Department of Electrical and Computer Engineering
College of Engineering and Applied Sciences
Western Michigan University
1903 · 2003

ECE 4510 Introduction to Microprocessors

Chapter 13

Dr. Bradley J. Bazuin
Associate Professor
Department of Electrical and Computer Engineering
College of Engineering and Applied Sciences

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Chapter 13

- Controller Area Network (CAN)
  - An automotive and industrial communications network
  - Initially developed and defined by Robert Bosch GmbH of Germany in early-to-mid 1980's.
  - Recognized/released by Society of Automotive Engineers (SAE) congress in Detroit, Michigan in 1986.
  - CAN Specification 2.0 published by Bosch in 1991.
  - Primary CAN physical layer specifications
    - # ISO 11898-1: CAN Data Link Layer and Physical Signaling
    - # ISO 11898-2: CAN High-Speed Medium Access Unit

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Lower Layer Communication Protocol

- Physical Layer – the wires and signaling
- Data Link Layer – turning the signaling into addresses, data, etc for higher layers of software to use

- Other protocols with well known lower layers
  - RS-232
  - Ethernet
  - ATM

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Layered Approach in CAN (1 of 3)

- Only the logical link and physical layers are described.
- Data link layer is divided into two sublayers: logical link control (LLC) and medium access control (MAC).
  - LLC sublayer deals with message acceptance filtering, overload notification, and error recovery management.
  - MAC sublayer presents incoming messages to the LLC sublayer and accepts messages to be transmitted forward by the LLC sublayer.
  - MAC sublayer is responsible for message framing, arbitration, acknowledgement, error detection, and signaling.
  - MAC sublayer is supervised by the fault confinement mechanism.

Layered Approach in CAN (2 of 3)

- The physical layer defines how signals are actually transmitted, dealing with the description of bit timing, bit encoding, and synchronization.
- CAN bus driver/receiver characteristics and the wiring and connectors are not specified in the CAN protocol.
- System designer can choose from several different media to transmit the CAN signals.
  - Sunseeker uses a 4-conductor round cable and T-connectors
  - CANOpen uses dB9 connectors and wires

Layered Approach in CAN (3 of 3)

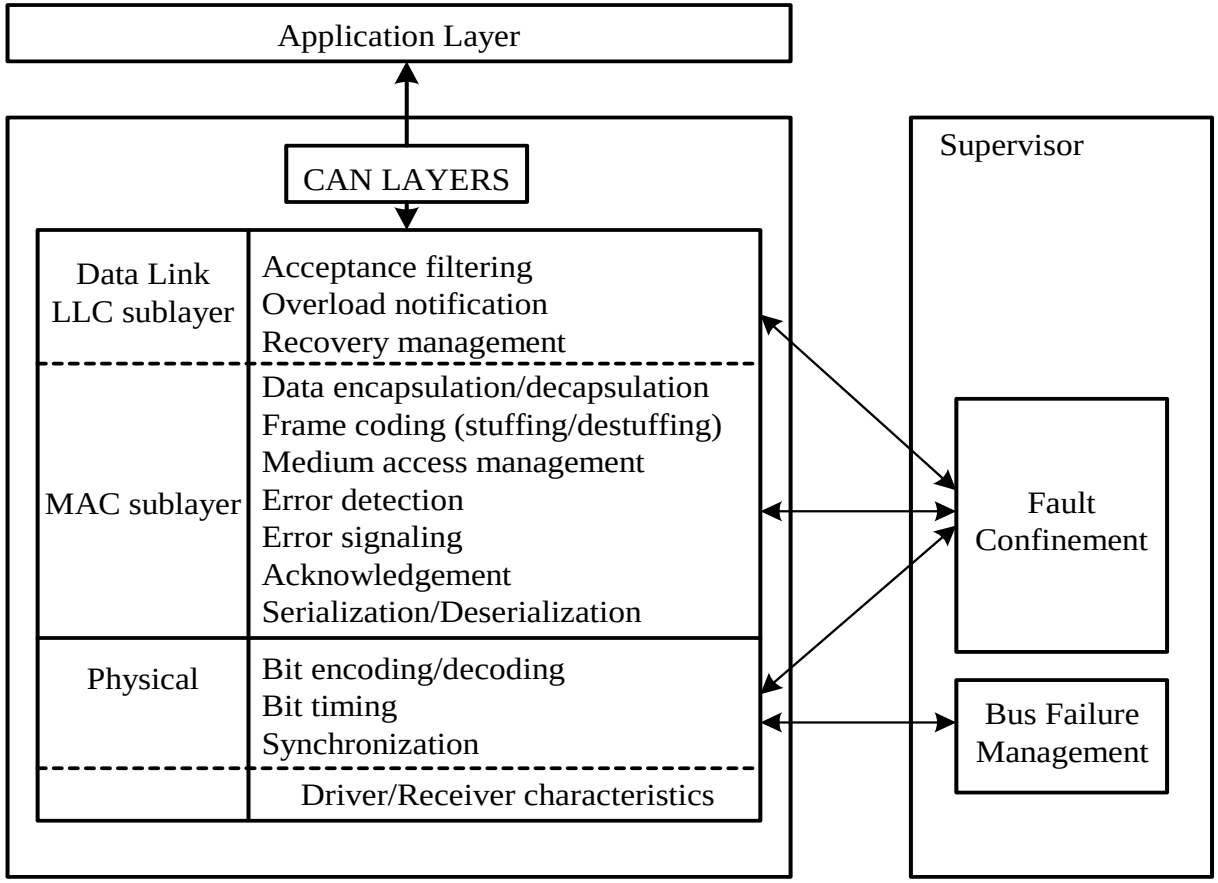


Figure 13.1 CAN layers

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

CAN Physical Layer

- Data Frames are transmitted on a two-wire common bus as CAN high (CAN_H) and CAN low (CAN_L) signals.

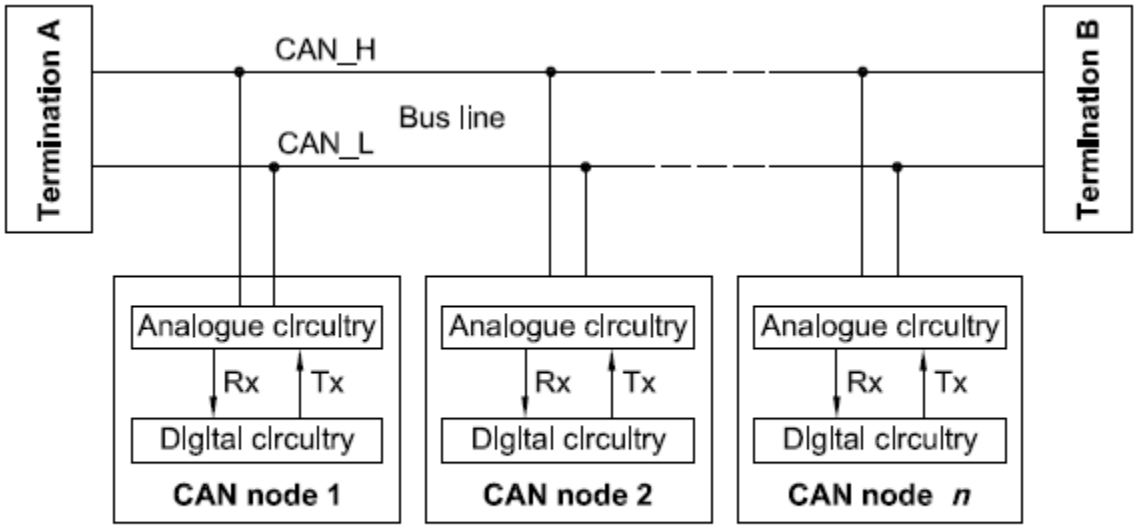


Figure 1 — Suggested electrical interconnection

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Physical Layer Signal Levels

- Both bus lines are at a nominal 2.5V
  - CAN_H goes from recessive (“1”) to dominant (“0”) by going high to nominally 3.5 V
  - CAN_L goes from recessive (“1”) to dominant (“0”) by going low to nominally 1.5 V

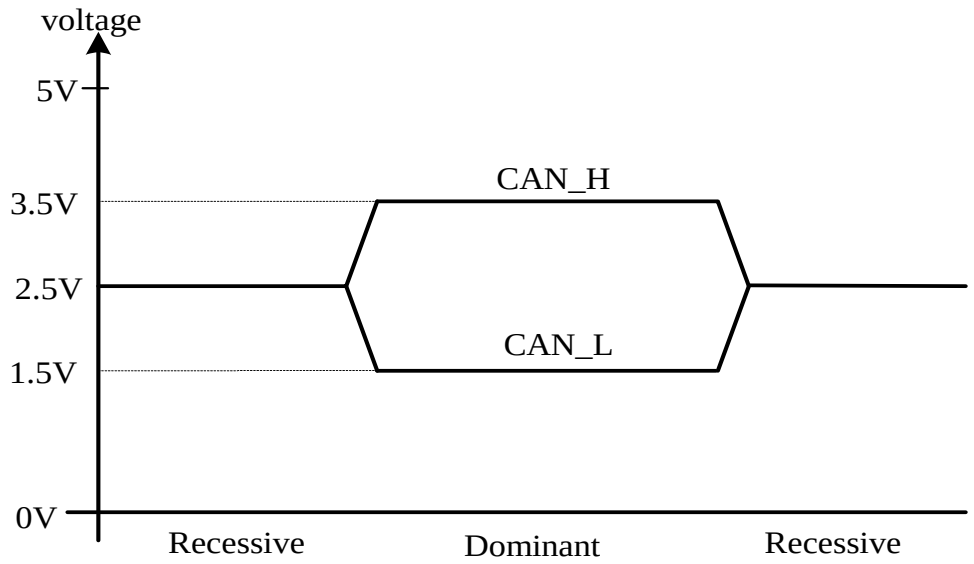


Figure 13.42 Nominal CAN bus levels

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

CAN Signaling Frame

- A data frame consists of seven fields: start-of-frame, arbitration, control, data, CRC, ACK, and end-of-frame.

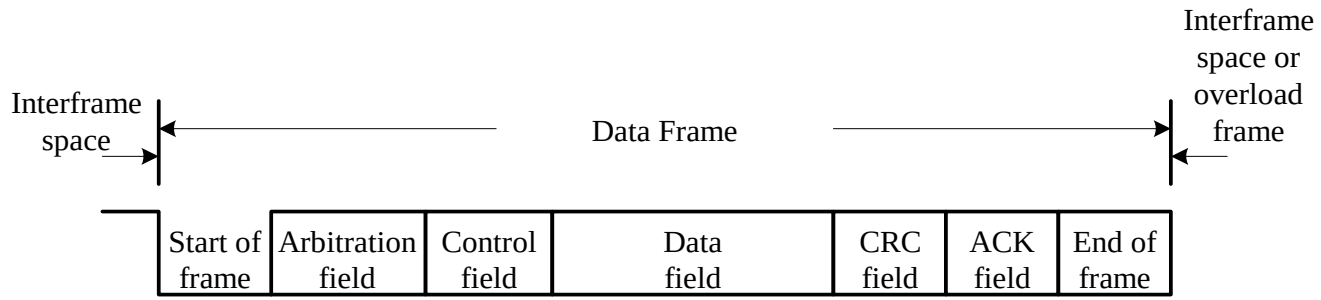


Figure 13.2 CAN Data frame

General Characteristics of CAN (1 of 3)

- Carrier Sense Multiple Access with Collision Detection (CSMA/CD)
  - Every node on the network must monitor the bus (carrier sense) for a period of no activity before trying to send a message on the bus.
  - Once the bus is idle, every node has equal opportunity to transmit a message.
  - If two nodes happen to transmit simultaneously, a nondestructive arbitration method is used to decide which node wins.
    - The node sending a dominant bit wins.
      The other node recognizes that the output doesn't match what was "sent" and is supposed to quits sending)

General Characteristics of CAN (2 of 3)

- Message-Based Communication
  - Each message contains an identifier (broadcast an ID).
    - Identifiers allow messages to arbitrate and also allow each node to decide whether to work on the incoming message.
    - The lower the value of the identifier, the higher the priority of the identifier.
  - Each node uses one or more filters to compare the incoming messages to decide whether to take actions on the message.
  - CAN protocol allows a node to request data transmission from other nodes.
  - There is no need to reconfigure the system when a new node joins the system.

General Characteristics of CAN (3 of 3)

- Error Detection and Fault Confinement
  - The CAN protocol requires each node to monitor the CAN bus to find out if the bus value and the transmitted bit value are identical.
  - The CRC checksum is used to perform error checking for each message.
  - The CAN protocol requires the physical layer to use bit stuffing to avoid long sequence of identical bit value.
  - Defective nodes are switched off from the CAN bus.

Types of CAN Messages (1 of 2)

- Data frame
- Remote frame
- Error frame
- Overload frame

Data Frame

- A data frame consists of seven fields: start-of-frame, arbitration, control, data, CRC, ACK, and end-of-frame.

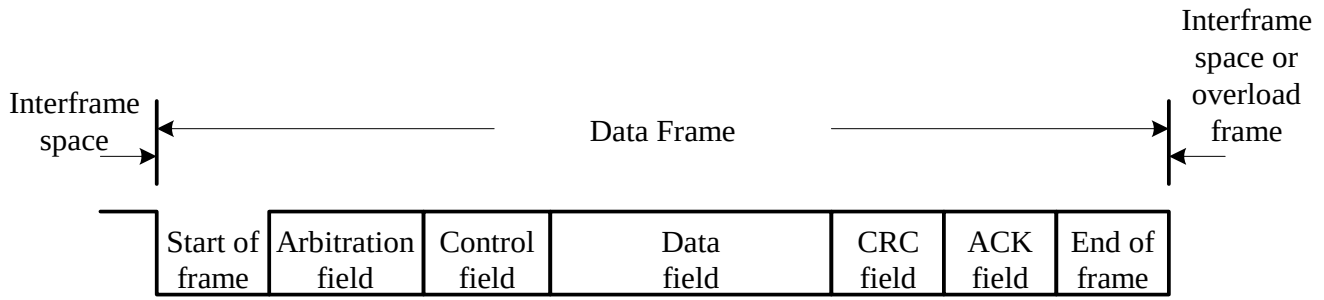


Figure 13.2 CAN Data frame

Start of Frame

- A single dominant bit to mark the beginning of a data frame.
- All nodes have to synchronize to the leading edge caused by this field.

Arbitration Field

- There are two formats for this field: standard format and extended format.

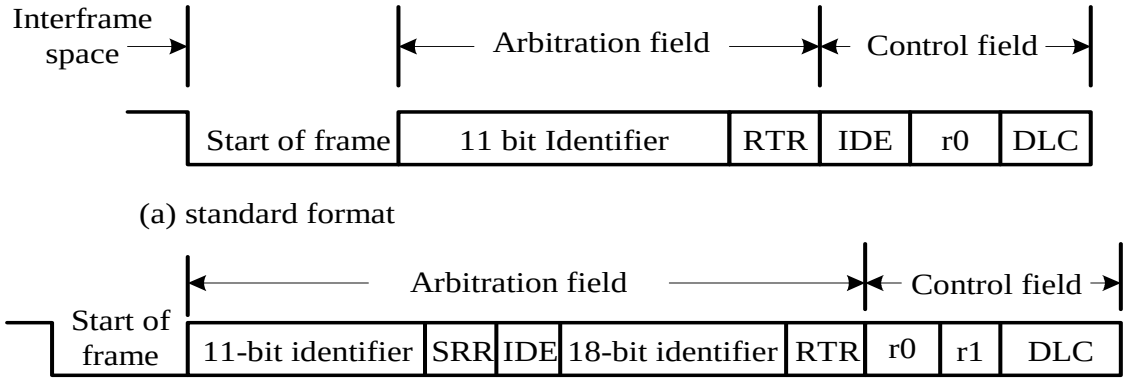


Figure 13.3 Arbitration field

- The identifier of the standard format corresponds to the base ID in the extended format.
- The RTR bit is the remote transmission request and must be 0 (dominant) in a data frame.
- The SRR bit is the substitute remote request and is recessive.
- The IDE field indicates whether the identifier is extended and should be recessive in the extended format.
- The extended format also contains the 18-bit extended identifier.

Control Field

- The first bit is IDE (ID Extended) bit for the standard format but is used as reserved bit r1 in extended format.
- r0 is reserved bit.
- DLC3…DLC0 stands for data length of the data field The data filed can be from 0000 (0) to 1000 (8) bytes.

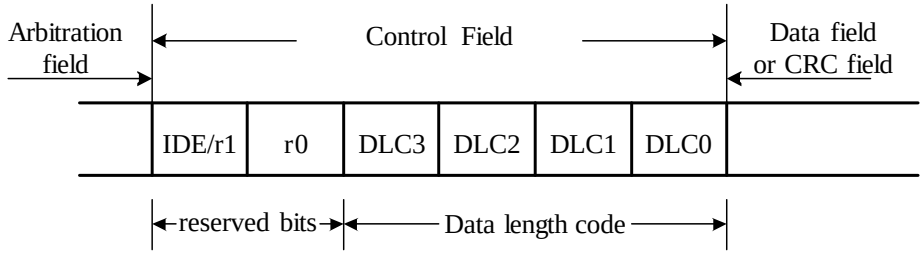


Figure 13.4 Control field

Data Field

- May contain 0 to 8 bytes of data

- Sunseeker uses 8 byte data fields consisting of
  - 2 32-bit floating point numbers
  - 2 32-bit integers (broken up into 4 integers or 8 characters) for binary “flags” (on/off for ignition key, brakes, turn signals, tetc.)

CRC Field

- It contains the 16-bit CRC sequence and a CRC delimiter.
- The CRC delimiter is a single recessive bit.

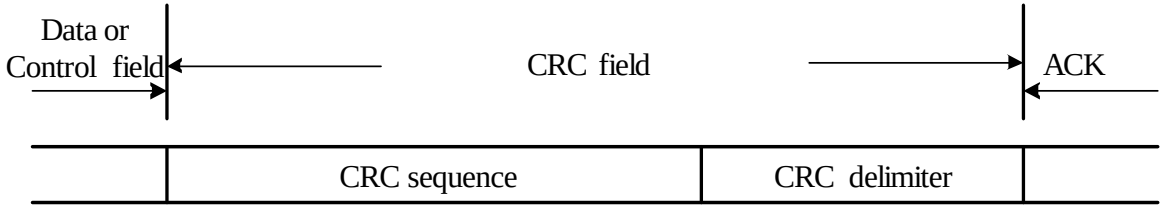


Figure 13.5 CRC field

ACK Field

- Consists of two bits
- The first bit is the acknowledgement bit.
  - This bit is set to recessive by the transmitter, but will be reset to dominant if a receiver acknowledges the data frame.
- The second bit is the ACK delimiter and is recessive.

Getting All the Bits Right

- The programmer writes/reads buffers in the peripheral devices for
  - Identifier
  - Control
  - Data length
  - Data
- The peripheral handles the rest of the details
  - CRC
  - ACK
  - Error detection and indication to host
  - resynchronization

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Automatic Retransmission

- If there is a transmission error
  and particularly if a message is not ACK (acknowledged)
  it will automatically be retransmitted.
  - Therefore for a bus with only one node:
    no ACK is possible, therefore the message is infinitely resent.
    (You can not just send CAN frames to see if they work,
    you must have another active CAN device connected!!)

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Remote Frame (Retransmit Requests or RTRs)

- Used by a node to request other nodes to send certain type of messages
- Has six fields as shown in Figure 13.7
  - These fields are identical to those of a data frame with the exception that the RTR bit in the arbitration field is recessive in the remote frame.

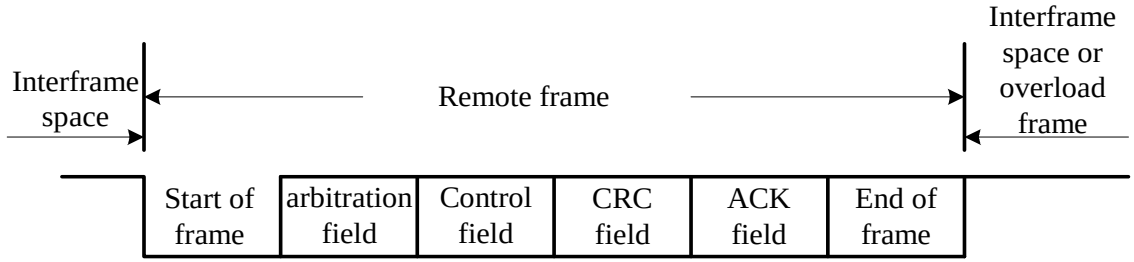
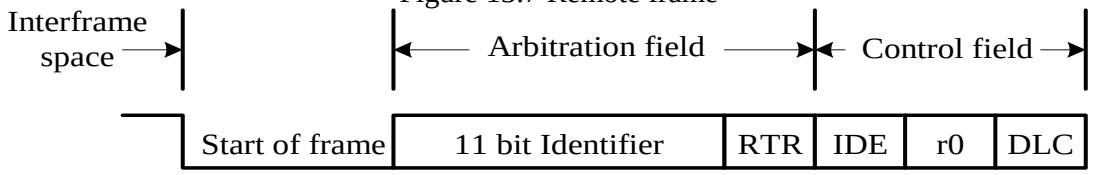


Figure 13.7 Remote frame



(a) standard format

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

CAN Message Bit Timing

- CAN uses a high rate clock to break the bit period into multiple segments.
- You program the segment lengths which sum up to the bit time.

- The setting of a bit time in a CAN system must allow a bit sent out by the transmitter to reach the far end of the CAN bus and allow the receiver to send back acknowledgement and reach the transmitter.
- The number of bits transmitted per second is defined as the nominal bit rate.

Nominal Bit Time

- The inverse of the nominal bit rate is the nominal bit time.
- A nominal bit time is divided into four segments as shown in Figure 13.12.

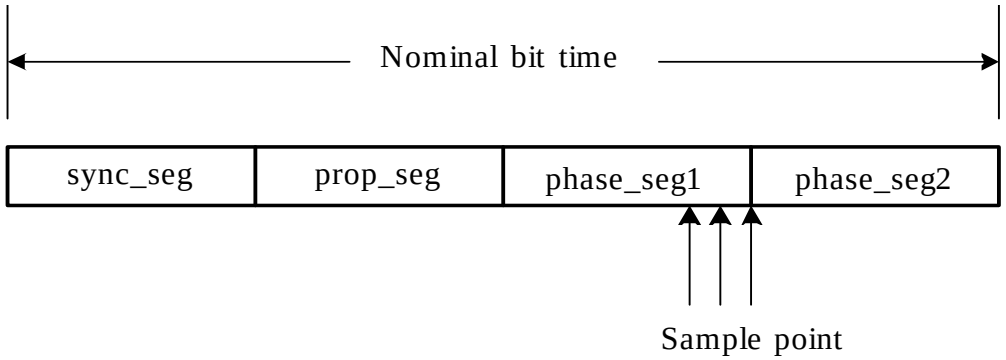


Figure 13.12 Nominal bit time

Time Quantum

- A fixed unit of time derived by dividing the oscillator period by a prescaler (derives the high rate clock)
- Length of time segments in time quantum
  - sync_seg is 1 time quantum long
  - prop_seg is programmable to be 1,2,…,8 time quanta long
  - phase_seg1 is programmable to be 1,2,…,8 time quanta long
  - phase_seg2 is programmable to be 2,3,…,8 time quanta long
- Information processing time is fixed at 2 time quanta for the HCS12.
- The total number of time quanta in a bit time must be programmable between 8 and 25.

Segments

- Sync_seg Segment (1 Tq)
  - It is used to synchronize the various nodes on the bus.
  - An edge is expected to lie in this segment.
- Prop_seg Segment (nominally 2 Tq)
  - Used to compensate for the physical delay times within the network (CAN busses have limited length!)
  - Equals twice the sum of the signal's propagation time on the CAN bus line, the comparator delay, and the output driver delay
- Phase_seg1 and Phase_seg2 Segment (TBD Tq)
  - Used to compensate for edge phase errors
  - Both can be lengthened or shortened by synchronization

Sample Point

- At the end of phase_seg1 segment.
- Users can choose to take three samples instead of one.
- A majority function determines the bit value when three samples are taken. (2 of 3 or 3 of 3)
- Each sample is separated by half time quantum from the next sample.
- The time spent on determining the bit value is the information processing time.

Nominal Bit Time (repeat)

- The inverse of the nominal bit rate is the nominal bit time.
- A nominal bit time is divided into four segments as shown in Figure 13.12.

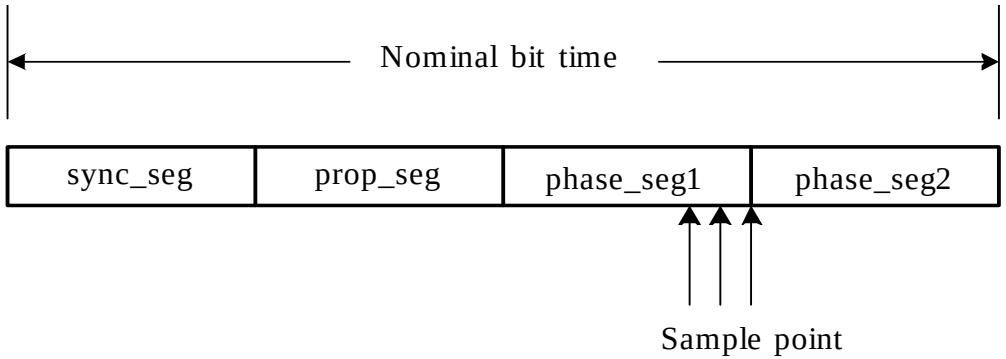


Figure 13.12 Nominal bit time

Bit_time (nTq) =

Sync_seg (1 Tq)+ (PROPSEG+TSEG1)(k Tq)+ TSEG2(m Tq)

Synchronization Issue

- All CAN nodes must be synchronized while receiving a transmission.
- The beginning of each received bit must occur during each node's sync_seg segment.
- Synchronization is needed to compensate for the difference in oscillator frequencies of each node, the change in propagation delay and other factors.
- Two types of synchronizations are defined: hard synchronization and resynchronization.
  - Hard synchronization is performed at the beginning of a message frame, when each CAN node aligns the sync_seg of its current bit time to the recessive-to-dominant transition.
  - Resynchronization is performed during the remainder of the message frame whenever a change of bit value from recessive to dominant occurs outside the expected sync_seg segment.

Resynchronization Jump Width

- The incoming recessive to dominant edge can occur
  - After the sync_seg segment but before the sample point. This is a late edge. A node will attempt to resynchronize by increasing the length of phase_seg1 segment.
  - After the sample point but before the sync_seg segment of the next bit. This is a early bit. The node will attempt to resynchronize by shortening the duration of phase_seg2 segment.
  - Within the sync_seg segment of the current bit time. No synchronization error.
- The amount of adjustment that can be made to the phase_seg1 or phase_seg2 is limited by the resynchronization jump width.
- The resynchronization jump width (SJW) is programmable to be between 1 and the smaller of 4 and phase_seg1 time quanta.

HCS12 MSCAN Peripheral

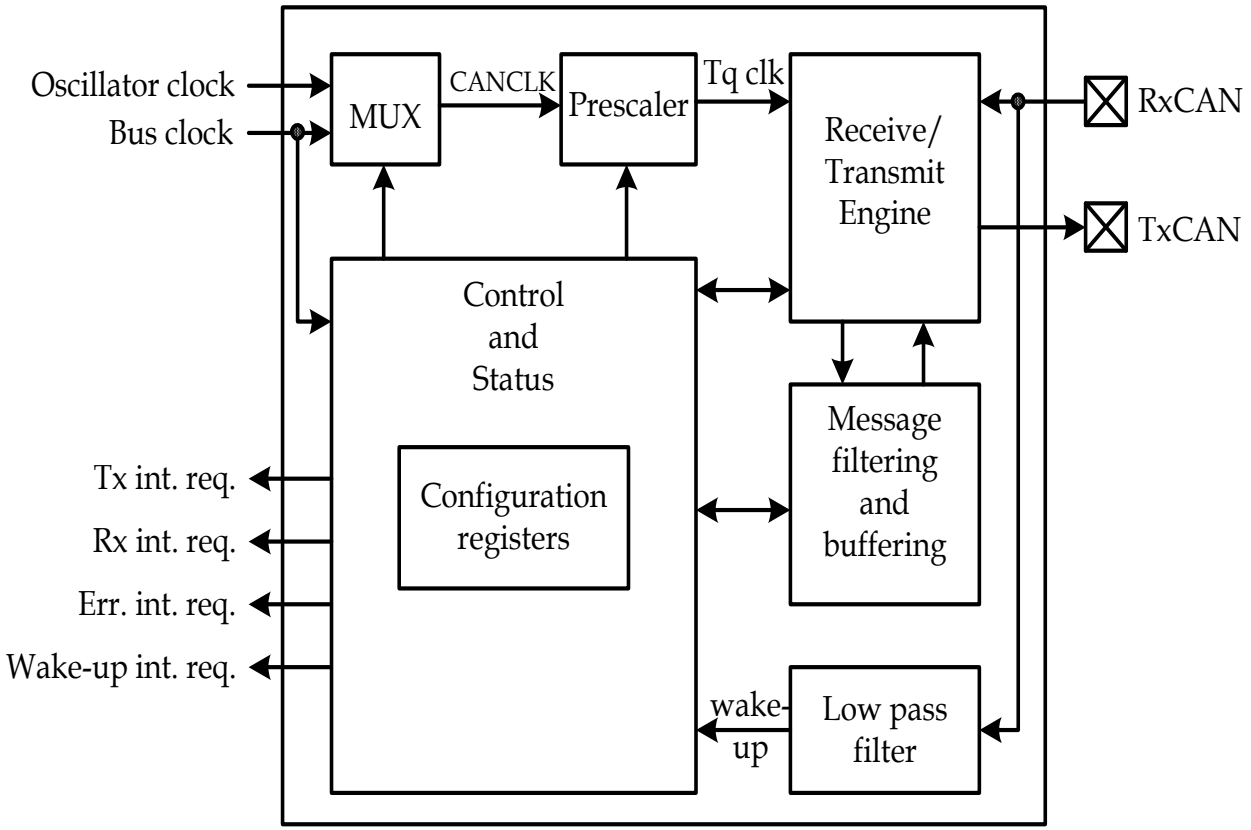
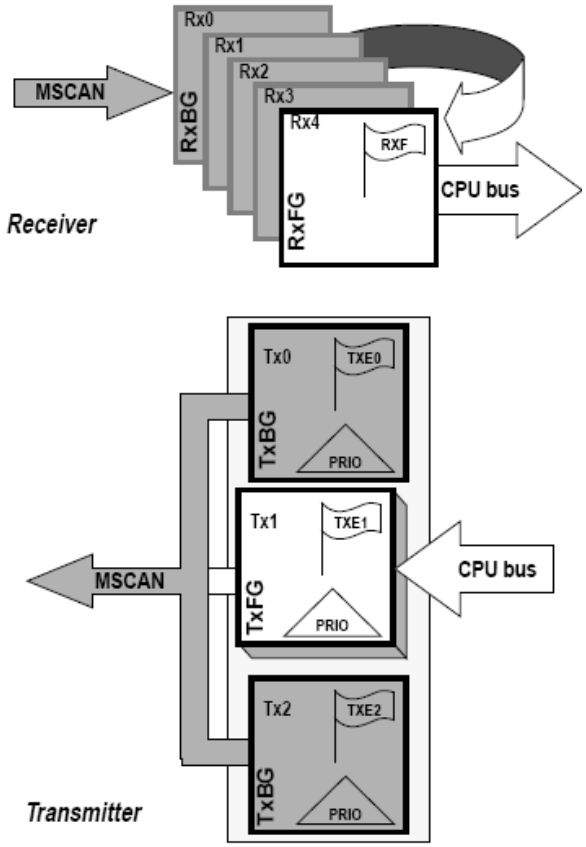


Figure 13.13 MSCAN12 block diagram

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Overview of the HCS12 CAN Module (1 of 2)

- An HCS12 device may have from one to five on-chip CAN modules.
- Each CAN module has five receive buffers with FIFO storage scheme and three transmit buffers.
- Each of the three transmit buffers may be assigned with a local priority.
- Maskable identifier filter supports two full size extended identifier filters (32-bit), four 16-bit filters, or eight 8-bit filters.
  - Only receive frames within a range of predefined identifiers (don't have to deal with all the traffic)
- The CAN module has a programmable loopback mode that supports self-test operation.
- The CAN module has a listen-only mode for monitoring of the CAN bus.



Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Overview of the HCS12 CAN Module (2 of 2)

- The CAN module has separate signaling and interrupts for all CAN receiver and transmitter error states (warning, error passive, and bus off).
- Clock signal for CAN bus can come from either the bus clock or oscillator clock. (Pick the OSCCLOCK)
- The CAN module supports time-stamping for received and transmitted messages
  - The CAN module has a 16-bit free-running timer.
- The CAN module requires a transceiver (e.g., MCP2551, PCA82C250) to interface with the CAN bus. It is built into your modules.

MSCAN Module Memory Map

- Each CAN module occupies 64 bytes of memory space.
- The MSCAN register organization is shown in Figure 13.15.
- Each receive buffer and each transmit buffer occupies 16 bytes of space.
- Only one of the three transmit buffers is accessible to the user at a time.
- Only one of the five receive buffers is accessible to the user at a time.

MSCAN Registers

address	register name	access
$_00	MSCAN control register 0 (CANCTL0)	R/W
$_01	MSCAN control register 1 (CANCTL1)	R/W
$_02	MSCAN bus timing register 0 (CANBTR0)	R/W
$_03	MSCAN bus timing register 1 (CANBTR1)	R/W
$_04	MSCAN receiver flag register (CANRFLG)	R/W
$_05	MSCAN receiver interrupt enable register (CANRIER)	R/W
$_06	MSCAN transmitter flag register (CANTFLG)	R/W
$_07	MSCAN transmitter interrupt enable register (CANTIER)	R/W
$_08	MSCAN transmitter message abort request(CANTARQ)	R/W
$_09	MSCAN transmitter message abort acknowledge (CANTAAK)	R
$_0A	MSCAN transmit buffer selection (CANTBSEL)	R/W
$_0B	MSCAN identifier acceptance control register (CANIDAC)	R/W
$_0C	reserved	
$_0D	reserved	
$_0E	MSCAN receive error counter register (CANRXERR)	R
$_0F	MSCAN transmit error counter register (CANTXERR)	R
$_10	MSCAN identifier acceptance register 0 (CANIDAR0)	R/W
$_11	MSCAN identifier acceptance register 1 (CANIDAR1)	R/W
$_12	MSCAN identifier acceptance register 2 (CANIDAR2)	R/W
$_13	MSCAN identifier acceptance register 3 (CANIDAR3)	R/W
$_14	MSCAN identifier mask register 0 (CANIDMR0)	R/W
$_15	MSCAN identifier mask register 1 (CANIDMR1)	R/W
$_16	MSCAN identifier mask register 2 (CANIDMR2)	R/W
$_17	MSCAN identifier mask register 3 (CANIDMR3)	R/W
$_18	MSCAN identifier acceptance register 4 (CANIDAR4)	R/W
$_19	MSCAN identifier acceptance register 5 (CANIDAR5)	R/W
$_1A	MSCAN identifier acceptance register 6 (CANIDAR6)	R/W
$_1B	MSCAN identifier acceptance register 7 (CANIDAR7)	R/W
$_1C	MSCAN identifier mask register 4 (CANIDMR4)	R/W
$_1D	MSCAN identifier mask register 5 (CANIDMR5)	R/W
$_1E	MSCAN identifier mask register 6 (CANIDMR6)	R/W
$_1F	MSCAN identifier mask register 7 (CANIDMR7)	R/W
$_20 $_2F	Foreground receive buffer (CANRXFG)	R
$_30 $_3F	Foreground transmit buffer (CANTXFG)	R/W

Figure 13.16 CAN module memory map

Setup

Flags

Int Enable

Identifiers Accepted & Identifier Masks

Frame: Identifiers & Data

MSCAN Message Buffers

- The receive message and transmit message buffers have the same outline.

address	register name
$_x0	Identifier register 0
$_x1	Identifier register 1
$_x2	Identifier register 2
$_x3	Identifier register 3
$_x4	Data segment register 0
$_x5	Data segment register 1
$_x6	Data segment register 2
$_x7	Data segment register 3
$_x8	Data segment register 4
$_x9	Data segment register 5
$_xA	Data segment register 6
$_xB	Data segment register 7
$_xC	Data length register
$_xD	Transmit buffer priority register[1]
$_xE	Time stamp register high byte[2]
$_xF	Time stamp register low byte[2]

Note 1. Not applicable for receive buffer.
2. Read only for CPU

Figure 13.33 MSCAN message buffer organization

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Message Buffers

- The receive message and transmit message buffers have the same outline.

address	register name
$_x0	Identifier register 0
$_x1	Identifier register 1
$_x2	Identifier register 2
$_x3	Identifier register 3
$_x4	Data segment register 0
$_x5	Data segment register 1
$_x6	Data segment register 2
$_x7	Data segment register 3
$_x8	Data segment register 4
$_x9	Data segment register 5
$_xA	Data segment register 6
$_xB	Data segment register 7
$_xC	Data length register
$_xD	Transmit buffer priority register[1]
$_xE	Time stamp register high byte[2]
$_xF	Time stamp register low byte[2]

Note 1. Not applicable for receive buffer.
2. Read only for CPU

Figure 13.33 MSCAN message buffer organization

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Identifier Registers (IDR0~IDR3)

- All four identifier registers are compared when a message with extended identifier is received.
- Only the first two identifier registers are compared when a message with standard identifier is received.

	7	6	5	4	3	2	1	0
IDR0	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
IDR1	ID20	ID19	ID18	SRR(=1)	IDE(=1)	ID17	ID16	ID15
IDR2	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
IDR3	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR

Figure 13.34 Receive/transmit message buffer extended identifier

	7	6	5	4	3	2	1	0
IDR0	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
IDR1	ID2	ID1	ID0	RTR	IDE(=0)			
IDR2								
IDR3								

Figure 13.35 Receive/transmit message buffer standard identifier

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Message Buffer Fields

- Data Segment Registers (DSR0~DSR7)
  - These registers contain the data to be transmitted or received.
  - The number of bytes to be transmitted or received is determined by the data length code.
- Data Length Register (DLR)
  - The lowest four bits of this register indicate the number of bytes contained in the message.
- Transmit Buffer Priority Register (TBPR)
  - This register defines the local priority of the associated message buffer.
  - All transmit buffer with a cleared TXEx flag participate in the prioritization.
  - The transmit buffer with the lowest local priority field wins the prioritization.
  - In case of more than one buffer having the same lowest priority, the message buffer with the lowest index number wins

CAN Foreground Receive Buffer Register Names

Table 13.2a CAN foreground receive buffer x variable names

Name	Address	Description
CANxRIDR0	$_0	CAN foreground receive buffer x identifier register 0
CANxRIDR1	$_1	CAN foreground receive buffer x identifier register 1
CANxRIDR2	$_2	CAN foreground receive buffer x identifier register 2
CANxRIDR3	$_3	CAN foreground receive buffer x identifier register 3
CANxRDSR0	$_4	CAN foreground receive buffer x data segment register 0
CANxRDSR1	$_5	CAN foreground receive buffer x data segment register 1
CANxRDSR2	$_6	CAN foreground receive buffer x data segment register 2
CANxRDSR3	$_7	CAN foreground receive buffer x data segment register 3
CANxRDSR4	$_8	CAN foreground receive buffer x data segment register 4
CANxRDSR5	$_9	CAN foreground receive buffer x data segment register 5
CANxRDSR6	$_A	CAN foreground receive buffer x data segment register 6
CANxRDSR7	$_B	CAN foreground receive buffer x data segment register 7
CANxRDLR	$_C	CAN foreground receive buffer x data length register

Note 1. x can be 0, 1, 2, or 3

2. The absolute address of each register is equal to the sum of the base address of the CAN foreground receive buffer base x address and the address field of the corresponding register.

CAN Foreground Transmit Buffer Register Names

Table 13.2b CAN foreground transmit buffer x variable names

Name	Address	Description
CANxTIDR0	$_0	CAN foreground transmit buffer x identifier register 0
CANxTIDR1	$_1	CAN foreground transmit buffer x identifier register 1
CANxTIDR2	$_2	CAN foreground transmit buffer x identifier register 2
CANxTIDR3	$_3	CAN foreground transmit buffer x identifier register 3
CANxTDSR0	$_4	CAN foreground transmit buffer x data segment register 0
CANxTDSR1	$_5	CAN foreground transmit buffer x data segment register 1
CANxTDSR2	$_6	CAN foreground transmit buffer x data segment register 2
CANxTDSR3	$_7	CAN foreground transmit buffer x data segment register 3
CANxTDSR4	$_8	CAN foreground transmit buffer x data segment register 4
CANxTDSR5	$_9	CAN foreground transmit buffer x data segment register 5
CANxTDSR6	$_A	CAN foreground transmit buffer x data segment register 6
CANxTDSR7	$_B	CAN foreground transmit buffer x data segment register 7
CANxTDLR	$_C	CAN foreground transmit buffer x data length register
CANxTBPR	$_D	CAN foreground transmit buffer x priority register
CANxTSRH	$_E	CAN foreground transmit buffer x time stamp register high
CANxTSRL	$_F	CAN foreground transmit buffer x time stamp register low

Note 1. x can be 0, 1, 2, or 3

2. The absolute address of each register is equal to the sum of the base address of the CAN foreground transmit buffer x and the address field of the corresponding register.

Transmit Storage Structure

- Multiple messages can be set up in advance and achieve real-time performance.
- A transmit buffer is made accessible to the user by writing appropriate value into the CANxTBSEL register.

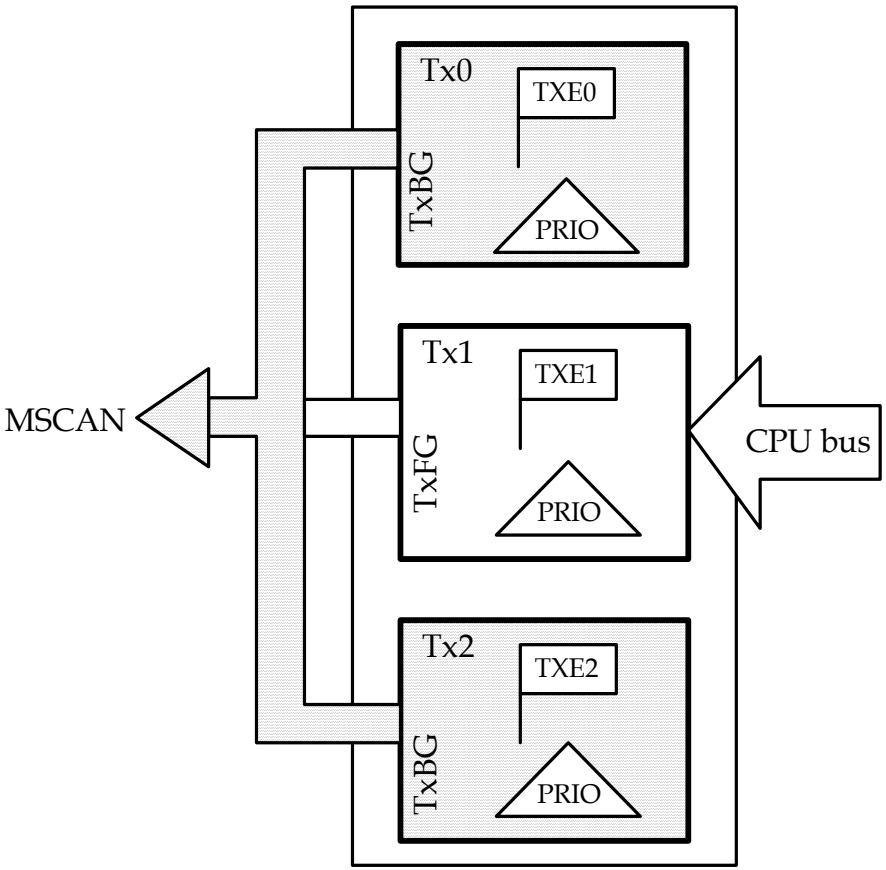


Figure 13.37 User model for transmit buffer organization

Procedure for Message Transmission

- Step 1
  - Identifying an available transmit buffer by checking the TXEx flag associated with the transmit buffer. (CANxTFLG)
- Step 2
  - Setting a pointer to the empty transmit buffer by writing the CANxTFLG register to the CANxTBSEL register. This makes the transmit buffer accessible to the user. (Moves it to the foreground.)
- Step 3
  - Storing the identifier, the control bits, and the data contents into one of the foreground transmit buffers.
- Step 4
  - Flagging the buffer as ready by clearing the associated TXE flag.

Receive Storage Structure (1 of 2)

- Received messages are stored in a five-stage FIFO data structure.
- The message buffers are alternately mapped into a single memory area referred to as the foreground receive buffer.
- The application reads the foreground receive buffer to access the received message.

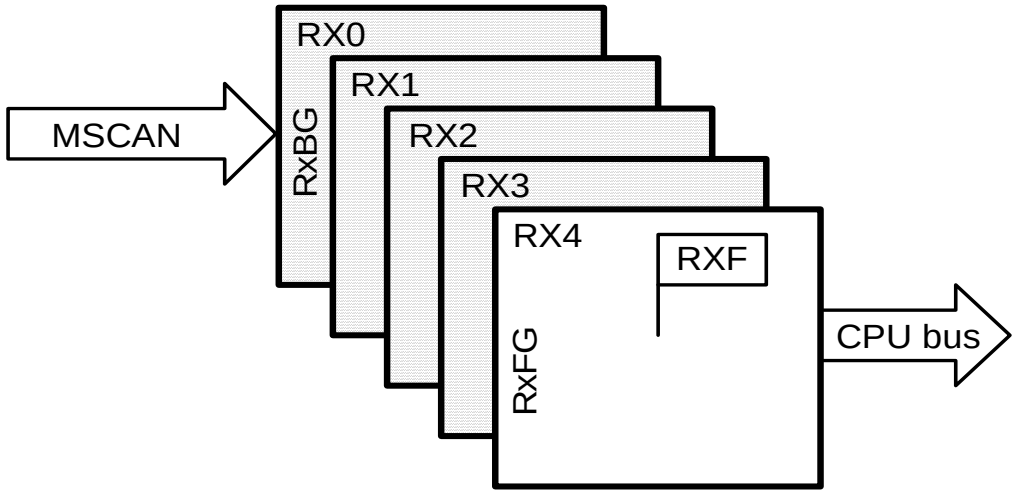


Figure 13.38 User model for receive buffer organization

Receive Storage Structure (2 of 2)

- When a valid message is received at the background receive buffer, it will be transferred to the foreground receive buffer and the RXF flag will be set to 1. (CANxRFLG & RXF)
- The user's program has to read the received message from the RxFG and then clear the RXF flag to acknowledge the interrupt and to release the foreground receive buffer.
- When all receive buffers in the FIFO are filled with received messages, an overrun condition may occur. (CANxRFLG & OVRIF)

Initialization

- 5.1 MSCAN initialization
  - The procedure to initially start up the MSCAN module out of reset is as follows:
    1. Assert CANE [CAN0CTL1 = CANE;]

    (INITRQ and INITAK cycle inserted here in ECE code)

    2. Write to the configuration registers in Initialization Mode
    3. Clear INITRQ to leave Initialization Mode and enter Normal Mode
       [CAN0CTL0 &= ~(INITRQ);
       while((CAN0CTL0 & SYNCH) == 0x00) { asm("nop");}]
  - If the configuration of registers which are writable in Initialization Mode only needs to be changed when the MSCAN module is in Normal Mode:
    1. Make sure that the MSCAN transmission queue gets empty and bring the module into Sleep Mode by asserting SLPRQ and awaiting SLPAK
    2. Enter Initialization Mode: Assert INITRQ and await INITAK
    3. Write to the configuration registers in Initialization Mode
    4. Clear INITRQ to leave Initialization Mode and continue in Normal Mode

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Initialization Request and Acknowledge

- Initialization request and acknowledgement phases

```
CAN0CTL0 = INITRQ;                //initialize the CAN
 while((CAN0CTL1 & INITAK)==0x00)
{
  asm("nop");                     // Waiting for the acknowledge of the init req.
}
```

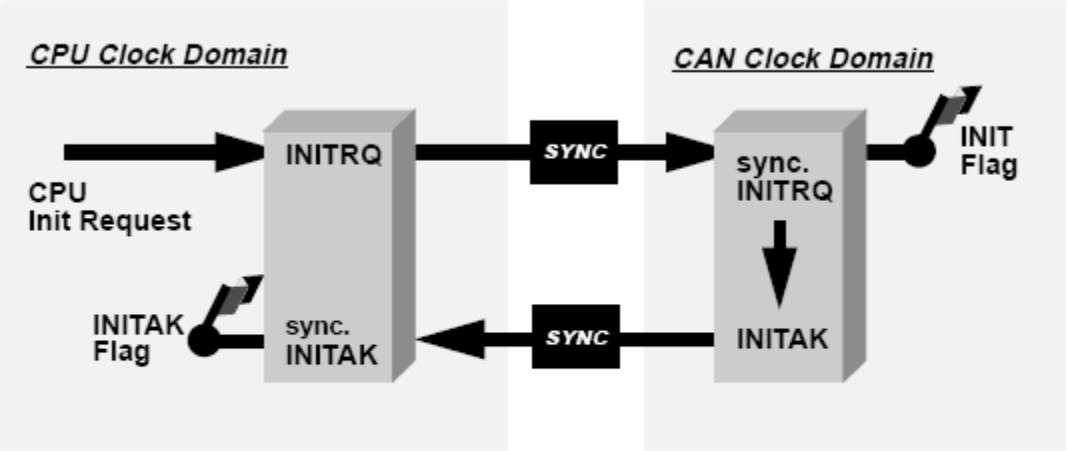


Figure 4-9 Initialization Request/Acknowledge Cycle

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Control Register 0 (CANxCTL0)

	7	6	5	4	3	2	1	0
	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ
reset:	0	0	0	0	0	0	0	1

RXFRM: Received frame flag
0 = no valid message was received
1 = a valid message was received since last clearing of this flag
RXACT: Receiver active status
0 = MSCAN is transmitting or idle
1 = MSCAN is receiving a message (including when arbitration is lost)
CSWAI: CAN stops in wait mode
0 = the module is not affected during wait mode
1 = the module ceases to be clocked during wait mode
SYNCH: synchronization status
0 = MSCAN is not synchronized to the CAN bus
1 = MSCAN is synchronized to the CAN bus
TIME: Timer enable
0 = disable internal MSCAN timer
1 = enable internal MSCAN timer and hence enable time stamp
WUPE: Wake-up enable
0 = wake-up disabled (MSCAN ignores traffic on CAN bus)
1 = wake-up enabled (MSCAN is able to restart)
SLPRQ: Sleep mode request
0 = running--The MSCAN functions normally
1 = sleep mode request--The MSCAN enters sleep mode when CAN is idle
INITRQ: Initialization mode request
0 = normal operation
1 = MSCAN in initialization mode

Figure 13.17 MSCAN control register 0 (CANxCTL0, x = 0, 1, 2, 3, or 4)

MSCAN Control Register 1 (CANxCTL1)

	7	6	5	4	3	2	1	0
	CANE	CLKSRC	LOOPB	LISTEN	0	WUPM	SLPAK	INITAK
reset:	0	0	0	1	0	0	0	1

CANE: MSCAN enable
- 0 = The MSCAN module is disabled.
- 1 = The MSCAN module is enabled.

CLKSRC: MSCAN clock source
- 0 = The MSCAN clock source is the oscillator clock.
- 1 = The MSCAN clock source is the bus clock.

LOOPB: Loop back self test mode
- 0 = Loop back self test disabled
- 1 = Loop back self test enabled

LISTEN: Listen only mode
- 0 = Normal operation
- 1 = Listen only mode activated.

WUPM: Wake-up mode
- 0 = MSCAN wakes up the CPU after any recessive to dominant edge on the CAN bus and WUPE bit of the CANCTL0 register is set to 1.
- 1 = MSCAN wakes up the CPU only in case of a dominant pulse on the CAN bus that has a length of T_{WUP} and the WUPE bit is set to 1.

SLPAK: Sleep mode acknowledge
- 0 = running--The MSCAN functions normally
- 1 = sleep mode active--The MSCAN has entered sleep mode.

INITAK: Initialization mode acknowledge
- 0 = normal operation--The MSCAN operates normally.
- 1 = Initialization mode active--The MSCAN is in initialization mode.

Figure 13.18 MSCAN control register 1 (CANxCTL1, x = 0, 1, 2, 3, or 4)

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

CAN Modes after Initialization

- Typically:
  Idle to TX/RX
  message active to Idle
  again.
- Sleep Mode used for
  powering down uP
  and CAN.

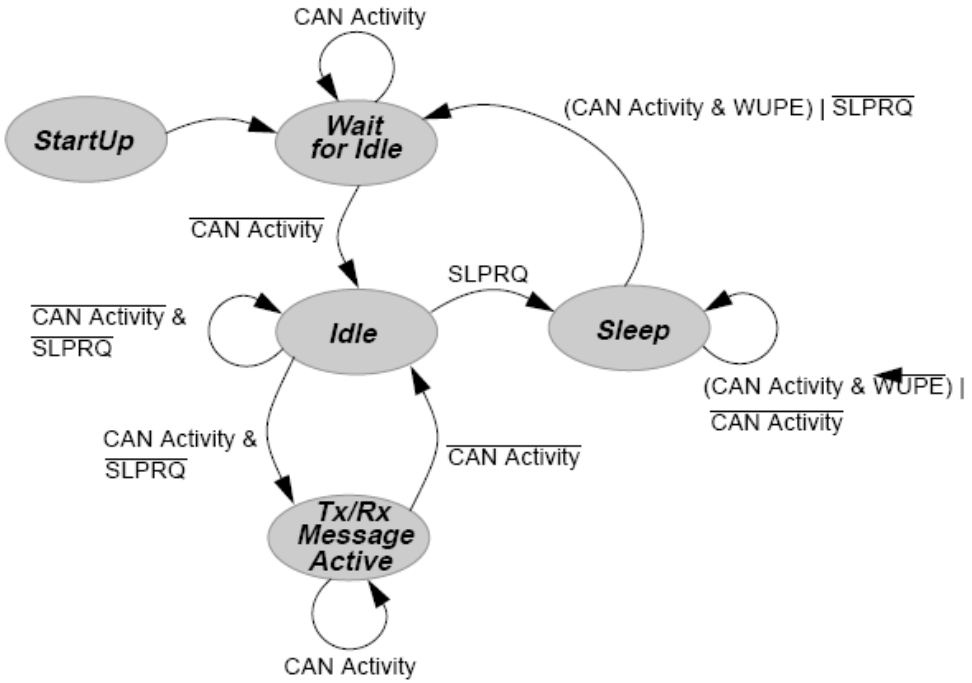


Figure 4-8 Simplified State Transitions for Entering/Leaving Sleep Mode

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Clock System

- Either the bus clock or the crystal oscillator output can be used as the CANCLK.
- The clock source has to be chosen so that it meets the 0.4% tolerance requirement of the CAN protocol. (Use OSCCLOCK)
- If the bus clock is generated from a PLL, it is recommended to select the oscillator clock rather than the bus clock due to the jitter considerations, especially at the higher baud rate.
- A programmable prescaler generates the time quanta (Tq) clock from the CANCLK.

$$f_{Tq} = f_{CANCLK} \div \text{prescaler}$$

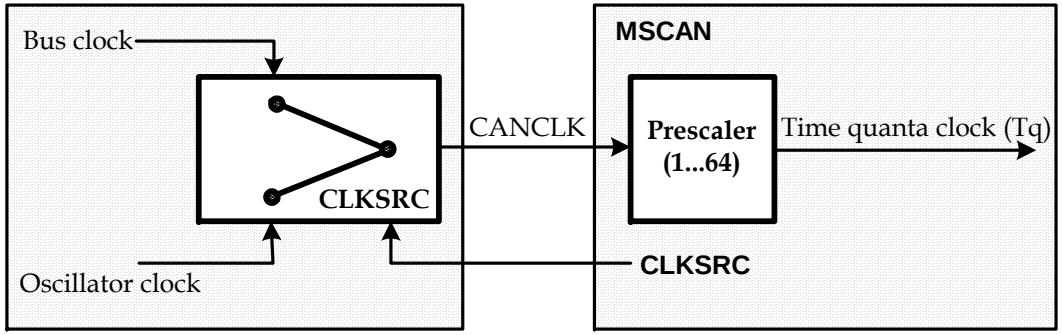


Figure 13.39 MSCAN clocking scheme

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Bit Time

- MSCAN divides a bit time into three segments:
  - Sync_seg: fixed at one time quantum
  - Time segment 1: This segment includes the prop_seg and phase_seg1 of the CAN standard.
  - Time segment 2: This segment represents the phase_seg2 of the CAN standard.

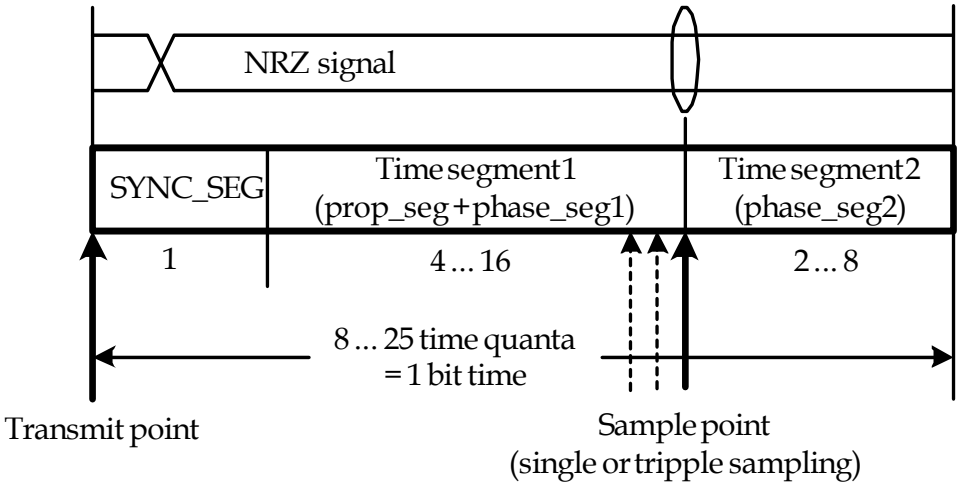


Figure 13.40 Segments within the bit time

```
//supports 1 Mbps using a 16 MHz crystal
//and 8 MHz OSCCLOCK
//Baud Rate Prescaler for 8 MHz TQ clock (8 MHz/1)
CAN0BTR0 |= 0x00;                    .
//synchronizaiton jump width are one (SJW = 1Tq).
CAN0BTR0 &= ~(SJW1 | SJW0);
 //one sample per bit, TSEG2 = 3 Tq, TSEG1+PROPSEG = 4 Tq
//8Tq = 1 Sync_seg + 4 (TSEG1+PROPSEG) + 3 TSEG2
CAN0BTR1 |= (TSEG21 |TSEG11 |TSEG10 );
```

MSCAN Bus Timing Register 0 (CANxBTR0)

- This register selects the synchronization jump width and the baud rate prescale factor.

	7	6	5	4	3	2	1	0
	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0
reset:	0	0	0	0	0	0	0	0

SJW1, SJW0: Synchronization jump width
- 00 = 1 Tq clock cycle
- 01 = 2 Tq clock cycle
- 10 = 3 Tq clock cycle
- 11 = 4 Tq clock cycle

BRP5~BRP0: Baud rate prescaler
- 000000 = 1
- 000001 = 2
- 000010 = 3
- ....
- 111110 = 63
- 111111 = 64

$$T_{qunata} = \frac{\text{BRPrescalar}}{f_{OSCCLOCK}}$$

Figure 13.19 MSCAN bus timeing register 0 (CANxBTR0, x = 0, 1, 2, 3, or 4)

MSCAN Bus Timing Register 1 (CANxBTR1)

- This register provides control on phase_seg1 and phase_seg2.
- Time Segment1 consists of prop_seg and phase_seg1.

	7	6	5	4	3	2	1	0
	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10
reset:	0	0	0	0	0	0	0	0

SAMP: Sampling
0 = One sample per bit
1 = Three samples per bit
TSEG22~TSEG20: Time segment 2
000 = 1 Tq clock cycle
001 = 2 Tq clock cycles
....
110 = 7 Tq clock cycles
111 = 8 Tq clock cycles
TSEG13~TSEG10: Time segment 1
0000 = 1 Tq clock cycle
0001 = 2 Tq clock cycles
....
1110 = 15 Tq clock cycles
1111 = 16 Tq clock cycles

$$Bit\_time = \frac{BRPrescalar}{f_{OSCCLOCK}} \cdot (1 + TSEG1 + TSEG2)$$

Figure 13.20 MSCAN bus timeing register 1 (CANxBTR1, x = 0, 1, 2, 3, or 4)

Identifier Acceptance Filter

- Identifier acceptance registers define the acceptance patterns of the standard or extended identifier.
- A message is accepted only if its associated identifier matches one of the identifier filters.
  - Any of the bits in the acceptance identifier can be marked “don’t care” in the MSCAN identifier mask registers.
- A filter hit is indicated by setting a RXF flag to 1 and the three hit bits in the CANIDAC register.
- The identifier acceptance filter can be programmed to operate in one of the four modes:
  - Two 32-bit identifier acceptance filters. This mode may cause up to 2 hits.
  - Four 16-bit identifier acceptance filters. This mode may cause up to 4 hits.
  - Eight 8-bit identifier acceptance filters. This mode may cause up to 8 hits.
  - Closed filter. No CAN message is copied into the foreground buffer RxFG.

32-bit Identifier Acceptance (MASK and ID Values)

- CANxIDMR0-7 provide masking of whether an identifier bit is tested or not
- CANxIDR0-7 provides the value to compare for unmasked bits

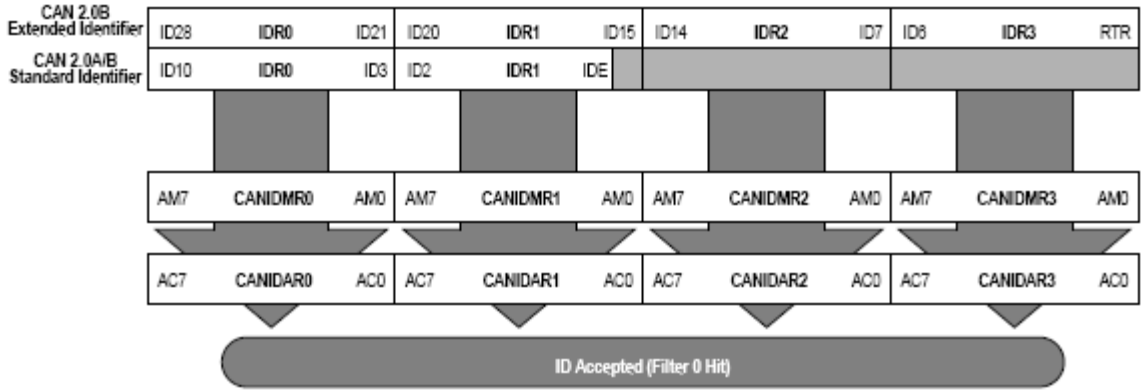


Figure 4-2 32-bit Maskable Identifier Acceptance Filter

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

16-bit Identifier Acceptance (MASK and ID Values)

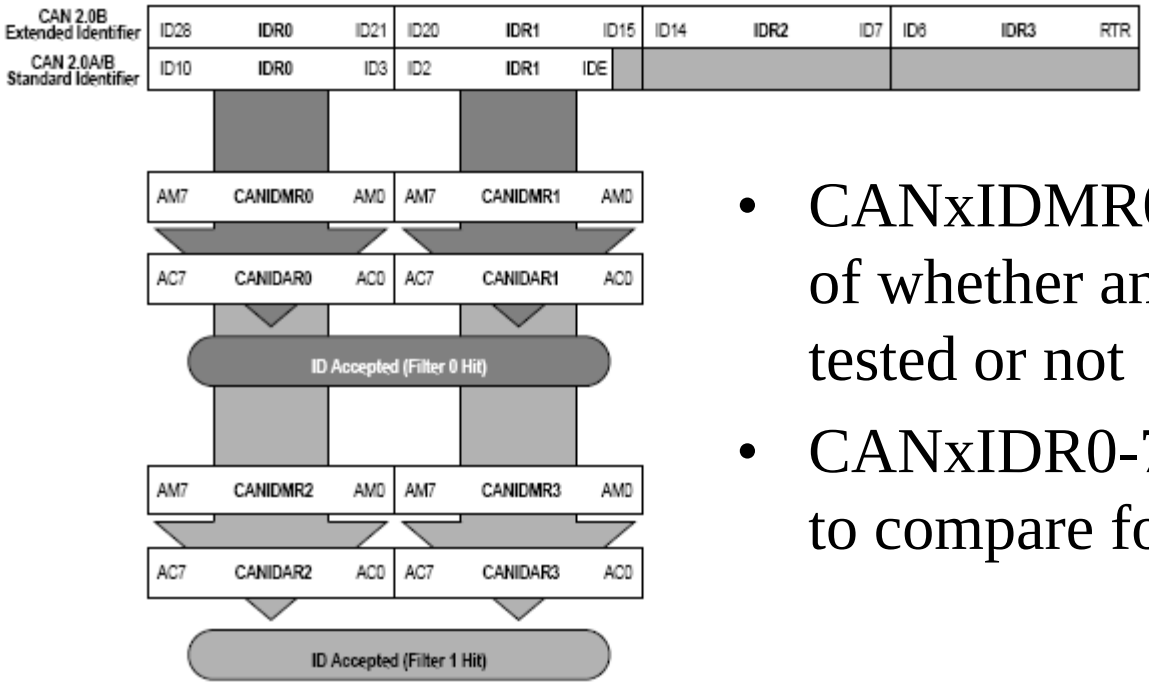


Figure 4-3 16-bit Maskable Identifier Acceptance Filters

- CANxIDMR0-7 provide masking of whether an identifier bit is tested or not
- CANxIDR0-7 provides the value to compare for unmasked bits

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Identifier Registers (IDR0~IDR7)

- Only the “first two” identifier registers are compared when a message with standard identifier is received.

	7	6	5	4	3	2	1	0
IDR0	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
IDR1	ID20	ID19	ID18	SRR(=1)	IDE(=1)	ID17	ID16	ID15
IDR2	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
IDR3	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR

Figure 13.34 Receive/transmit message buffer extended identifier

	7	6	5	4	3	2	1	0
IDR0	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
IDR1	ID2	ID1	ID0	RTR	IDE(=0)			
IDR2								
IDR3								

Figure 13.35 Receive/transmit message buffer standard identifier

CANxIDAC

32-bit: IDR0-3 and IDR4-7

16-bit: IDR0-1, IDR2-3, IDR4-5, IDR6-7

8-bit: IDRx, x=0-7

For 11-bit address frames, use 4x16-bit

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

ID MASK Registers (IDMR0~IDMR7)

- Masks whether to compare (set to 0) or not compare the bit (set to 1).

	7	6	5	4	3	2	1	0
IDR0	ID28	ID27	ID26	ID25	ID24	ID23	ID22	ID21
IDR1	ID20	ID19	ID18	SRR(=1)	IDE(=1)	ID17	ID16	ID15
IDR2	ID14	ID13	ID12	ID11	ID10	ID9	ID8	ID7
IDR3	ID6	ID5	ID4	ID3	ID2	ID1	ID0	RTR

Figure 13.34 Receive/transmit message buffer extended identifier

	7	6	5	4	3	2	1	0
IDR0	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
IDR1	ID2	ID1	ID0	RTR	IDE(=0)			
IDR2								
IDR3								

Figure 13.35 Receive/transmit message buffer standard identifier

CANxIDAC

32-bit: IDMR0-3 and IDMR4-7

16-bit: IDMR0-1, IDMR2-3, IDMR4-5, IDMR6-7

8-bit: IDMRx, x=0-7

For 11-bit address frames, use 4x16-bit

MSCAN Interrupt Operation

- Transmit interrupt (CANxTFLG and CANxTIER)
  - At least one of the three transmit buffers is empty, its TXEx flag is set.
- Receive interrupt (CANxRFLG and CANxRIER)
  - When a message is successfully received and shifted to the foreground buffer of the receive FIFO. The associated RXF flag is set.
- Wakeup interrupt (CANxRFLG and CANxRIER)
  - Activity on the CAN bus occurred during the MSCAN internal sleep mode generates this type of interrupts.
- Error interrupt (CANxRFLG and CANxRIER)
  - An overrun of the receiver FIFO, error, warning, or bus-off condition may generate an error interrupt.

Receiving Data

- CANxRFLG
  - Check for RXF to make sure foreground receibve buffer is full
- Collect the address, 11-bit in CANxIDRO0 and CANxIDRO1
  - Right align address
- Check Data size
  - Read CANxRXDLR
- Collect the Received data into a buffer

	7	6	5	4	3	2	1	0
IDR0	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3
IDR1	ID2	ID1	ID0	RTR	IDE(=0)			
IDR2								
IDR3								

Figure 13.35 Receive/transmit message buffer standard identifier

```
can.data.data_u8[0] = CAN0RXDSR0;
can.data.data_u8[1] = CAN0RXDSR1;
can.data.data_u8[2] = CAN0RXDSR2;
can.data.data_u8[3] = CAN0RXDSR3;
can.data.data_u8[4] = CAN0RXDSR4;
can.data.data_u8[5] = CAN0RXDSR5;
can.data.data_u8[6] = CAN0RXDSR6;
can.data.data_u8[7] = CAN0RXDSR7;
```


Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Receiver Flag Register (CANxRFLG)

- The flag bits WUPIF, CSCIF, OVRIF, and RXF are cleared by writing a “1” to them.

	7	6	5	4	3	2	1	0
	WUPIF	CSCIF	RSTAT1	RSTAT0	TSTAT1	TSTAT0	OVRIF	RXF
reset:	0	0	0	0	0	0	0	0

WUPIF: Wake-up interrupt flag
0 = No wake-up activity observed while in sleep mode
1 = MSCAN detected activity on the bus and requested wake-up
CSCIF: CAN status change interrupt flag
0 = No change in bus status occurred since last interrupt
1 = MSCAN changed current bus status
RSTAT1~RSTAT0: Receiver status bits
00 = RxOK: 0 ≤ Receive error counter ≤ 96
01 = RxWRN: 96 < Receive error counter ≤ 127
10 = RxERR: 127 < Receive error counter
11 = Bus-off[1]: Transmit error counter > 255
TSTAT1~TSTAT0: Transmitter status bits
00 = TxOK: 0 ≤ Transmit error counter ≤ 96
01 = TxWRN: 96 < Transmit error counter ≤ 127
10 = TxERR: 127 < Transmit error counter
11 = Bus-off: Transmit error counter > 255
OVRIF: Overrun interrupt flag
0 = No data overrun occurred
1 = A data overrun detected
RXF: Receive buffer full flag
0 = No new message availale within the RxFG
1 = The receive FIFO is not empty. A new message is available in the RxFG.
Note 1. This information is redundant. As soon as the transmitter leaves its bus off state, the receiver state skips to RxOK too.

Figure 13.21 MSCAN receiver flag register (CANxRFLG, x = 0, 1, 2, 3, or 4)

Transmit Data

- Find an empty transmit buffer
  - CANxTFLG: TXE2, TXE1 or TXE0
  - Wait until one is available
- Select the transmit buffer to write to
  - CANxTBSEL: TX2, TX1, or TX0
  - CANxTBSEL = CANxTFLG; // select the lowest numbered TX buffer
- Write Address (CANxTXIDRO0-4)
- Write Data Length (CANxTXDLR)
- Write Data (CANxTXDSR0-7)
- Set the buffer as ready to send
  - CANxTFLG |= CANTBSEL

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Transmitter Flag Register (CANxTFLG)

- This flag indicates that the associated transmit message buffer is empty, and thus not scheduled for transmission.
- The CPU must clear the flag after a message is set up in the transmit buffer and is due for transmission. Write of ‘1’ clears flag, write of ‘0’ ignored
- The MSCAN sets the flag after the message is sent successfully.

	7	6	5	4	3	2	1	0
	0	0	0	0	0	TXE2	TXE1	TXE0
reset:	0	0	0	0	0	1	1	1

TXE2~TXE0: Transmitter buffer empty
0 = The associated message buffer is full (loaded with a message due for transmission).
1 = The associated message buffer is empty.

Figure 13.23 MSCAN transmitter flag register (CANxTFLG, x = 0, 1, 2, 3, or 4)

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

MSCAN Transmit Buffer Selection (CANxTBSEL)

- This register selects the actual message buffer that will be accessible in the CANTxFG register space.
- The lowest numbered bit which is set makes the respective transmit buffer accessible to the user.

	7	6	5	4	3	2	1	0
	0	0	0	0	0	TX2	TX1	TX0
reset:	0	0	0	0	0	0	0	0

TX2~TX0: Transmit buffer select bits
0 = The associated message buffer is deselected.
1 = The associated message buffer is selected, if it is the lowest numbered bit.

Figure 13.27 MSCAN transmitter buffer select register
(CANxTBSEL, x = 0, 1, 2, 3, or 4)

Physical CAN Bus Connection

- CAN is designed for data communication over a short distance.
- CAN protocol does not specify what medium to use for data communication.
- Using a shielded or unshielded cable is recommended for a short distance communication.
- A typical CAN bus setup using a cable is shown.

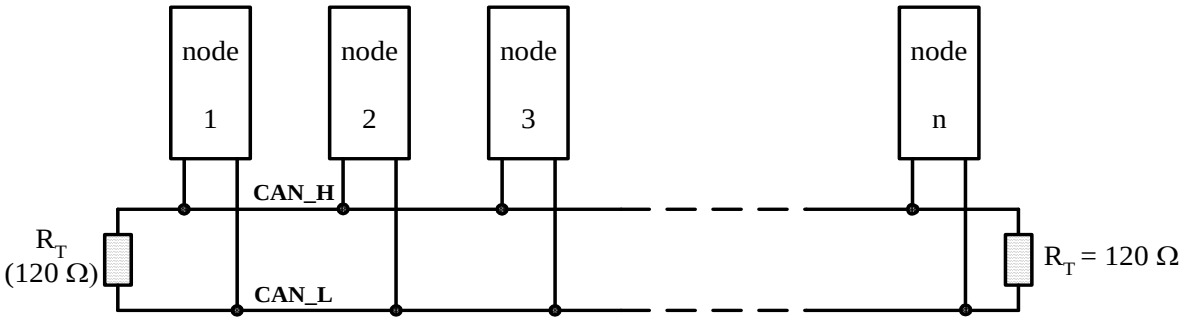


Figure 13.41 A typical CAN bus setup using cable

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Interfacing the CAN to the HCS12

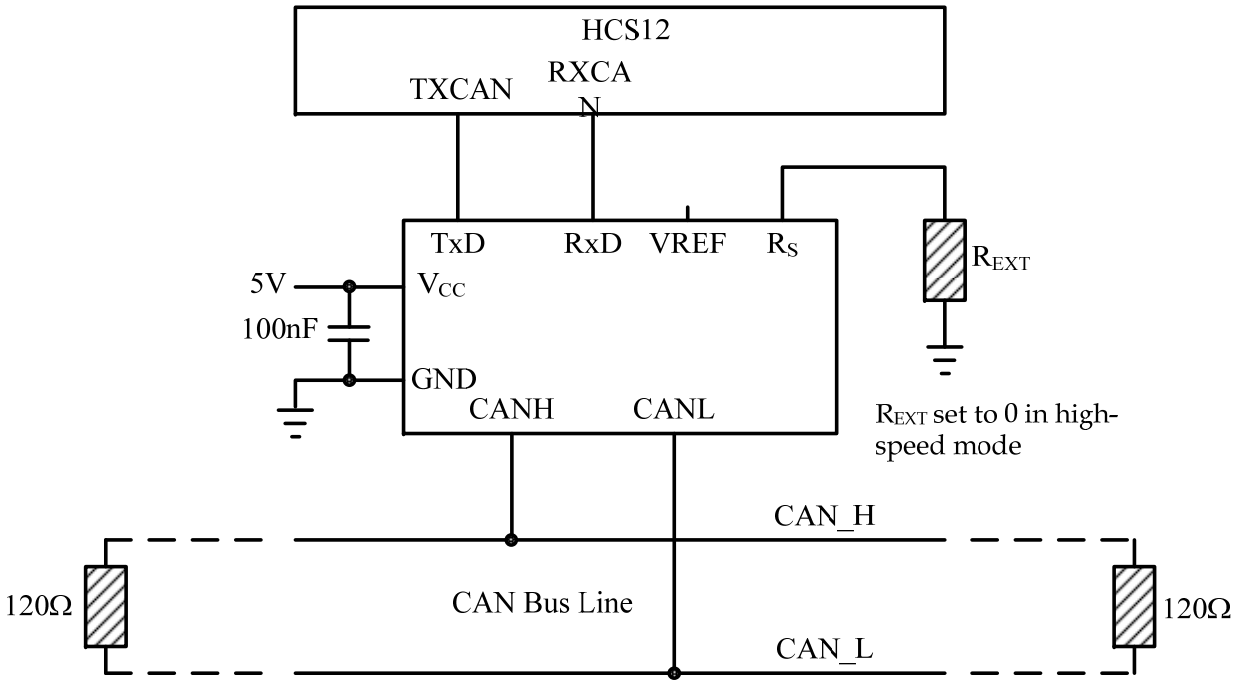


Figure 13.45 Interfacing l with the HCS12

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

CAN Transceiver PCA82C250

- Built into the Adapt modules for CAN0 (J6) and CAN1 (J7)
  - Rs=10kΩ

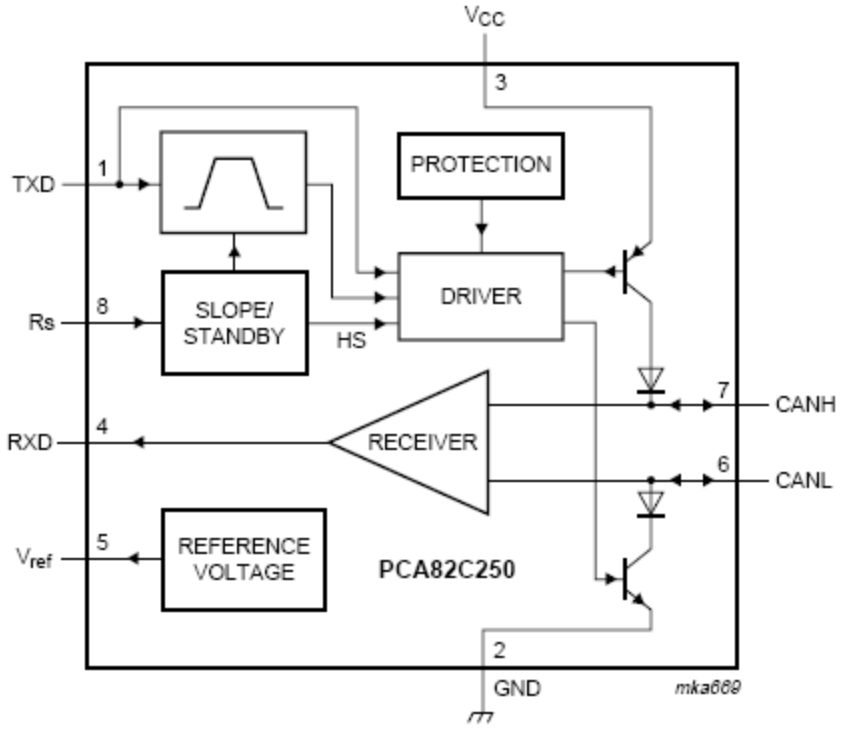


Fig 1. Block diagram

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Sunseeker Based CAN C functions

- CAN function calls for initialization, can_receive and can_transmit.
  - can0_hcs.c
  - can0_hcs.h
    - Transmit uses of to three predefined addresses or makes a new one when a transmit is to be sen
    - Receive is based on an interrupt or poll identifying that a can_receive is required prior to the function call.
- The original Tritium driver controller MSP430 code that calls can-transmit and can-receive.
  - tri63.c
  - tri63.h

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Miscellaneous Textbook Info

- CAN Timing Parameters
- Optimal Bit Timing Parameters
- Example 13.1
- C function to initialize the CAN1 Module
- C function for CAN1 Data Transmission
- C function for Interrupt Driven Data Reception

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Setting the CAN Timing Parameters (1 of 2)

- Let tBUS, tTX, and tRX represent the data traveling time on the bus, transmitter propagation delay, and receiver propagation delay, respectively.
  - The worst-case value for tPROP_SEG is
  - $$tPROP\_SEG = 2 \times (tBUS + tTX + tRX) \quad (13.4)$$
  - In units of time quantum,
  - $$prop\_seg = round\_up\ (tPROP\_SEG \div tQ) \quad (13.5)$$

Table 13.3 CAN bus bit rate /bus length relation

Bit rate (kbit/s)	Bus length
1000	40
500	100
250	250
125	500
62.5	1000

Material from or based on: *The HCS12/9S12: An Introduction to Software & Hardware Interfacing*, Thomson Delmar Learning, 2006.

Setting the CAN Timing Parameters (2 of 2)

- In the absence of bus errors, bit stuffing guarantees a maximum 10-bit period between resynchronization edges.
- The accumulated phase errors are due to the tolerance in the CAN system clock. This requirement can be expressed as
  - $(2 \times \Delta f) \times 10 \times tNBT < tRJW$ (13.6)
  - where, Δ f is the largest crystal oscillator frequency variation.
- When bus error exists, an error flag from an error active node consists of six dominant bits, and there could be up to six dominant bits before the error flag, if, for example, the error was a stuff error.
- A node must correctly sample the 13th bit after the last resynchronization. This can be expressed as
  - $(2 \times \Delta f) \times (13 \times tNBT - tPHASE\_SEG2) < \min(tPHASE\_SEG1, tPHASE\_SEG2)$ (13.7)

Procedure for Determining the Optimum Bit Timing Parameters (1 of 2)

- Step 1
  - Determine the minimum permissible tprop_seg using equation 13.4.
- Step 2
  - Choose the CAN system clock frequency. The CAN system clock frequency will be either the CPU oscillator output or the bus clock divided by a prescale factor. The chosen clock frequency must make the tNBT an integral multiple of tQ from 8 to 25.
- Step 3
  - Calculate the prop_seg duration using equation 13.5. If the resultant value is greater than 8, go back to Step 2 and choose a lower CAN system clock frequency.

Procedure for Determining the Optimum Bit Timing Parameters (2 of 2)

- Step 4
  - Determine phase1_seg and phase_seg2. Subtract the prop_seg value and 1 from the time quanta contained in a bit time. If the difference is less than 3 than go back to Step 2 and select a higher CAN system clock frequency. If the difference is 3, then phase_seg1 = 1 and phase_seg2 = 2 and only one sample per bit may be chosen. If the difference is an odd number greater than 3, then add 1 to the prop_seg value and recalculate. Otherwise divide the remaining number by two and assign the result to phase_seg1 and phase_seg2.
- Step 5
  - Determine the resynchronization jump width (RJW). RJW is the smaller one of 4 and phase_seg1.
- Step 6
  - Calculate the required oscillator tolerance from equation 13.6 and 13.7. If phase_seg1 > 4, it is recommended that you repeat Steps 2 to 6 with a larger value for the prescaler.

Example 13.1

- **Example 13.1** Calculate the CAN bit segments for the following constraints:
  - Bit rate = 1 Mbps
  - Bus length = 25 m
  - Bus propagation delay = 5×10^{-9} sec/m
  - CAN transceiver plus receiver propagation delay = 150 ns at 85°C
  - CPU oscillator frequency = 24 MHz
- **Solution:**
  - **Step 1**

    Physical delay of the CAN bus = $25 \times 5 = 125$ ns

    $t_{PROP\_SEG} = 2 \times (125 + 150) = 550$ ns
  - **Step 2**

    A prescaler of 1 for 24 MHz gives a time quantum of 41.67 ns.

    One bit time is 1/1 Mbps = 1 μs.

    One bit time (NBT) corresponds to 24 (= 1000 ns ÷ 41.67) time quanta.

– **Step 3**

Prop_seg = round_up (550 ns ÷ 41.67) = 14 > 8. Set prescaler to 2. Then one time quantum is 83.33 ns and one bit time is 12 time quanta. The new prop_seg = 7.

– **Step 4**

NBT – prop_seg1 – sync_seg = 12 – 7 – 1 = 4.

phase_seg1 = 4/2 = 2,

phase_seg2 = 4 – phase_seg1 = 2

– **Step 5**

RJW = min (4, phase_seg1) = 2

– **Step 6**

From equation 13.7,

$$\Delta f < RJW \div (20 \times NBT) = 2 \div (20 \times 12) = 0.83\%$$

From equation 13.8,

$$\Delta f < \min(phase\_seg1, phase\_seg2) \div [2 \times (13 \times NBT - phase\_seg2)]$$
$$= 2 \div 308 = 0.65\%$$

The desired oscillator tolerance is 0.65%.

- Most crystal oscillators have tolerance smaller than 0.65%.
- In **summary**,

Prescaler	= 2
Nominal bit time	= 12
prop_seg	= 7
sync_seg	= 1
phase_seg1	= 2
phase_seg2	= 2
RJW	= 2
oscillator tolerance	= 0.65%

C Function to Initialize the CAN1 Module

```
void openCan1(void)
{
        CAN1CTL1        |= CANE;          /* enable CAN, required after reset */
        CAN1CTL0        |= INITRQ;        /* request to enter initialization mode */
        while(!(CAN1CTL1&INITAK)); /* wait until initialization mode is entered */
        CAN1CTL1        = 0x84;           /* enable CAN1, select oscillator as MSCAN clock
                                             source, enable wakeup filter */
        CAN1BTR0        = 0x41;           /* set SJW to 2, set prescaler to 2 */
        CAN1BTR1        = 0x18;           /* set phase_seg2 to 2Tq, phase_seg1 to 2Tq,
                                           prop_seg to 7 Tq */
        CAN1IDAR0       = 0x54;           /* set acceptance identifier "T1" */
        CAN1IDAR1       = 0x3C;           /* "   */
        CAN1IDAR2       = 0x40;           /* "   */
        CAN1IDAR3       = 0x00;           /* "   */
        CAN1IDMR0       = 0x00;           /* acceptance mask for "T1" */
        CAN1IDMR1       = 0x00;           /* "         */
        CAN1IDMR2       = 0x3F;           /* "         */
        CAN1IDMR3       = 0xFF;           /* "         */
```

```
        CAN1IDAR4       = 0x50;         /* set acceptance identifier "P1" */
        CAN1IDAR5     = 0x3C;           /* "   */   CAN1IDAR6 = 0x40;        /* "   */
        CAN1IDAR7     = 0x00;           /* "   */
        CAN1IDMR4     = 0x00;           /* acceptance mask for "P1" */
        CAN1IDMR5     = 0x00;           /* "         */
        CAN1IDMR6     = 0x3F;           /* "         */
        CAN1IDMR7     = 0xFF;           /* "         */
        CAN1IDAC      = 0x00;           /* select two 32-bit filter mode */

        CAN1CTL0      &= ~INITRQ;       /* exit initialization mode *
        CAN1CTL0      = 0x24;           /* stop clock on wait mode, enable wake up */
}
```

C Function for CAN1 Data Transmission

```
void snd2can1(char *ptr)
{
        int     tb,i,*pt1,*pt2;
        pt1     = (int *)ptr;    /* convert to integer pointer */
        while(1) {               /* find an empty transmit buffer */
        if(CAN1TFLG & 0x01){
                        tb = 0;
                        break;
        }
        if(CAN1TFLG & 0x02){
                        tb = 1;
                        break;
        }
        if(CAN1TFLG & 0x04){
                        tb = 2;
                        break;
        }
        }
        CAN1TBSEL = CAN1TFLG; /* make empty transmit buffer accessible */
        pt2 = (int *)&CAN1TIDR0;      /* pt2 points to the IDR0 of TXFG */
```

```
        for (i = 0; i < 7; i++) /* copy the whole transmit buffer */
        *pt2++ = *pt1++;
        if (tb == 0)
                CAN1TFLG = 0x01;   /* mark buffer 0 ready for transmission */
        else if (tb == 1)
                CAN1TFLG = 0x02;   /* mark buffer 1 ready for transmission */
        else
                CAN1TFLG = 0x04;   /* mark buffer 2 ready for transmission */
}
```

C Program for the Interrupt-driven Data Reception in CAN Bus

```
#include      “c:\egnu091\include\hcs12.h”
#include      “c:\egnu091\include\vectors12.h”
#include      “c:\egnu091\include\delay.c”
#include      “c:\egnu091\include\lcd_util_SSE256.c”
#define       INTERRUPT __attribute__((interrupt))
void    INTERRUPT RxISR(void);
void    openCan1(void);
char    *t1Msg = "Temperature is";
char    *v1Msg = "Voltage is";
int main (void)
{
        UserMSCAN1Rx = (unsigned short) &RxISR;
        openCan1();
        openlcd();
        CAN1RIER = 0x01;                /* enable CAN1 RXF interrupt only */
        asm("cli");
        while(1);                       /* wait for RXF interrupt */
        return 0;
}
```

```
void    INTERRUPT RxISR (void)
{
        char tmp,i,*ptr;
        if (!(CAN1RFLG & RXF))    /* interrupt not caused by RXF, return */
                return;
        tmp = CAN1IDAC & 0x07;                  /* extract filter hit info */
        if (tmp == 0) {                         /* filter 0 hit */
                if (CAN1RDLR==0)                /* data length 0, do nothing */
                        return;
                cmd2lcd(0x80);                  /* set LCD cursor to first row */
                puts2lcd(t1Msg);                /* output "Temperature is" on LCD */
                cmd2lcd(0xC0);                  /* set LCD cursor to second row */
                ptr = (char *)&CAN1RDSR0;  /* ptr points to the first data byte */
                for (i = 0; i < CAN1RDLR; i++)
                        putc2lcd(*ptr++);       /* output temperature value on the LCD 2nd row */
        }
        else if (tmp == 1) { /* filter 1 hit */
                if(CAN1RDLR == 0)               /* data length 0, do nothing */
                        return;
                cmd2lcd(0x80);                  /* set LCD cursor to first row */
                puts2lcd(v1Msg);                /* output "Voltage is" on the 1st row of LCD */
```

```
            cmd2lcd(0xC0);              /* set LCD cursor to second row */
            ptr = (char *)&CAN1RDSR0;  /* PTR points to the first data byte */
            for(i = 0; i < CAN1RDLR; i++)
                putc2lcd(*ptr++);       /* output voltage value on the 2nd row of LCD */
        }
        else asm("nop");                /* other hit, do nothing */
}
```