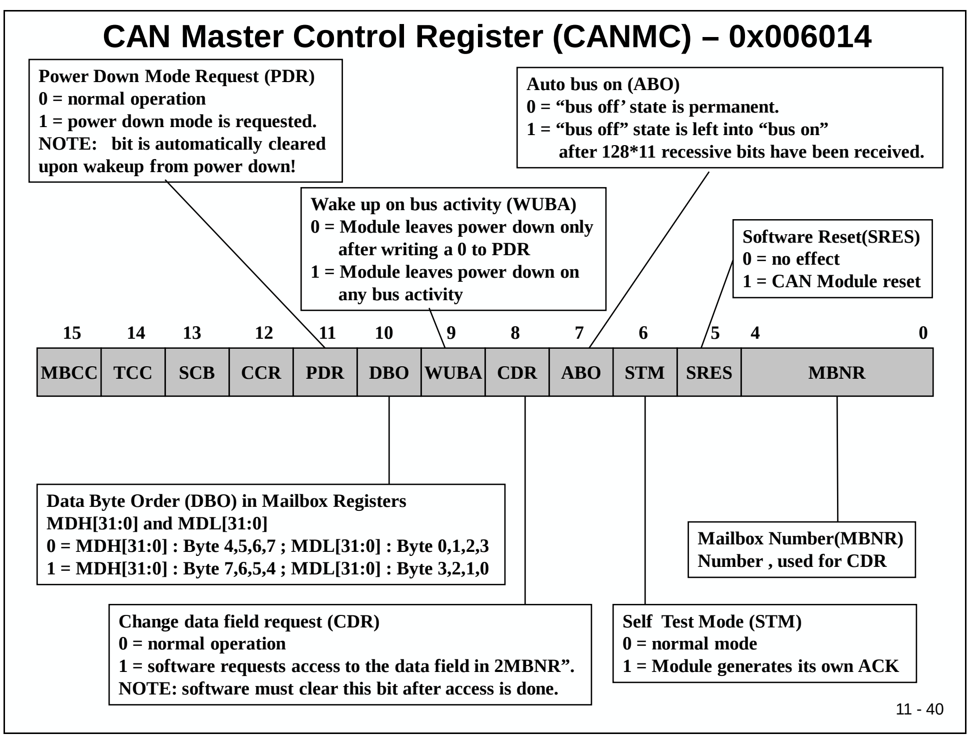
For example, if we operate in HECC mode and set LAM0 = 0x0000 0007, mailbox 0 will ignore bits 0, 1 and 2 of the incoming identifier and will store the message, if the rest of the identifier bits match the combination in register MSGID of mailbox 0.

SCC or HECC - mode is selected by bit “SCB” in register CANMC - see following slide.

Note that after reset SCC is the default mode!

Master Control Register - CANMC





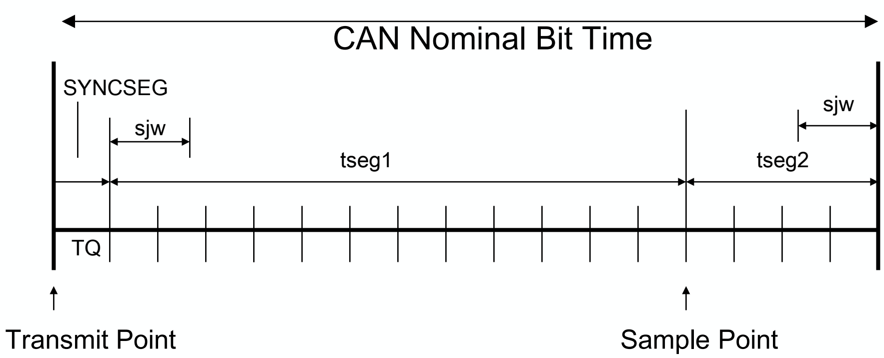
CAN Bit - Timing

CAN Bit-Timing Configuration

- **CAN protocol specification splits the nominal bit time into four different time segments:**
  - **SYNC_SEG**
    - **Used to synchronize nodes**
    - **Length : always 1 Time Quantum (TQ)**
  - **PROP_SEG**
    - **Compensation time for the physical delay times within the net**
    - **Twice the sum of the signal's propagation time on the bus line, the input comparator delay and the output driver delay.**
    - **Programmable from 1 to 8 TQ**
  - **PHASE_SEG1**
    - **Compensation for positive edge phase shift**
    - **Programmable from 1 to 8 TQ**
  - **PHASE_SEG2**
    - **Compensation time for negative edge phase shift**
    - **Programmable from 2 to 8 TQ**

11 - 41

CAN Bit-Timing Configuration



- **tseg1:** **PROP_SEG + PHASE_SEG1**
- **tseg2:** **PHASE_SEG2**
- **TQ:** **SYNCSEG**

$$T_{CAN} = TQ + tseg1 + tseg2$$

11 - 42

CAN Bit-Timing Configuration

- According to the CAN – Standard the following bit timing rules apply:
  - tseg1 $\geq$ tseg2
  - 3/BRP $\leq$ tseg1 $\leq$ 16 TQ
  - 3/BRP $\leq$ tseg2 $\leq$ 8 TQ
  - 1 TQ $\leq$ sjw $\leq$ MIN[4*TQ , tseg2]
  - BRP $\geq$ 5, if three sample mode is used

11 - 43

Bit-Timing Configuration - CANBTC

CAN Bit-Timing Configuration Register (CANBTC) – 0x006016

31 … 24	23							16
reserved	BRP.7	BRP.6	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0

Baud Rate Prescaler (BRP):
defines the Time Quantum (TQ):

$$TQ = \frac{BRP+1}{BaseCLK}$$

Note:
BaseCLK = SYSCLK / 2 for 283xx, 2803x devices
BaseCLK = SYSCLK for 281x, 280x and 2801x devices

11 - 44

CAN Bit-Timing Configuration Register (CANBTC) – 0x006016

15 – 11	10	9 – 8	7	6 – 3	2 – 0
reserved	SBG	SJW	SAM	TSEG1	TSEG2

Synchronisation Jump Width (SJW)

$$sjw = TQ * (SJW + 1)$$

Time Segment 1(tseg1)

$$tseg1 = TQ * (TSEG1 + 1)$$

Synchronisation Edge Select (SBG)
0 = re synchronisation with falling edge only
1 = re-sync. with rising & falling edge

Time Segment 2(tseg2)

$$tseg2 = TQ * (TSEG2 + 1)$$

Sample Points (SAM)
0 = one sample at sample point
1 = 3 samples at sample point – majority vote

11 - 45

CAN Bit-Timing Examples

- **Bit Configuration for BaseCLK = 75 MHz**
  - **Sample Point at 80% of Bit Time :**

CAN - data rate	BRP	TSEG1	TSEG2
1 Mbit/s	4	10	2
500 kbit/s	9	10	2
250 kbit/s	19	10	2
125 kbit/s	39	10	2
100 kbit/s	49	10	2
50 kbit/s	99	10	2

- **Example 100 kbit/s**

TQ = (49+1)/ 75 MHz = 0.667 µs
tseg1 = 0.667 µs (10 + 1) = 7.337 µs ➔ t_{CAN} = 10 µs;
tseg2 = 0.667 µs (2 + 1) = 2 µs

11 - 46

CAN Error Register

Error and Status - CANES

CAN Error and Status Register (CANES) – 0x006018

31–24	24	23	22	21	20	19	18	17	16
reserved	FE	BE	SA1	CRCE	SE	ACKE	BO	EP	EW

Form Error (FE)
0 = normal operation
1 = one of the fixed form bit fields of a message was wrong.

Bit Error (BE)
0 = no bit error detected
1 = a received bit does not match a transmitted bit (outside of the arbitration field).

Stuck at dominant Error (SA1)
0 = The CAN module detected a recessive bit
1 = The CAN module never detected a recessive bit.

Cyclic Redundancy Check Error (CRCE)
0 = normal operation
1 = a wrong CRC was received.

Stuff Bit Error (SE)
0 = normal operation
1 = a stuff bit error has occurred.

Acknowledgement Error (ACKE)
0 = normal operation
1 = CAN module has not received an ACK.

Bus Off State (BO)
0 = normal operation
1 = CANTEC has reached the limit of 256. Module has been switched of the bus.

Error Passive State (EP)
0 = CAN is in Error Active Mode
1 = CAN is in Error Passive Mode

Warning Status (EW)
0 = values of both error counters are less than 96
1 = one error counter has reached 96

11 - 47

CAN Error and Status Register (CANES) – 0x006018

15–6	5	4	3	2	1	0
reserved	SMA	CCE	PDA	Res.	RM	TM

Suspend Mode Acknowledge (SMA)
0 = normal operation
1 = CAN module has entered suspend mode.
Note: Suspend mode is activated by the debugger when the DSP is not in run mode.

Change Configuration Enable (CCE)
0 = CPU cannot write into configuration registers.
1 = CPU has write access into configuration registers.

Power Down Mode Acknowledge (PDA)
0 = normal operation
1 = CAN module has entered power down mode.

Receive Mode (RM)
0 = CAN controller is not receiving a message.
1 = CAN controller is receiving a message.

Transmit Mode (TM)
0 = CAN controller is not transmitting a message.
1 = CAN controller is transmitting a message.

11 - 48

CAN Error Counter – CANTEC / CANREC

CAN Transmit Error Counter Register (CANTEC) – 0x00601A

31 – 16
reserved

15 – 8	7 – 0
reserved	TEC

Transmit Error Counter (TEC)
Value TEC is incremented or decremented according to the CAN protocol specification

CAN Receive Error Counter Register (CANREC) – 0x00601C

31 – 16
reserved

15 – 8	7 – 0
reserved	REC

Receive Error Counter (REC)
Value REC is incremented or decremented according to the CAN protocol specification

11 - 49

CAN Interrupt Register

Global Interrupt Mask - CANGIM

CAN Global Interrupt Mask Register (CANGIM) – 0x006020

31 – 18	17	16
reserved	MTOM	TCOM

15	14	13	12	11	10	9	8	7 – 3	2	1	0
Res.	AAM	WDIM	WUIM	RMLIM	BOIM	EPIM	WLIM	reserved	GIL	I1EN	I0EN

Interrupt Mask Bits:

MTOM	= Mailbox Timeout Mask
TCOM	= Timestamp Counter Overflow Mask
AAM	= Abort Acknowledge Interrupt Mask
WDIM	= Write Denied Interrupt Mask
WUIM	= Wake-up Interrupt Mask
RMLIM	= Receive message lost Interrupt Mask
BOIM	= Bus Off Interrupt Mask
EPIM	= Error Passive Interrupt Mask
WLIM	= Warning level Interrupt Mask

Interrupt Mask Bits
0 = Interrupt disabled
1 = Interrupt enabled

Global Interrupt Level (GIL)
For Interrupts TCOF,WDIF,WUIF,BOIF and WLIF
0 = mapped into HECC_INT_REQ[0] line – GIF0
1 = mapped into HECC_INT_REQ[1] line – GIF1

Interrupt 1 Enable (I1EN)
0 = HECC_INT_REQ[1] line is disabled
1 = HECC_INT_REQ[1] line is enabled

Interrupt 0 Enable (I0EN)
0 = HECC_INT_REQ[0] line is disabled
1 = HECC_INT_REQ[0] line is enabled

11 - 50

Global Interrupt 0 Flag - CANGIF0

CAN Global Interrupt Flag 0 Register (CANGIF0) – 0x00601E

31–18	17	16
reserved	MTOF0	TCOF0

15	14	13	12	11	10	9	8	7-5	4	3	2	1	0
GMIF0	AAIF0	WDIF0	WUIF0	RMLIF0	BOIF0	EPIF0	WLIF0	Res.	MIV0.4	MIV0.3	MIV0.2	MIV0.1	MIV0.0

Interrupt Flag Bits:

MTOF0 = Mailbox Timeout Flag
TCOF0 = Timestamp Counter Overflow Flag
GMIF0 = Global Mailbox Interrupt Flag
AAIF0 = Abort Acknowledge Interrupt Flag
WDIF0 = Write Denied Interrupt Flag
WUIF0 = Wake-up Interrupt Flag
RMLIF0 = Receive message lost Interrupt Flag
BOIF0 = Bus Off Interrupt Flag
EPIF0 = Error Passive Interrupt Flag
WLIF0 = Warning level Interrupt Flag

Interrupt Flag Bits
0 = Interrupt has not occurred
1 = Interrupt has occurred

Mailbox Interrupt Vector (MIV0)
Indicates the number of the message object that set the global mailbox interrupt flag (GMIF0)

11 - 51

Global Interrupt 1 Flag - CANGIF1

CAN Global Interrupt Flag 1 Register (CANGIF1) – 0x006022

31–18	17	16
reserved	MTOF1	TCOF1

15	14	13	12	11	10	9	8	7-5	4	3	2	1	0
GMIF1	AAIF1	WDIF1	WUIF1	RMLIF1	BOIF1	EPIF1	WLIF1	Res.	MIV1.4	MIV1.3	MIV1.2	MIV1.1	MIV1.0

Interrupt Flag Bits:

MTOF1 = Mailbox Timeout Flag
TCOF1 = Timestamp Counter Overflow Flag
GMIF1 = Global Mailbox Interrupt Flag
AAIF1 = Abort Acknowledge Interrupt Flag
WDIF1 = Write Denied Interrupt Flag
WUIF1 = Wake-up Interrupt Flag
RMLIF1 = Receive message lost Interrupt Flag
BOIF1 = Bus Off Interrupt Flag
EPIF1 = Error Passive Interrupt Flag
WLIF1 = Warning level Interrupt Flag

Interrupt Flag Bits
0 = Interrupt has not occurred
1 = Interrupt has occurred

Mailbox Interrupt Vector (MIV1)
Indicates the number of the message object that set the global mailbox interrupt flag (GMIF1)

11 - 52

Mailbox Interrupt Mask - CANMIM

CAN Mailbox Interrupt Mask Register (CANMIM) – 0x006024

31	16
CANMIM[31:16]	

15	0
CANMIM[15:0]	

Mailbox Interrupt Mask Bits (MIM)
0 = mailbox interrupt is disabled.
1 = mailbox interrupt is enabled. An Interrupt is generated if a message has been transmitted successfully or if a message has been received without an error.

CAN Mailbox Interrupt Level Register (CANMIL) – 0x006026

31	16
CANMIL[31:16]	

15	0
CANMIL[15:0]	

Mailbox Interrupt Level Bits (MIL)
0 = mailbox interrupt is generated on HECC_INT_REQ[0] line.
1 = mailbox interrupt is generated on HECC_INT_REQ[1] line.

11 - 53

Overwrite Protection Control - CANOPC

CAN Overwrite Protection Control Register (CANOPC) – 0x006028

31	16
CANOPC[31:16]	

15	0
CANOPC[15:0]	

Overwrite Protection Control Bits (OPC)
0 = the old message in mailbox N may be overwritten by a new one.
This will be notified by the receive message lost bit RML[n].
1 = an old message in mailbox N is protected against being overwritten by a new one.
Thus, the next mailboxes are checked for a matching ID.
If no other mailbox is found, the new message is lost.

11 - 54

Transmit I/O Control - CANTIOC

CAN I/O Control Register (CANTIOC) – 0x00602A

31–16
reserved

15–4	3	2	1	0
reserved	TXFUNC	TXDIR	TXOUT	TXIN

TXFUNC
0 = CANTX pin is a normal I/O pin.
1 = CANTX is used for CAN transmit functions.

TXDIR
0 = CANTX pin is an input pin if configured as a normal I/O pin.
1 = CANTX pin is an output pin if configured as a normal I/O pin.

TXOUT
Output value for CANTX pin, if configured as normal output pin

TXIN
0 = Logic 0 present on pin CANTX.
1 = Logic 1 present on pin CANTX.

11 - 55

Receive I/O Control - CANRIOC

CAN I/O Control Register (CANRIOC) – 0x00602C

31–16
reserved

15–4	3	2	1	0
reserved	RXFUNC	RXDIR	RXOUT	RXIN

RXFUNC
0 = CANRX pin is a normal I/O pin.
1 = CANRX is used for CAN receive functions.

RXDIR
0 = CANRX pin is an input pin if configured as a normal I/O pin.
1 = CANRX pin is an output pin if configured as a normal I/O pin.

RXOUT
Output value for CANRX pin, if configured as normal output pin

RXIN
0 = Logic 0 present on pin CANRX.
1 = Logic 1 present on pin CANRX.

11 - 56

Alarm / Time Out Register

Local Network Time - CANLNT

CAN Local Network Time Register (CANLNT) – 0x00602E

31	16
LNT[31:16]	

15	0
LNT[15:0]	

- **LNT is a Free Running Counter, Clocked from the bit clock of the CAN module.**
- **LNT is written into the time stamp register (MOTS) of the corresponding mailbox when a received message has been stored or a message has been transmitted.**
- **LNT is cleared when mailbox #16 is transmitted or received. Thus mailbox #16 can be used for a global network time synchronization.**

11 - 57

Time Out Control - CANTIOC

CAN Time Out Control Register (CANTOC) – 0x006030

31	0
TOC[31:0]	

Time Out Control Bits (TOC)
0 = Time Out function is disabled for mailbox n.
1 = Time Out function is enabled for mailbox n.
If LNT is greater than the corresponding MOTO register, a time out event will be generated

CAN Time Out Status Register (CANTOS) – 0x006032

31	0
TOS[31:0]	

Time Out Status Flags (TOS)
0 = No Time Out occurred for mailbox n.
1 = The value in LNT is greater or equal to the value in the corresponding MOTO register

11 - 58

Local Acceptance Mask - LAMn

CAN Local Acceptance Mask Register

0x00 6040 - 0x00 607F

0 = IDE bit of mailbox determines which message shall be received
1 = extended or standard frames can be received.
extended: all 29 bit of LAM are used for filter against all 29 bit of mailbox .
standard: only first eleven bits of mailbox and LAM [28-18] are used.

31	30-29	28 — 16
LAMI	reserved	LAMn[28:16]

15 — 0
LAMn[15:0]

LAMn[28-0]: Masking of identifier bits of incoming messages
1 = don't care (accept 1 or 0 for this bit position) of incoming identifier.
0 = received identifier bit must match the corresponding message identifier bit (MID).

Note: There are two operating modes of the CAN module : "HECC" and "SCC".
In "SCC" (default after reset) LAM0 is used for mailboxes 0 to 2, LAM3 is used for mailboxes 3 to 5 and the global acceptance mask (CANGAM) is used for mailboxes 6 to 15.

In "HECC" (CANMC:13 = 1) each mailbox has its own mask register LAM0 to LAM31.

11 - 59

Message Object Time Stamp - MOTSn

CAN Message Object Time Stamp

0x00 6080 - 0x00 60BF

31 — 16
MOTSn[31:16]

15 — 0
MOTSn[15:0]

A free running counter (register CANLNT) is used to get a stamp of the time of reception or transmission of a message.

CANLNT is a 32 bit timer that is clocked by the CAN – bit – time unit.

The current content of CANLNT is written into MOTSn when a received message has been stored or a message has been transmitted successfully.

11 - 60

Message Object Time Out - MOTOn

CAN Message Object Time-Out

0x00 60C0 - 0x00 60FF

31	16
MOTOn[31:16]	
15	**0**
MOTOn[15:0]	

If the value in CANLNT is equal or greater than the value in MOTOn, the appropriate bit in register CANTOS will be set , assuming this feature was allowed in CANTOC.

Also, an Interrupt Service can be triggered from such an event.

11 - 61

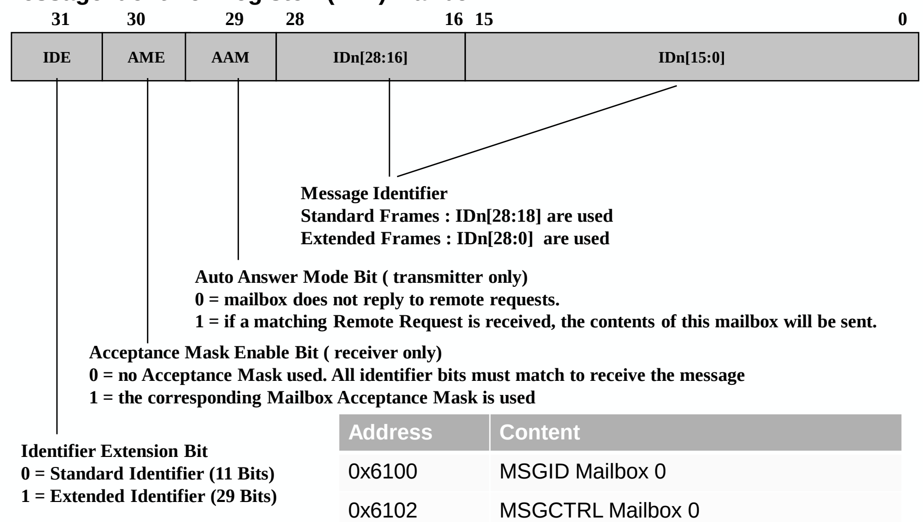
Mailbox Memory

Message Identifier - CANMID

CAN Mailbox Memory

0x00 6100 - 0x00 61FF

Message Identifier Register (MID) Mailbox n



Address	Content
0x6100	MSGID Mailbox 0
0x6102	MSGCTRL Mailbox 0
0x6104	CANMDL Mailbox 0; 4 lower data bytes
0x6106	CANMDH Mailbox 0; 4 upper data bytes

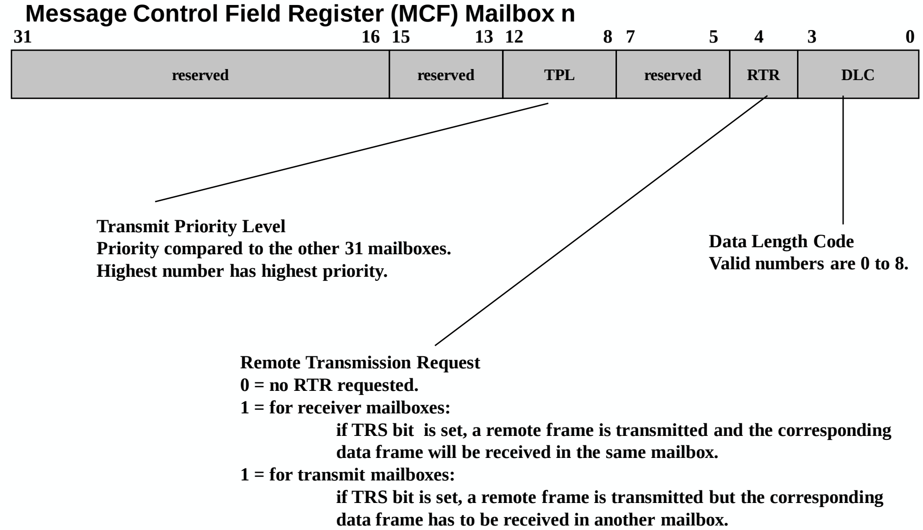
62

Message Control Field - CANMCF

CAN Mailbox Memory

0x00 6100 - 0x00 61FF

Message Control Field Register (MCF) Mailbox n



11 - 63

Message Data Field Low - CANMDL

CAN Mailbox Memory

0x00 6100 - 0x00 61FF

Message Data Low (MDL) Register with DBO = 0 Mailbox n

31 – 24	23 – 16	15 – 8	7 – 0
Data Byte 0	Data Byte 1	Data Byte 2	Data Byte 3

Message Data Low (MDL) Register with DBO = 1 Mailbox n

31 – 24	23 – 16	15 – 8	7 – 0
Data Byte 3	Data Byte 2	Data Byte 1	Data Byte 0

11 - 64

Message Data Field High - CANMDH

CAN Mailbox Memory

0x00 6100 - 0x00 61FF

Message Data High (MDH) Register with DBO = 0 Mailbox n

31 – 24	23 – 16	15 – 8	7 – 0
Data Byte 4	Data Byte 5	Data Byte 6	Data Byte 7

Message Data High (MDH) Register with DBO = 1 Mailbox n

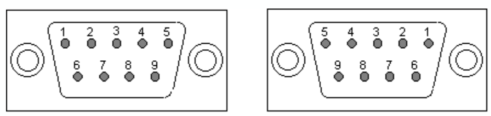
31 – 24	23 – 16	15 – 8	7 – 0
Data Byte 7	Data Byte 6	Data Byte 5	Data Byte 4

11 - 65

Lab Exercise 11_1

CAN Example: transmit a frame

- **Lab 11_1: Transmit a CAN message**
  - **CAN baud rate: 100 kBit/s**
  - **Transmit a one byte message every second**
  - **Message Identifier 0x 1000 0000 (extended frame)**
  - **Use Mailbox #5 as transmit mailbox**
  - **Message content: current value of a binary counter**
  - **Transceiver SN65HVD230 in use**
  - **Connect CAN at header J4 of Peripheral Explorer**
    - **J4-1: CAN_H**
    - **J4-2: CAN_L**



Pins Socket

11 - 66

Preface

After this lengthy (and boring) discussion of all CAN registers in an F2833x, it is time for an exercise. Again, it is a good idea to start with some simple experiments to get our hardware to work. Later, we can try to refine the projects by setting up enhanced operation modes such as "Remote Transmission Request", "Auto Answer Mode", "Pipelined Mailboxes" or "Wakeup Mode". We will also refrain from using the powerful error recognition and error management, which of course would be an essential part of a real - world project. To keep it simple, we will first use a polling method instead of an interrupt driven communication between the core of the DSP and the CAN mailbox server. Once you have a working example, it is much simpler to improve the code in this project by adding more enhanced operating modes to it.

The CAN physical layer requires a transceiver circuit between the digital signals of the F2833x and the bus lines to adjust the physical voltages. The Peripheral Explorer Board is equipped with a Texas Instruments SN65HVD230 for high speed ISO 11898 applications. This transceiver is connected to GPIO30 (CAN - RX) and GPIO31 (CAN - TX).

The physical CAN lines for ISO 11898 require a correct line termination at the ends of the transmission lines by 120 Ohm terminator resistors. The Peripheral Explorer Board has a terminator of 120 Ohm (R8) connected between CANH and CANL. This resistor can be activated by closing header J24 of the Peripheral Explorer Board. However, if your laboratory layout consists of a group of devices, only the two outmost devices should be equipped with that terminator resistor. In such circumstances all inner boards should keep jumper J24 open.

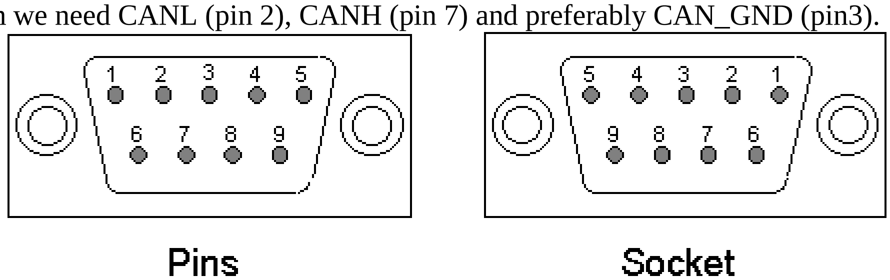
Recall that the overall line resistance should match 60 Ohms. If you are in doubt, ask your teacher which set up is the correct one.

To test your code, you will need a partner team with a second F2833x doing Lab 11_2. This lab is an experiment to receive a CAN message and display its data at GPIO9, GPIO11, GPIO34 and GPIO49 (LEDs LD1 to LD4) on the Peripheral Explorer Board.

The lines CANH and CANL are available at header J4 of the Peripheral Explorer Board. A common technique according to CiA DS 102 (www.can-cia.org) for physical CAN cables is based on DB9 connectors:

Pin Nr.	Signal	Description
1	-	Reserved
2	CAN_L	CAN Bus Signal (dominant low)
3	CAN_GND	CAN ground
4	-	Reserved
5	CAN_SHLD	Optional shield
6	GND	Optional CAN ground
7	CAN_H	CAN Bus Signal (dominant high)
8	-	Reserved
9	CAN_V+	Optional external voltage supply Vcc

At minimum we need CANL (pin 2), CANH (pin 7) and preferably CAN_GND (pin3).



Before you start the hard wiring, ask your teacher or a laboratory technician what exactly you are supposed to do to connect the boards!

Objective

- The objective of Lab 11_1 is to transmit a one byte data frame every second via CAN.
- The transmitted data byte is the current value of a binary counter, which is incremented after each transmission.
- The baud rate for this CAN exercise should be set to 100 kbit/s.
- The exercise will use extended identifier 0x1000 0000 for the transmit message. You can also use any other number as identifier, but please make sure that your partner team (Lab 11_2) knows about your intentions. If several Peripheral Explorer Boards in your classroom are in use simultaneously, there is the option to set-up pairs of teams sharing the CAN by using different identifiers. It is also

possible that due to the structure of the laboratory set-up at your university, not all identifier combinations might be available to you. You surely don't want inadvertently to start the ignition of a combustion engine control unit that is also connected to the CAN for some other experiments. Before you select other identifiers, ask your teacher!

- Use Mailbox #5 as your transmit mailbox.
- Once you have started a CAN transmission, wait for completion by polling the status bit. Doing so we can avoid using CAN interrupts for this first CAN exercise.
- Use CPU core timer 0 to generate the one second interval.

Procedure

Open Files, Create Project File

1. Using Code Composer Studio, create a new project, called **Lab11.pjt** in C:\DSP2833x_V4\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).
2. A good point to start with is the source code of Lab6.c, which produces a hardware based time period using CPU core timer 0. Open the file Lab6.c from C:\DSP2833x_V4\Labs\Lab6 and save it as Lab11_1.c in folder C:\DSP2833x_V4\Labs\Lab11.
3. Define the size of the C system stack. In the project window, right click at project "Lab11" and select "Properties". In category "C/C++ Build", "C2000 Linker", "Basic Options" set the C stack size to 0x400.

Link some of the source code files, provided by Texas Instruments, to the project:

4. In the C/C++ perspective, right click at project "Lab8" and select "**Link Files to Project**". Go to folder "*C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\source*" and link:

   - **DSP2833x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link:

   - **DSP2833x_PieCtrl.c**
   - **DSP2833x_PieVect.c**
   - **DSP2833x_DefaultIsr.c**
   - **DSP2833x_CpuTimers.c**
   - **DSP2833x_SysCtrl.c**
   - **DSP2833x_CodeStartBranch.asm**
   - **DSP2833x_ADC_cal.asm**
   - **DSP2833x_usDelay.asm**

   From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd* link:

- **DSP2833x_Headers_nonBIOS.cmd**

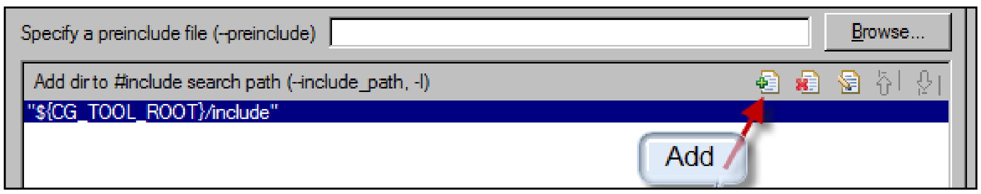
Project Build Options

5. We have to extent the search path of the C-Compiler for include files. Right click at project “Lab11” and select “Properties”. Select “C/C++ Build”, “C2000 Compiler”, “Include Options”. In the box: “Add dir to #include search path”, add the following lines:

   C:\tidcs\C28\dsp2833x\v131\DSP2833x_headers\include

   C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\include

   Note: Use the “Add” Icon to add the new paths:



   Close the Property Window by Clicking <**OK**>.

Preliminary Test

6. So far we have just created a new project “Lab11.pjt” with the same functionality as in Lab6. A good step would be to rebuild Lab11, load the code into the controller and verify the binary counter at LEDs LD1 to LD4 of the Peripheral Explorer Board. The LEDs should display the counter at 100 milliseconds time steps.

7. Now change time step size in “Lab11_1.c” from 100 ms to 1 second. All you need to do is to change the initialization call for CPU Timer 0:

   ConfigCpuTimer(&CpuTimer0,150,1000000);

8. Rebuild the code and test again; the counter frequency should be 1 second.

   Is your result as expected? NO, the LEDs are not blinking anymore!

   Do you have the answer?

   Well, we forgot to take care of the watchdog unit! When you inspect the while(1)-loop in main, you see that we wait until variable “CpuTimer0.InterruptCount” gets set to 1. Because of our change in the Timer 0 setup we now wait exactly 1000 milliseconds, which is too long for the watchdog unit.

   What can be done? We have to include the watchdog service instructions (0x55 and 0xAA) into the wait - construction.

   Change the code accordingly, rebuild and test again.

   The LEDs should now change once every second.

Note: To place both watchdog service instructions into the same place in the program is not the best solution. A better initialization would be to keep the first service instruction inside the CPU Timer 0 Interrupt service function and to add the second service instruction only into the wait - construction. However, we have to reduce the period of CPU - Timer 0 back to 100 milliseconds to keep it inside the watchdog range. In this case we have to wait until variable “CpuTimer0.InterruptCount” gets set to 10 to get the 1 second interval. If your laboratory time permits, you should try to improve your code in such a way.

Add CAN Initialization Code

9. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link to your project:

   - **DSP2833x_ECan.c**

   Before we can start editing our own code we have to inspect two files, which have been provided by Texas Instruments.

10. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\include* open “DSP2833x_Examples.h”.

    Verify that the following macros are defined as below:

```
#define DSP28_DIVSEL  2             // Enable /2 for SYSCLKOUT
#define DSP28_PLLCR   10            // multiply by 10/2
#define CPU_RATE   6.667L           // for 150MHz  (SYSCLKOUT)
#define CPU_FRQ_150MHZ   1          // 150 MHz CPU Freq (30 MHz Osc.)
```

    The source code in “DSP2833x_ECan.c” uses the macro “CPU_FRQ_150MHZ” to initialize the CAN data rate; therefore we have to make sure that this macro is set to 1.

11. Open and edit file **“DSP2833x_ECan.c”.**

    We have to set the CAN data rate to 100 kbit/s. If the F2833x runs at SYSCLKOUT = 150MHz, the CAN input clock is 75 MHz. According to the numbers given in Slide 11 - 46, we have to initialize register CANBTC with:

    - **BRP = 49**
    - **TSEG1 = 10**
    - **TSEG2 = 2**

CAN Bit-Timing Examples

- **Bit Configuration for BaseCLK = 75 MHz**
  - **Sample Point at 80% of Bit Time :**

CAN - data rate	BRP	TSEG1	TSEG2
1 Mbit/s	4	10	2
500 kbit/s	9	10	2
250 kbit/s	19	10	2
125 kbit/s	39	10	2
100 kbit/s	49	10	2
50 kbit/s	99	10	2

- **Example 100 kbit/s**

TQ = (49+1)/ 75 MHz = 0.667 µs
tseg1 = 0.667 µs (10 + 1) = 7.337 µs ➔ t_{CAN} = 10 µs;
tseg2 = 0.667 µs (2 + 1) = 2 µs

11 - 46

In function “InitECana(void)” search for the line

#if (CPU_FRQ_150MHZ)

and change the initialization values for BRPREG, TSEG1REG and TSEG2REG.

Initialize CAN Mailbox

12. Now open Lab11_1.c to edit.

    First, add a new structure “ECanaShadow” as a local variable in main:

    struct ECAN_REGS ECanaShadow;

    This structure will be used as a local copy of the original CAN registers. A manipulation of individual bits is done inside the copy. At the end of the access, the whole copy is reloaded into the original CAN structures. This operation is necessary because of the inner structure of the CAN unit; some registers are only accessible by 32 - bit accesses and by copying the whole structure, we make sure to generate 32 - bit accesses only.

13. In “main()”, after the function call “Gpio_select()”, add a function call of “InitECan()”. Also, add an external prototype for that function at the beginning of “main()”.

14. Next, inside function “Gpio_select()”, enable the peripheral function of CANA_TX and CANA_RX connected to lines GPIO30 and GPIO31.

15. In “main()”, after the function call to “InitECan()”, add code to prepare the transmit mailbox. In this exercise, we will use mailbox #5, an extended identifier of 0x10000000 and a data length code of 1. Add the following steps:

    - Write the identifier 0x10000000 into register “EcanaMboxes.MBOX5.MSGID”.
    - To transmit with extended identifiers set bit “IDE” of register “EcanaMboxes.MBOX5.MSGID” to 1.
    - Configure Mailbox #5 as a transmit mailbox. This is done by setting bit MD5 of register “ECanaRegs.CANMD” to 0. Caution! Due to the internal structure of the CAN-unit, we cannot execute single bit accesses to the original CAN registers. A good practice is to copy the whole register into a shadow register, manipulate the shadow register and copy the modified 32 - bit shadow value back into the original register :

      ECanaShadow.CANMD.all = ECanaRegs.CANMD.all;

      ECanaShadow.CANMD.bit.MD5 = 0;

      ECanaRegs.CANMD.all = ECanaShadow.CANMD.all;

    - Enable Mailbox #5:

      ECanaShadow.CANME.all = ECanaRegs.CANME.all;

      ECanaShadow.CANME.bit.ME5 = 1;

      ECanaRegs.CANME.all = ECanaShadow.CANME.all;

    - Set up the Data Length Code Field (DLC) in Message Control Register “ECanaMboxes.MBOX5.MSGCTRL” to 1 and clear all remaining bits of this register.

Add the Data Byte and Transmit

16. Now we are almost done. The last part of code modification is the periodical loading of the data byte into the mailbox and the transmit request command. This must be done inside the while(1)-loop of “main()”. Locate the code where we waited for the next period of 1 second. Here add:

    - Load the current value of variable counter into register “ECanaMboxes.MBOX5.MDL.byte.BYTE0”. Recall that we would like to send a one - byte message; therefore we have to load only the lower 8 bits of “counter”!
    - Request a transmission of mailbox #5. Init register “ECanaShadow.CANTRS”. Set bit TRS5=1 and all other 31 bits to 0. Next, load the whole register into “ECanaRegs.CANTRS”
    - Wait until the CAN unit has acknowledged the transmit request. The flag “ECanaRegs.CANTA.bit.TA5” will be set to 1 if your request has been acknowledged.
    - Clear bit “ECanaRegs.CANTA.bit.TA5”. Again the access must be made as a 32 - bit access:

      ECanaShadow.CANTA.all = 0;

      ECanaShadow.CANTA.bit.TA5 = 1;

ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;

17. Remove the old code that was used to display the binary counter at LEDs LD1 to LD4. Just keep the increment instruction for "counter".

Build, Load and Run

18. Click the "Rebuild Active Project " button or perform:

    Project → Rebuild All (Alt +B)

    and watch the tools run in the build window. If you get errors or warnings debug as necessary.

19. Load the output file in the debugger session:

    Target → Debug Active Project

    and switch into the "Debug" perspective.

20. Verify that in the debug perspective the window of the source code "Lab11_1.c" is high-lighted and that the blue arrow for the current Program Counter position is placed under the line "void main(void)".

21. Perform a real time run.

    Target → Run

    Providing you have found a partner team with another F2833x connected to your laboratory CAN system that has prepared the receiver task (Lab11_2) you can do a real network test. The current value from variable "counter" should be transmitted every second via CAN.

    If your teacher can provide a CAN analyser you should be able to trace your data frames at CAN.

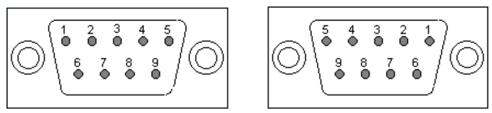
    If you end up in a fight between the two teams about whose code might be wrong, ask your teacher to provide a working receiver node. Recommendation for teachers: Store a working receiver code version in the internal Flash of one node and start this node out of flash memory.

End of Lab 11_1

Lab Exercise 11_2

CAN Example : receive a frame

- **Lab 11_2: Receive a CAN message**
  - **CAN baud rate : 100 kBit/s**
  - **Message Identifier 0x 1000 0000 (extended frame)**
  - **Use Mailbox #1 as receive mailbox**
  - **Display the binary counter at LEDs LD1 to LD4 (GPIO9, GPIO11, GPIO34 and GPIO49)**



Pins Socket

Pin Nr.	Signal	Description
1	-	Reserved
2	CAN_L	CAN Bus Signal (dominant low)
3	CAN_GND	CAN ground
4	-	Reserved
5	CAN_SHLD	Optional shield
6	GND	Optional CAN ground
7	CAN_H	CAN Bus Signal (dominant high)
8	-	Reserved
9	CAN_V+	Optional external voltage supply Vcc

11 - 67

Preface

This laboratory experiment is the second part of a CAN-Lab. Again we have to set up the physical CAN-layer according to the layout of your laboratory.

The CAN physical layer requires a transceiver circuit between the digital CAN signal levels of the F2833x and the bus lines to adjust the physical voltages. The Peripheral Explorer Board is equipped with a Texas Instruments SN65HVD230 for high speed ISO 11898 applications. This transceiver is connected to GPIO30 (CAN - RX) and GPIO31 (CAN - TX).

The physical CAN lines for ISO 11898 require a correct line termination at the ends of the transmission lines by 120 Ohm terminator resistors. The Peripheral Explorer Board has a terminator of 120 Ohm (R8) connected between CANH and CANL. This resistor can be enabled by closing header J24 of the Peripheral Explorer Board. However, if your laboratory layout consists of a group of devices, only the two out-most devices should be equipped with that terminator resistor. In such circumstances all inner boards should keep jumper J24 open. Recall that the overall line resistance should match 60 Ohms. If you are in doubt, ask your teacher which set up is the correct one.

To test your code you will need a partner team with a second F2833x doing Lab 11_1, e.g. sending a one byte message with identifier 0x10 000 000 every second.
Before you start the hard wiring, ask your teacher or a laboratory technician what exactly you are supposed to do to connect the boards!

Objective

- The objective of Lab 11_2 is to receive a one byte data message from CAN and display the four least significant bits of that byte at LEDs LD1 to LD4 (GPIO9, GPIO11, GPIO34 and GPIO49) of the Peripheral Explorer Board.

- The CAN data rate must be set to 100 kbit/s to match with Lab11_1.

- Also, to be compatible with Lab11_1, this exercise should use extended identifier 0x1000 0000 for the receive filter of mailbox 1. You can also use any other number as identifier, but please make sure that your partner team (Lab 11_1) knows about your change. If several Peripheral Explorer Boards in your classroom are in use simultaneously, it could be an option to set up pairs of teams sharing the CAN by using different identifiers.

- Use Mailbox #1 as your receiver mailbox

- Once you have initialized the CAN module, wait for a reception of mailbox #1 by polling the status bit. Again, we do not need to use CAN interrupts for this CAN exercise.

Procedure

Open Files, Create Project File

1. If you have already completed Lab11_1, you can use project Lab11.pjt as a starting point. In this case, open project Lab11 and continue with procedure step #13.

   If this Lab is your first CAN exercise, you will have to setup a new project. Using Code Composer Studio, create a new project, called **Lab11.pjt** in C:\DSP2833x_V4\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).

2. A good point to start with is the source code of Lab6.c, which produces a hardware based time period using CPU core timer 0. Open the file Lab6.c from C:\DSP2833x_V4\Labs\Lab6 and save it as Lab11_2.c in C:\DSP2833x_V4\Labs\Lab11.

3. Define the size of the C system stack. In the project window, right click at project "Lab11" and select "Properties". In category "C/C++ Build", "C2000 Linker", "Basic Options" set the C stack size to 0x400.

Link some of the source code files, provided by Texas Instruments, to the project:

4. In the C/C++ perspective, right click at project "Lab11" and select "**Link Files to Project**". Go to folder "*C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\source*" and link:

   - **DSP2833x_GlobalVariableDefs.c**

From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link:

- **DSP2833x_PieCtrl.c**
- **DSP2833x_PieVect.c**
- **DSP2833x_DefaultIsr.c**
- **DSP2833x_CpuTimers.c**
- **DSP2833x_SysCtrl.c**
- **DSP2833x_CodeStartBranch.asm**
- **DSP2833x_ADC_cal.asm**
- **DSP2833x_usDelay.asm**

From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd* link:

- **DSP2833x_Headers_nonBIOS.cmd**

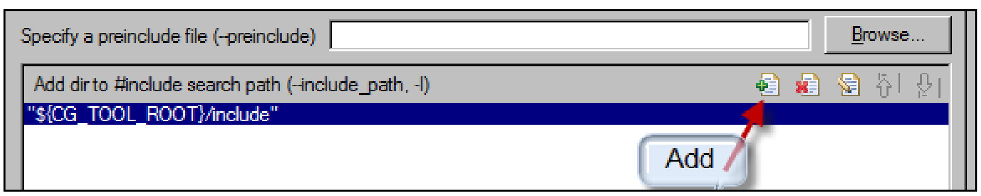
Project Build Options

5. We have to extent the search path of the C-Compiler for include files. Right click at project "Lab11" and select "Properties". Select "C/C++ Build", "C2000 Compiler", "Include Options". In the box: "Add dir to #include search path", add the following lines:

C:\tidcs\C28\dsp2833x\v131\DSP2833x_headers\include

C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\include

Note: Use the "Add" Icon to add the new paths:



Close the Property Window by Clicking <**OK**>.

Preliminary Test

6. So far we have just created a new project "Lab11.pjt" with the same functionality as in Lab6. A good step would be to rebuild Lab11, load the code into the controller and verify the binary counter at LEDs LD1 to LD4 of the Peripheral Explorer Board. The LEDs should display the counter at 100 milliseconds time steps.

Add CAN Initialization Code

7. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link:

    - **DSP2833x_ECan.c**

    Before we can start editing our own code, we have to modify two files, which have been provided by Texas Instruments:

8. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\include* open "DSP2833x_Examples.h".

    Verify that the following macros are defined as:

    ```
    #define DSP28_DIVSEL  2        // Enable /2 for SYSCLKOUT
    #define DSP28_PLLCR   10       // multiply by 10/2
    #define CPU_RATE   6.667L     // for 150MHz CPU SYSCLKOUT
    #define CPU_FRQ_150MHZ   1// 150 MHz CPU Freq (30 MHz Osc.)
    ```

    The source code in "DSP2833x_ECan.c" uses the macro "CPU_FRQ_150MHZ" to initialize the CAN data rate; therefore we have to make sure that this macro is set to 1.

9. Open and edit file **"DSP2833x_ECan.c".**

    We have to set the CAN data rate to 100 Kbit/s. If the F2833x runs at SYSCLKOUT = 150MHz, the CAN input clock is 75 MHz. According to the numbers given in Slide 11 - 46, we have to initialize register CANBTC with:

    - **BRP = 49**
    - **TSEG1 = 10**
    - **TSEG2 = 2**

CAN Bit-Timing Examples

- **Bit Configuration for BaseCLK = 75 MHz**
    - **Sample Point at 80% of Bit Time :**

CAN - data rate	BRP	TSEG1	TSEG2
1 Mbit/s	4	10	2
500 kbit/s	9	10	2
250 kbit/s	19	10	2
125 kbit/s	39	10	2
100 kbit/s	49	10	2
50 kbit/s	99	10	2

- **Example 100 kbit/s**

TQ = (49+1)/ 75 MHz = 0.667 µs
tseg1 = 0.667 µs (10 + 1) = 7.337 µs ➔ t_{CAN} = 10 µs;
tseg2 = 0.667 µs (2 + 1) = 2 µs

11 - 46

In function “InitECana(void)” search for the line

#if (CPU_FRQ_150MHZ)

and change the initialization values for BRPREG, TSEG1REG and TSEG2REG.

Save and close file “DSP2833x_ECAN.c”.

Modify Source Code

10. Open Lab11_2.c to edit.

    In “main()”, remove local variable “counter” and all instructions that use “counter” to display bits 0, 1, 2 and 3 of “counter” at GPIO9, GPIO11, GPIO34 and GPIO49.

    Add a new structure “ECanaShadow” as a local variable in main:

    struct ECAN_REGS ECanaShadow;

    This structure will be used as a local copy of the original CAN registers. A manipulation of individual bits is done inside the copy. At the end of the access the whole copy is reloaded into the original CAN structures. This principle of operation is necessary because of the inner structure of the CAN unit; some registers are only accessible by 32-bit accesses and by copying the whole structure, we make sure to generate 32-bit accesses only.

11. In “main()”, after the function call “Gpio_select()”, add a function call to “InitECan()”. Also, add an external prototype for this function at the beginning of “main()”.

12. In function “Gpio_select()”, enable the peripheral function of CANA_TX and CANA_RX connected to lines GPIO30 and GPIO31.

    Continue with procedure step #16!

13. If you have already completed Lab11_1, open the file Lab11_1.c from C:\DSP2833x_V4\Labs\Lab11 and save it as Lab11_2.c in C:\DSP2833x_V4\Labs\Lab11.

14. Exclude file “Lab11_1.c” from build. Use a right mouse click at file “Lab11_1.c”, and enable “Exclude File(s) from Build”.

15. In function “main()” of the file “lab11_2”, remove all the code, which we used to initialize the transmit mailbox #5 and the code to transmit messages with mailbox #5.

Prepare Receiver Mailbox #1

16. In “main()”, after the function call of “InitECan()”, add code to prepare the receiver mailbox. In this exercise, we will use mailbox #1, an extended identifier of 0x10000000 and a data length code of 1. Add the following steps:

    - Write the identifier into register “EcanaMboxes.MBOX1.MSGID”.

- To transmit with extended identifiers set bit "IDE" of register "EcanaMboxes.MBOX1.MSGID" to 1.
- Configure Mailbox #1 as a receive mailbox. This is done by setting bit MD1 of register "ECanaRegs.CANMD" to 1. Caution! Due to the internal structure of the CAN-unit, we cannot execute single bit accesses to the original CAN registers. A good practice is to copy the whole register into a shadow register, manipulate the shadow register and copy the modified 32 - bit shadow value back into the original register :

  ECanaShadow.CANMD.all = ECanaRegs.CANMD.all;

  ECanaShadow.CANMD.bit.MD1 = 1;

  ECanaRegs.CANMD.all = ECanaShadow.CANMD.all;

- Enable Mailbox #1:

  ECanaShadow.CANME.all = ECanaRegs.CANME.all;

  ECanaShadow.CANME.bit.ME1 = 1;

  ECanaRegs.CANME.all = ECanaShadow.CANME.all;

Wait for a message in mailbox 1

17. Now we are almost done. The last missing piece is a poll a status flag "RMP1" to see, if we have received data in mailbox 1. The best position to do this is after the 100 millisecond "while(…)" - wait construct in "main()". Register "ECanaRegs.CANRMP" - bit field "RMP1" will be set to 1 if a valid message has been received. If this bit has been set, we can proceed and process the new message.

18. If bit "RMP1" was set to 1 by the CAN - Mailbox logic we can read the data byte 0 from the mailbox and load it into a local Uint16 variable "temp":

    temp = ECanaMboxes.MBOX1.MDL.byte.BYTE0;

    Of course, we have to define "temp" at the beginning of "main()".

    Next, we have to reset bit RMP1. This is done by writing a '1' to it:

    ECanaRegs.CANRMP.bit.RMP1 = 1;

19. Finally we need some code to decode bits 0, 1, 2 and 3 of "temp" and update the LEDs at GPIO9, GPIO11, GPIO34 and GPIO49.

Build, Load and Run

20. Click the "Rebuild Active Project " button or perform:

    Project → Rebuild All (Alt +B)

    and watch the tools run in the build window. If you get errors or warnings debug as necessary.

21. Load the output file in the debugger session:

Target → Debug Active Project

and switch into the “Debug” perspective.

22. Verify that in the debug perspective the window of the source code “Lab11_2.c” is high-lighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.

23. Perform a real time run.

    Target → Run

24. Assuming you have paired with another team which transmits a one-byte data frame with identifier 0x10000000 you can do a real network test. Ask your partner team to start their board and transmit a binary counter every second.

    If your teacher can provide a CAN analyzer you can also generate a transmit message from this CAN analyzer.

    If you end up in a fight between the two teams about whose code might be wrong, ask your teacher to provide a working transmitter node.

    Recommendation for teachers: Store a working transmitter code version in the internal Flash of one node and start this node out of flash memory.

End of Lab 11_2

What’s next?

Congratulations! You’ve successfully finished your first two lab exercises using Controller Area Network. As mentioned earlier in this chapter these two labs were chosen as a sort of “getting started” with CAN. To learn more about CAN it is necessary to book additional classes at your university.

To experiment a little bit more with CAN, choose one or more of the following **optional exercises**:

Lab 11_3:

Combine Lab11_1 (CAN - Transmit) and Lab11_2 (CAN-Receive) into a bi-directional solution. The task for your node is to transmit the status of the 4-bit hex encoder (GPIO12...15) every second (or optional: every time the status has changed) with a one-byte frame and identifier 0x10 000 000. Simultaneously, your node must also be able to receive CAN messages with identifier 0x11 000 000 and display bits 0 to 3 of that message’s byte 0 at the LEDs (GPIO9 , GPIO11, GPIO34 and GPIO49) of the Peripheral Explorer Board.

Lab 11_4:

Try to improve Lab11_2 and Lab11_3 by using the F2833x Interrupt System for the receiver part of the exercises. Instead of polling the "CANRMP-bit field" to wait for an incoming message your task is to use a mailbox interrupt request to read out the mailbox when necessary.

Lab 11_5:

We did not consider any possible error situations on the CAN side so far. That is not a good solution for a real - world project. Try to improve your previous CAN experiments by including the servicing of potential CAN errors. Review the CAN error status register flags and all possible errors. A good solution would be to allow CAN error interrupts to request their individual service routines in case of a CAN failure. What should be done in the case of an error request? Answer: Well, our Peripheral Explorer Board does not feature a lot of additional hardware that we could use to indicate such an error situation. So let us just switch LED LD1 to ON in case of a failure.

Another option could be to monitor the status of the two CAN - error counters. If one of the two counters goes above 50, switch on LED LD2.

If your laboratory is equipped with a CAN failure generator like "CANstress" (Vector Informatik GmbH, Germany) you can generate reproducible disturbance of the physical layer, you can destroy certain messages and manipulate certain bit fields with bit resolution. Ask your laboratory technician whether you have access to this type of equipment to invoke CAN errors.

Lab 11_6:

An enhanced experiment is to request a remote transmission from another CAN-node. An operating mode, that is quite often used is the so-called "automatic answer mode". A transmit mailbox, that receives a remote transmission request ("RTR") answers automatically by transmitting a predefined frame. Try to establish this operating mode for the transmitter node (Lab11_1 or Lab11_3). Wait for a RTR and send the current status of the 4-bit hex encoder (GPIO12...15) back to the requesting node. The node that has requested the remote transmission should be initialized to wait for the requested answer and display the four LSBs of byte 1 from the received data frame at LEDs LD1 to LD4(GPIO9, GPIO11, GPIO34 and GPIO49).

There are a lot more options for RTR operations available. Again, look out for additional CAN classes at your university!