CAN Products from National Semiconductor Users Guide and Application Note

National Semiconductor
Application Note-1074
Tobias Wenzel
Martin Embacher
June 1997

ABSTRACT

The intent of this application note is to provide the design engineer with a comprehensive guide to the implementation of the systems employing CAN products from National Semiconductor. The integrated programmable CAN interface block is available within the microcontrollers COP884BC and COP888EB. Additionally National provides a software which support a SLIO-CAN application based on the COP884BC. Further detailed informations about the SLIO application are summarized in another application note AN-1073, "SLIO-CAN, a CAN-linked I/O based on COP884BC".

This CAN interface is highly optimized for reduced die size and, hence, for low cost implementations. As a result of this optimization, the registers for CAN data communications have been reduced to two data transmit and two data receive registers. This implies that message transfers two data bytes are fully automatic when transmitted at bus speeds up to 1 Mbit/second with a CPU clock frequency of 10 MHz. For messages containing data longer than two bytes, receive/transmit buffer software handlers permit bus speeds up to 125 kBit/second.

The features of the COPCAN interface are summarized below:

FEATURES - COPCAN INTERFACE

- **conform with the CAN specification 2.0. part B (passive)**
  - — 8 byte data message transfer (up to 125 kbit/s)
  - — 2 byte data message transfer (up to 1 Mbit/s)
- **various bus configurations**
  - — differential bus mode
  - — single wire bus modes
- **built in reference voltage of $V_{CC}/2$**
- **Power Save operation**
- **wake up capability over the CAN bus**

1.0 PHYSICAL BUS INTERFACE

1.1 Physical Can Bus Interface

The physical bus connection of the CAN interface is supported with an on chip circuit. On the transmitter side there are two high current drive transistors. Each of those can be individually enabled, by setting the bits TXEN0 or TXEN1. Setting either bit will also enable the CAN interface. The resulting bus level is defined as shown in *Table 1*.

TABLE 1. Bus Level Definition

Bus Level	Pin Tx0	Pin Tx1
"dominant"	Drive Low (GND)	Drive High (V_{CC})
"recessive"	TRI-STATE®	TRI-STATE

On the receiver side there is a main receive comparator and two smaller wake-up comparators. Both inputs of the receive comparator can be routed to the internal voltage reference (V_{ref}) by setting the bits RXREF0 and RXREF1. With these functions both, the differential and two single wire bus modes are supported. *Figure 1* shows the internal circuit for the transmitter and receiver section. The two wake-up comparators are hardwired to V_{ref} on one terminal and to the pins RX0 and RX1 with the other terminal. Their outputs are combined to form the wake-up signal on an internal port pin. The respective wake-up enable bit for port L7 is fixed high and the wake up edge bit on the pin is fixed to falling edges on the COP884BC, hence the device will always wake-up with a recessive to dominate transition of the bus. To prevent errors caused by corrupted programs resetting bits this feature can not be disabled by software. On the COP888EB the wake up capability is programmable with the port M7 WKEN bit. With this versatile bus interface the device can be connected directly to the CAN bus for low-speed applications (<125 kbit/s) or using an external transceiver part for higher CAN bus speeds.

TRI-STATE® is a registered trademark of National Semiconductor Corporation.

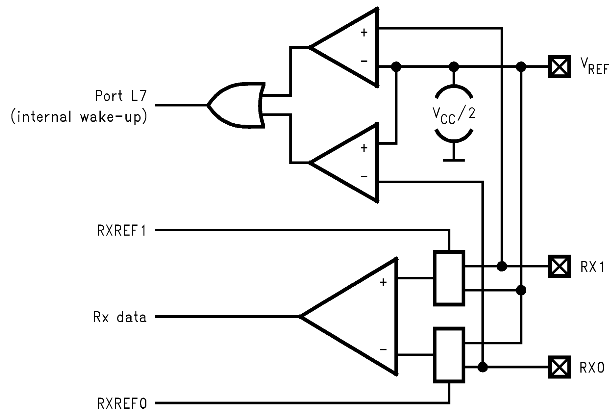


FIGURE 1. On-Chip CAN RxInterface

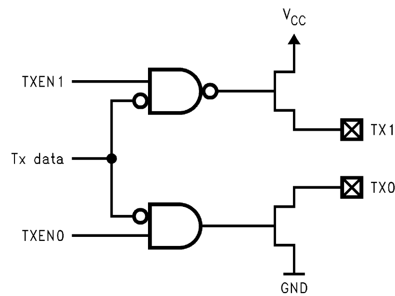


FIGURE 2. On-Chip CAN TX Circuit

1.2 Bus Interface Examples

The following section provides examples for different physical bus interfaces which can be used with the COPCAN interface. A software setup example is given for every applicable mode. Please note that all the software examples read the contents of the configuration register, modify it and then write the modified contents back. This is to ensure there is no intermediate bus mode selected.

1.2.1 ISO Low Speed Interface

The next section provides programming examples to set up the different bus modes if the interface is used with an ISO low speed interface as shown in *Figure 3*.

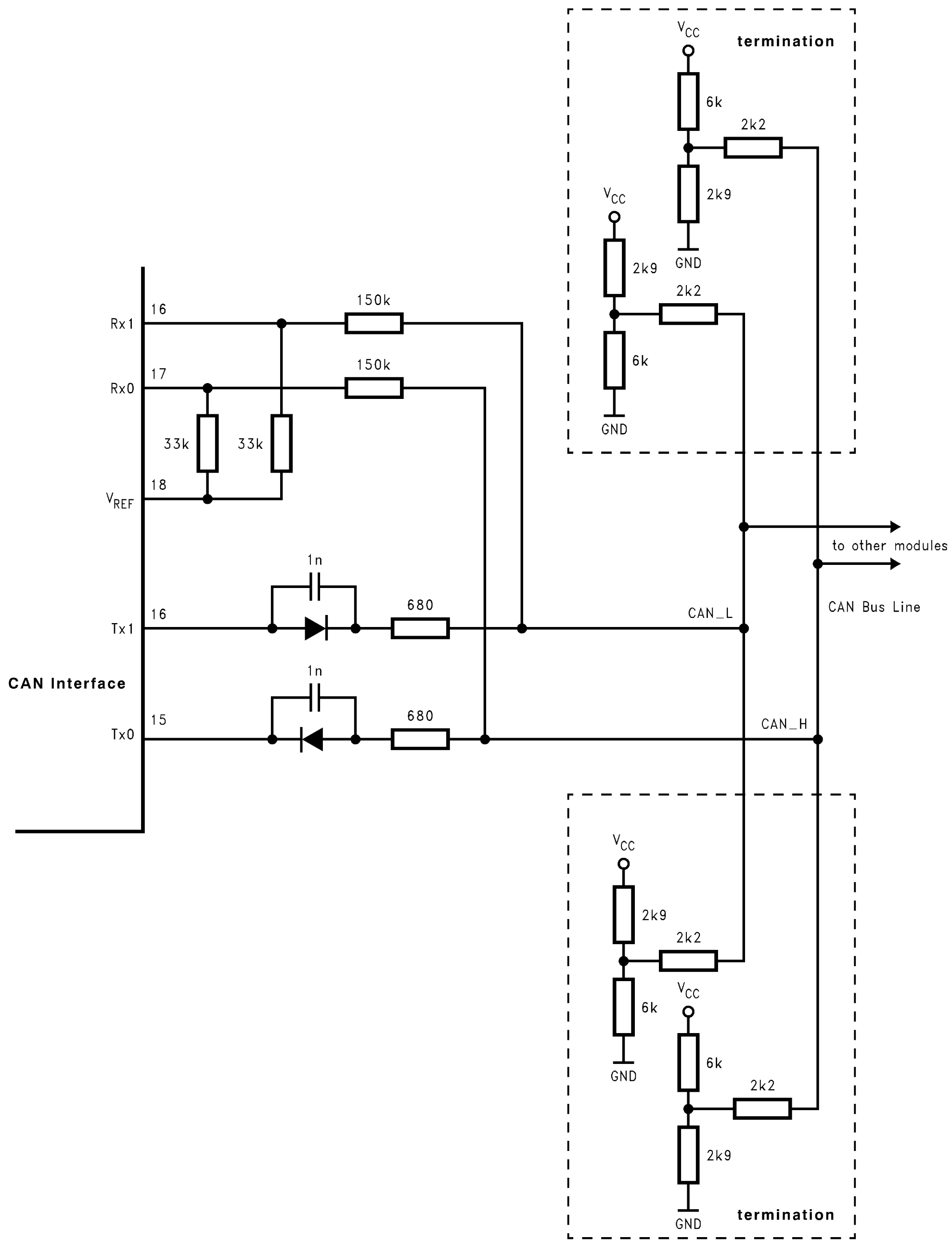


FIGURE 3. ISO Low Speed Interface (Switch Logic)

PROGRAMMING EXAMPLES OF THE ISO LOW SPEED INTERFACE

Example: Differential Mode TX0, TX1 and RX0, RX1—transmission on CAN_L and CAN_H

```
LD   B,  CBUS             ; point to CAN bus control register
LD   A,  [B]              ; get contents
AND  A,  #b'01000011      ; reset RXREF0, RXREF1
OR   A,  #b'00110000      ; set TXEN0, TXEN1
X    A,  [B]              ; re-write in one instruction
```

Example: Single Ended Mode TX0 and RX0—transmission on CAN_L only

```
LD   B,  CBUS             ; point to CAN bus control register
LD   A,  [B]              ; get contents
AND  A,  #b'01000011      ; reset TXEN1, RXREF0
OR   A,  #b'00011000      ; set TXEN0, RXREF1
X    A,  [B]              ; re-write in one instruction
```

Example: Single Ended Mode TX1 and RX1—transmission on CAN_H only

```
LD   B,  CBUS             ; point to CAN bus control register
LD   A,  [B]              ; get contents
AND  A,  #b'01000011      ; reset TXEN0, RXREF1
OR   A,  #b'00100100      ; set TXEN1, RXREF0
X    A,  [B]              ; re-write in one instruction
```

1.2.2 Inverted ISO Low Speed Interface

If the CAN interface is used with the ISO low speed mode and external drive transistors both single ended mode set-ups have to be inverted as shown below in *Figure 4*.

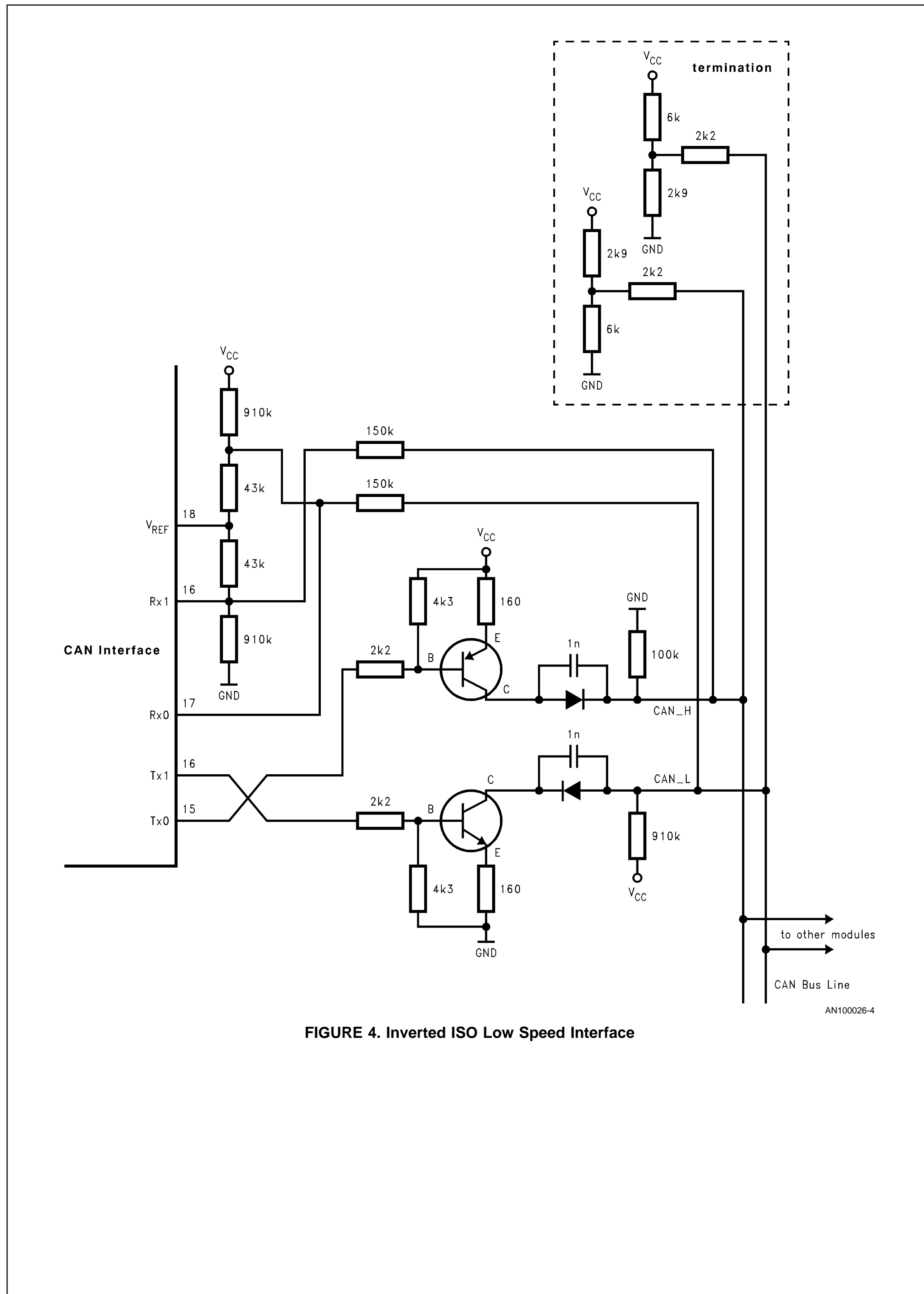


FIGURE 4. Inverted ISO Low Speed Interface

PROGRAMMING EXAMPLES OF THE INVERTED ISO LOW SPEED INTERFACE

Example: Differential Mode TX0, TX1 and RX0, RX1—transmission on CAN_L and CAN_H

```
LD   B,  CBUS           ; point to CAN bus control register
LD   A,  [B]            ; get contents
AND  A,  #b'01000011    ; reset RXREF0, RXREF1
OR   A,  #b'00110000    ; set TXEN0, TXEN1
X    A,  [B]            ; re-write in one instruction
```

Example: Single Ended Mode/TX1 and RX0 — transmission on CAN_L only

```
LD   B,  #CBUS          ; point to CAN bus control register
LD   A,  [B]            ; get contents
AND  A,  #b'01000011    ; reset TXEN0, RXREF0
OR    A,  #b'00101000   ; set TXEN01, RXREF1
X    A,  [B]            ; re-write in one instruction
```

Example: Single Ended Mode/TX0 and RX1—transmission on CAN_H only

```
LD   B,  #CBUS          ; point to CAN bus control register
LD   A,  [B]            ; get contents
AND  A,  #b'01000011    ; reset TXEN1, RXREF1
OR   A,  #b'00010100    ; set TXEN0, RXREF0
X    A,  [B]            ; re-write in one instruction
```

1.2.3 External Transceiver Chip

An external transceiver chip can be connected to the device as shown in *Figure 5*. Although the bus runs in differential mode the device is configured to run in single wire mode using TX0 and RX0. Care must be taken when using this type of interface together with the wake-up feature. Commonly the RX1 input of the CAN interface is connected to a Voltage reference of the transceiver chip. This, however, may lead to an unwanted setting of the L7 wake-up pending bit resulting from the connection of V_{ref} to both input terminals of one of the wake-up comparators terminals (see *Figure 1* for signal routing). This results in the device failing to go to IDLE or to HALT mode. For this reason the RX1 input must not be connected to the V_{ref} of the transceiver device - it should be terminated to GND with a pull-down resistor. Additionally, during high speed applications a pull up resistor must be connected to TX0. In low speed applications this resistor is not needed.

Using the configuration in *Figure 5* the CBUS register must be configured in Single Ended Mode for TX0 and RX0.

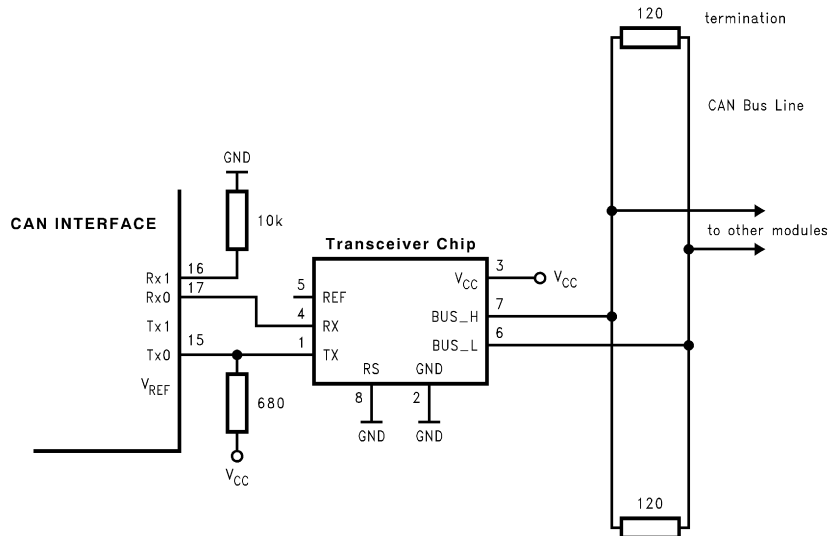


FIGURE 5. External Transceiver Connection (ISO High Speed)

PROGRAMMING EXAMPLES OF THE EXTERNAL TRANSCEIVER CHIP

Example: Single ended Mode TX0 and RX0

```
LD   B,  CBUS           ; point to CAN bus control register
LD   A,  [B]            ; get contents
AND  A,  #b'01000011    ; reset TXEN1, RXREF0
OR   A,  #b'00011000    ; set TXEN0, RXREF1
X    A,  [B]            ; re-write in one instruction
```

1.2.4 Bus Mode Switch Subroutine

For the ISO low speed interface as well as for the inverted ISO low speed interface a simple program can be written to allow switching bus modes in a subroutine. For the high speed interface (with an external transceiver chip) only two selections can be done as the external chip does not allow single wire operation. This example uses the ISO low speed interface, however adoptions to the other modes can be easily accomplished by modification. The desired bus mode is passed to the subroutine by the accumulator with the following values:

```
LD  A,#0   ; dual wire RX0, RX1, TX0, TX1
LD  A,#1   ; single wire RX0, TX0
LD  A,#2   ; single wire RX1, TX1
LD  A,#3   ; disable CAN interface
```

The program reads as follows.

```
.sect bus_mode, rom, inpage
      LD    B,  #CBUS      ; point to bus control register
      AND   A,  #003       ; mask unused bits
      ADD   A,  #low(b_tab); add jump table offset
      JID                  ; jump
b_tab:
      .db   dual_wire
      .db   single_rx0
      .db   single_rx1
      .db   no_wire
dual_wire:
      LD    A,  [B]             ; get contents
      AND   A,  #b'01000011     ; reset RXREF0, RXREF1
      OR    A,  #b'00110000     ; set TXEN0, TXEN1
      JP    end_bus_mode
single_rx0:
      LD    A,  [B]             ; get contents
      AND   A,  #b'01000011     ; reset TXEN1, RXREF0
      OR    A,  #b'00011000     ; set TXEN0, RXREF1
      JP    end_bus_mode
single_rx1:
      LD    A,  [B]             ; get contents
      AND   A,  #b'01000011     ; reset TXEN0, RXREF1
      OR    A,  #b'00100100     ; set TXEN1, RXREF0
      JP    end_bus_mode
no_wire:
      LD    A,  [B]             ; get contents
      AND   A,  #b'01000011     ; reset TXEN1, RXREF0
end_bus_mode:
      X     A,  [B]             ; write CBUS
      RET
.endsect
```

1.3 Bit Time Logic Setup

The bit time settings can be configured with the memory mapped register CSCAL, CTIM and with the clock frequency CKI. Hereby the synchronization jump width will be programmed through the length of the phase segments as it is described in *Table 2*.

TABLE 2. Synchronization Jump Width

PS2	PS1	PS0	Length of Phase Segment 1/2	Synchronization Jump Width
0	0	0	1 tcan	1 tcan
0	0	1	2 tcan	2 tcan
0	1	0	3 tcan	3 tcan
0	1	1	4 tcan	4 tcan
1	0	0	5 tcan	4 tcan
1	0	1	6 tcan	4 tcan
1	1	0	7 tcan	4 tcan
1	1	1	8 tcan	4 tcan

The resulting bus clock frequency can be computed by the formula below:

$$f_{bus} = \frac{CKI}{(1 + divider) \times (1 + 2 \times PS + PPS)}$$

1.3.1 Low Speed

CAN Low Speed is defined as a bus speed less than or equal to 125 kbit/s.

Example: calculation of 75 kbit/s (CKI = 5 MHz):

$$f_{bus} = \frac{CKI}{(1 + divider) \times (1 + 2 \times PS + PPS)}$$

CKI = 5 MHz

CSCAL = divider = 005_{hex}

CTIM = $02C_{hex}$ = 00101100_2

PPS = 2 = $(001_2 + 1_2)$

PS = 4 = $(011_2 + 1_2)$

$$f_{bus} = \frac{5\,MHz}{(1 + 5) \times (1 + 2 \times 4 + 2)} = 75{,}75\ kBaud$$

$$f_{min} = \frac{5\,MHz}{(1 + 5) \times (11 + 4)} = 55{,}56\ kBaud$$

$$f_{max} = \frac{5\,MHz}{(1 + 5) \times (11 - 3)} = 104{,}17\ kBaud$$

1.3.2 High Speed

CAN High speed is defined as a bus speed greater than 125 kbits and less than or equal to 1 Mbit/s.

Example: calculation of 500kbit/s:

$$f_{bus} = \frac{CKI}{(1 + divider) \times (1 + 2 \times PS + PPS)}$$

CKI = 10 MHz

CSCAL = divider = 000_{hex}

CTIM = $05C_{hex}$ = 01011100_2

PPS = 3 = $(010_2 + 1_2)$

PS = 8 = $(011_2 + 1_2)$

$$f_{bus} = \frac{10\,MHz}{(1 + 0) \times (1 + 2 \times 8 + 3)} = 500\ kBaud$$

$$f_{min} = \frac{10\,MHz}{(1) \times (20 + 4)} = 416{,}7\ kBaud$$

$$f_{max} = \frac{10\,MHz}{(1) \times (20 - 4)} = 625\ kBaud$$

2.0 CAN SOFTWARE DRIVER ROUTINES

2.1 Introduction

Due to the limitation imposed by having two receive and two transmit registers within the CAN interface the following receive/transmit driver routines are distinguished by:

- **2 byte message only routines up to 1 Mbit/s bus speed**
- **generic 0 to 8 byte message routines up to 125 kbit/s bus speed**

Usually, due to the asynchronous nature of CAN message reception, the receiver side interrupt routines are used. Two different receive interrupt examples are described for the receive process. On the transmitter side two byte messages do not need to be interrupt driven, because two byte transmittals are fully automatic. Therefore only the generic transmit routines use the transmit interrupt.

2.2 Transmit and Receive Object Definitions

The software routines described in the following section use different object types for the transmit and receive sides. This objects are able to handle the communication between the driver and the main routine. They consist of the identifiers, the data length code and the data bytes to be transmitted or received. Basically two different object types have been defined, one for messages of up to eight bytes and another for those up to two data bytes.

2.2.1 Message Objects of 2 Bytes and Less

Figure 6 describes the 2 bytes definition for the transmit/receive sides.

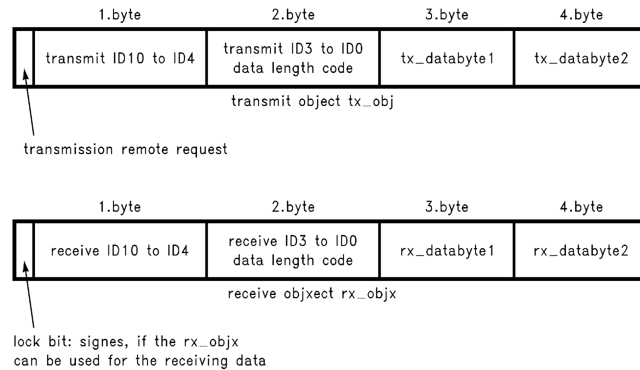


FIGURE 6. Message Objects of 2 Bytes and Less

Example: allocation of 2 byte message objects

```
.sect msg_buf, base
     tx_obj:     .dsb 4
     ; transmit object format:
     ; tx_obj[0] = trtr, tid[10:4]
     ; tx_obj[1] = tid[3.0], tdlc[3.0]
     ; tx_obj[2] = txd1 !
     ; tx_obj[3] = txd2 !
     rx_obj0:     .dsb 4
     ; rx_obj[0] = lock, rid[10:4]
     ; tx_obj[1] = rid[3:0], rdlc[3:0]
     ; tx_obj[2] = rxd1 !
     ; tx_obj[3] = rxd2 !
     rx_obj1:     .dsb 4
.endsect
```

2.2.2 Message Objects of 8 Bytes and Less

In addition to the allocation of 10 bytes for 8 data byte message objects a pointer (tx_ptr) is set to indicate the first byte of the transmit message object. Returned values of tx_ptr give information concerning the success of transmission.

```
tx_ptr = 0xFF→bus is busy
tx_ptr = 0x00→transmission done
tx_ptr = (other)→transmission still in progress
```

RAM location tx_data_start signifies the start of transmit data within the transmit object.

Example: allocation of Tx/Rx message objects

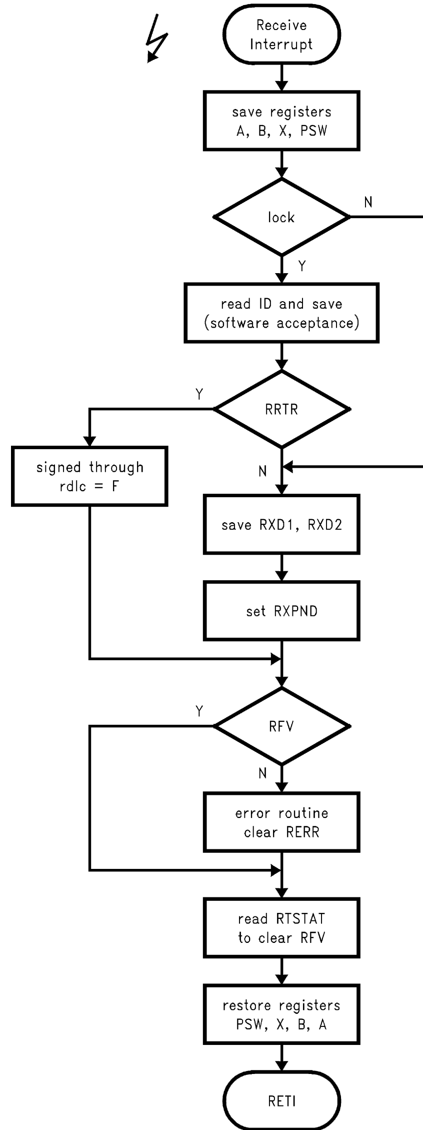
```
.sect msg_buf, ram
     tx_ptr:            .dsb 1   ; one pointer is required
     tx_data_start:     .dsb 1   ; one tx_start is required
     ; object format:
     ; obj[0] = rtr, id[10:4]
     ; obj[1] = id[3:0], dlc[3:0]
     ; obj[2:9] = data[1:8]
     tx_obj0:           .dsb 10 ; more object can be defined as needed
     rx_obj0:           .dsb 10
     rx_obj1:           .dsb 10
     rx_ptr:            .dsb 1
     rx_status:         .dsb 1
.endsect
.sect code_can_tx, rom
; this code transmits a 0 to 8 byte or remote CAN message
; from a transmit buffer tx_obj[0:9]
; this code works in conjuction with the TX interrupt
; which is used to copy more data bytes or indicate
; a successfull transmission
;
```

```
; parameters:
; tx_ptr:       pointer to transmit object
; tx_obj[0:9]   data to transmit
;
; return value:
; tx_ptr = 0xff - bus busy
; tx_ptr = 0x00 - transmission done (interrupt driven)
; tx_ptr = (other value) - transmission in progress
;
; registers used:
; A, X, B
```

2.3 Can Receive Routines

2.3.1 Two or Less Bytes Interrupt Receive - High Speed

Example: CAN Receiver Interrupt Routine (high speed) for messages 0 to 2 bytes including RTR



AN100026-7

Note 1: Lock indicates whether the receive buffer is free and will be cleared during program progress.

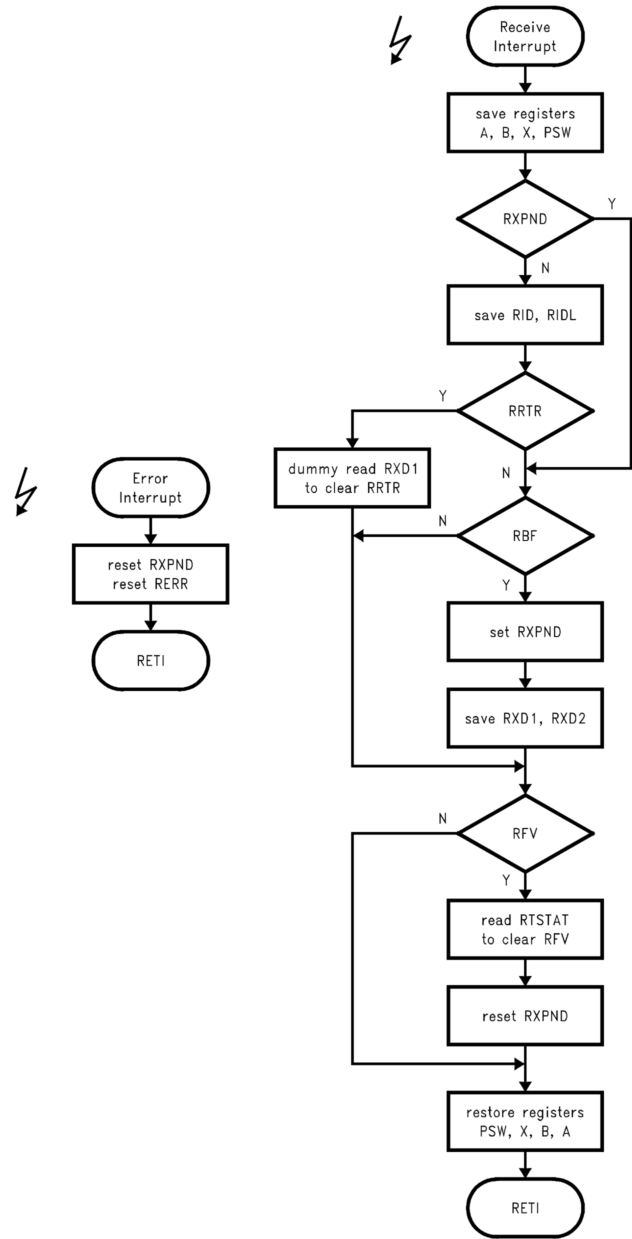
Note 2: The user has to provide a receive buffer in RAM with an even number of bytes greater than or equal to the maximum DLC of message to be received.

Note 3: Error check is done in this routine error interrupt must not be enabled.

```
.sect code_can_rx, rom      ; from interrupt
can_rx:
        ; this interrupt is triggerd by RBF, RRTR or RFV
        ; RRTR and RBF are cleared by reading or b's pointing to RXD1
        ; RFV is cleared by reading RTSTAT to A
        ; or executing the equiv. of LD B, #RSTAT; LD A, #xx
        ld      b, #rx_obj0   ; (*) receive id hi ; * only with RIAF = 0
        ifbit   7, [b]         ; buffer free
        jp      receive_msg    ; .. yes then receive
        ld      b, #rx_obj1    ; next buffer
        ifbit   7, [b]         ; buffer free
        jp      receive_msg   ; .. then receive msg
        jp      can_rx_exit   ; else exit
receive_msg:
        rbit    7, [b]
        ld      a, rid        ; (*) get received id
        ifne    a, [b]        ; (*) check if accept
        jp      can_rx_exit   ; (*) .. no then exit
        x       a, [b+]
        ld      a, ridl       ; get received IDLC
        x       a, [b]        ; save message
        ifbit   RRTR, RTSTAT   ; received frame remote frame?
        jp      can_rx_rtr    ; yes
        jp      save_data     ; no
can_rx_rtr:
        ld      a,[b]         ; remote frame is signed
        or      a,#0F         ; through rdlc = F
        x       a,[b]         ;
        jp      wai_rx        ;
save_data:
     ld      a,[b+]           ;dummy read →point rx_data register
        ld      a, RXD1       ;
        x       a, [b+]
        ld      a, RXD2
        x       a, [b]
        ld      b, #RTSTAT
wait_rx:
        ifbit   RFV, [b]
        jp      rx_done
        ifbit  RERR, TCNTL
        jp      rx_error
        jp      wait_rx
; this is the error routine error interrupt must not be enabled
rx_error:
        ld      b, #rx_obj1
        ifbit   7, [b]
        jp      check_obj0
        jp      end_error
check_obj0:
        ld      b, #rx_obj0
end_error:
        sbit    7, [b]         ; free buffer
        rbit    RERR, TCNTL
rx_done:
can_rx_exit:
        ld      a, RXD1        ; dummy read to clear RBF, RTR
        ld      a, RTSTAT      ; dummy read to clear RFV
        jp      int_end
.endsect
```

2.3.2 Generic Interrupt Receive

Example: CAN Receiver Interrupt Routine for messages 0 to 8 bytes including RTR



Note 1: RXPND is a software flag controlled by the user.

Note 2: The user has to provide a receive buffer in RAM with an even number of bytes greater than or equal to the maximum DLC of message to be received.

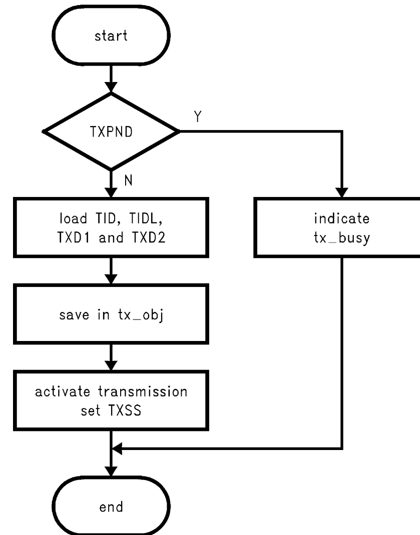
Note 3: No error check is done in this routine as a receive error will automatically generate an interrupt (if enabled) right after the receive interrupt.

2.4 CAN Transmit Routines

2.4.1 Two or Less Bytes Transmit (High Speed)

Example: CAN Transmit Routine for messages 0 to 2 bytes and remote frames

This code transmits a 0 to 2 byte or remote CAN message from a transmit buffer tx_obj[0:3]. The routine intentionally does not check for remote or DLC (data length code) as the COPCAN interface will automatically transmit no data bytes in a remote frame and transmit not more than DLC data bytes. If the CAN transmit is working, the routine will be finished without any actions.

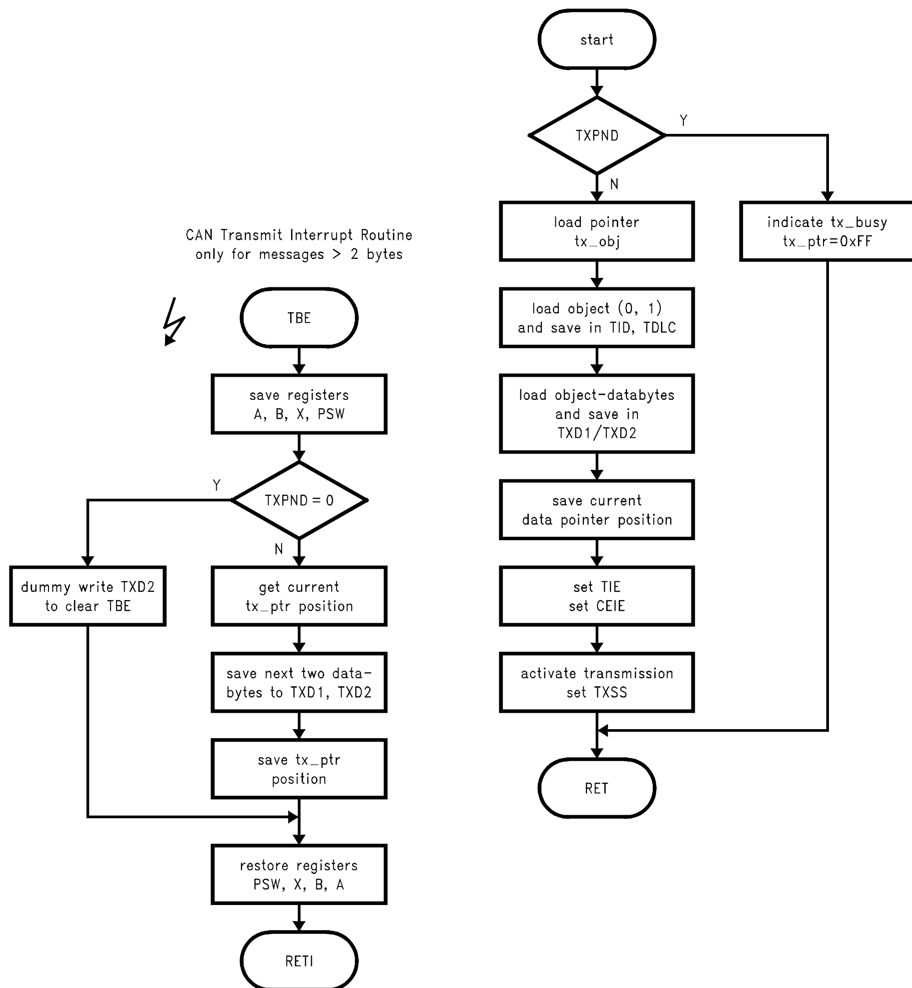


Note: Errors are signed with the carry flag (or any other flag!) and have to handle separately.

```
.sect   code_can_tx, rom
can_tx:
        rc                          ; (*) reset error flag
        ifbit   TXPND, RTSTAT       ; check if transmit busy
        jp      tx_busy             ; .. yes then exit
        ld      b, #tx_obj          ; point to tx_obj[0]
        ld      x, #TID             ; point to TID
        ld      a, [b+]             ; get tx_obj[0]; point to tx_obj[1]
        x       a, [x-]             ; .. and save
        ld      a, [b+]             ; get tx_obj[1]; point to tx_obj[2]
        x       a, [x-]             ; .. and save
        ld      a, [b+]             ; point to tx_obj[3]
        ld      a, [b-]             ; get tx_obj[3]; point to tx_obj[2]
        x       a, [x-]             ; save to TXD1
        ld      a, [b]              ; get tx_obj[3]
        x       a, [x]              ; save to TXD2
tx_done:
        sbit    txss, tcntl         ;set pending transmission
        ;automatic reset of txss after transmission
        ret                         ; exit without error
tx_busy:
        sc                          ; (*) indicate tx_busy
        ret                         ; (*) exit with error
        ; retsk                     ; optional use retsk instead
                                    ; 1st and last 2 lines to skip next
.endsect
```

2.4.2 Generic Interrupt Transmit (Low Speed)

This code transmits a 0 to 8 byte or remote CAN message from a transmit buffer tx_obj[0:9] this code works in conjunction with the TX interrupt which is used to copy more data bytes or indicate a successful transmission.



Note: No error check is done in this routine as a transmit error will automatically generate an interrupt (if enabled) right.

```
.sect    code_can_tx, rom
;
; parameters:
; tx_ptr:        pointer to transmit object
; tx_obj[0:9]    data to transmit
;
; return value:
; tx_ptr = 0xff - bus busy
; tx_ptr = 0x00 - transmission done (interrupt driven)
; tx_ptr = (other value) - transmission in progress
;
; registers used:
; A, X, B
can_tx:
        ifbit   TXPND, RTSTAT     ; check if transmit busy
        jp      tx_busy           ; .. yes then exit
        ld      a, tx_ptr         ; get tx pointer
        x       a, b              ; save to b
        ld      x, #TID           ; point to TID
        ld      a, [b+]           ; get tx_obj[0]; point to tx_obj[1]
        x       a, [x-]           ; .. and save
        ld      a, [b+]           ; get tx_obj[1]; point to tx_obj[2]
        x       a, [x-]           ; .. and save
        ld      a, b              ; get data start value
        x       a, tx_data_start  ; save
        ld      a, [b+]           ; point to tx_obj[3]
        ld      a, [b-]           ; get tx_obj[3]; point to tx_obj[2]
        x       a, [x-]           ; save to TXD1
        ld      a, [b+]           ; get tx_obj[3]
        x       a, [x]            ; save to TXD2
        ld      a, b              ; get pointer value
        inc     a
        x       a, tx_ptr         ; save to pointer
        ld      b, #TCNTL
        sbit    TIE, [b]          ; enable transmit interrupt
        sbit    CEIE, [b]         ; enable CAN error interrupt
        sbit    TXSS, [b]         ; start transmission
        ret                       ; exit
tx_busy:
        ld      tx_ptr, #0ff      ; indicate bus busy
        ret                      ; exit
; interrupt driven CAN transmit routine
; assumes A, X and B register can be used
can_txint:
        ld      b, #RTSTAT        ; temporary (!) point to RTSTAT
        ifbit   TXPND, [b]        ; transmission done
        jp      tx_bytes          ; .. no then continue
end_txint:                        ; .. yes then exit
        ld      b, #TXD2          ; point to TXD2
        ld      TXD2, #0          ; dummy write to clear TBE
        ld      tx_ptr, #0        ; indicate transmission done
        rbit    TIE, TCNTL        ; disable interrupt
        jp      int_end           ; global RETI
tx_bytes:
        ld      a, tx_ptr         ; get current pointer
        x       a, x              ; put to x
        ld      b, #TXD1
        ld      a, [x+]           ; get next odd data byte
        x       a, [b+]           ; save to TXD1
        ld      a, [x+]           ; get next even data byte
        x       a, [b]            ; save to TXD2
        ld      a, x              ; get current pointer
        x       a, tx_ptr         ; save pointer
        jp      int_end           ; global RETI
; CAN transmit error routine
; assumes CAN error routine:
;
```

```
tx_error:
        ld      a, tx_data_start  ; get data start
        x       a, x              ; copy to x
        ld      b, #TXD1          ; point to TXD1
        ld      a, [x+]           ; get 1st data byte
        x       a, [b+]           ; save in TXD1
        ld      a, [x+]           ; get next data byte
        x       a, [b]            ; save in TXD2
        ld      a, x              ; get tx pointer
        x       a, tx_ptr         ; save
        rbit    TERR, TCNTL       ; clear pending
        jp      int_end           ; global RETI
.endsect
;============================================================================
.sect   code_can_error, rom
; interrupt driven CAN error routine. this code checks
; for standard errors and goes to the respective interrupt
; routine
; an error is assumed "standard" for a transmitter if the
; transmit error bit is set (TERR = 1) and the
; transmit interrupt is enabled (TIE = 1) and the
; transmission is in progress (TXSS = 1)
;
; an error is assumed to be "standard" for a receiver if the
; receive error bit is set (RERR = 1) and the
; receive interrupt is enabled (RIE = 1)
;
can_error:
        ld      b, #TCNTL
        ifbit   TERR, [b]
        jp      tx_error          ; .. then tx error
        ifbit   RERR, [b]
        jp      rx_error          ; .. then rx error
                        ; else there is no standard error
                        ; but the device could be bus-off
rx_error: ; temp
        rbit    TERR, [b]         ; reset TERR pending
        rbit    RERR, [b]         ; reset RERR pending
        jp      int_end           ; golbal RETI
.endsect
```

2.5 Acceptance Filter

Because CAN is a message orientated system, the identifiers in the CAN frame stand for the type of the message. Without acceptance, filtering all types of messages are processed by all nodes connected on the CAN bus. Therefore every CAN node must determine whether to process a message or not. In order to reduce the software expense, the CAN interface supports a hardware acceptance filtering of the upper identifiers ID4 to ID10. The CAN interface provides the capability to mask these identifiers through hardware. Masking enabled, if the RIAF bit (CBUS register) is set to zero. If enabled, the RID register is compared with the received identifiers, ID4 to ID10, as shown in *Figure 7*.

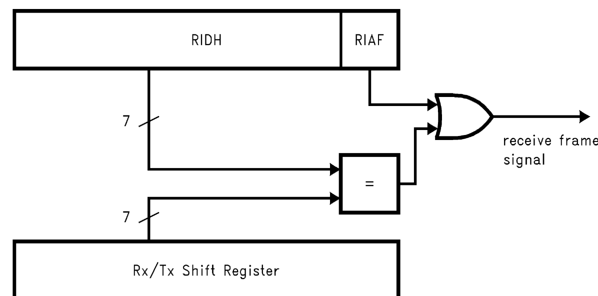


FIGURE 7. Acceptance Filter Block Diagram

The lower 4 identifiers ID0 to ID3 can not be masked by the acceptance filter. This means that, should the acceptance filter be enabled, 16 different messages will always be accepted.

Example: acceptance of the identifiers 730 h to 73F h

```
can_rx:
acc_filter:
      ld    B,#CBUS
      sbit  6,[B]
      ld    RID,#073
  ;If the RIAF bit is set to one, all messages on the bus will be processed
```

Example: acceptance of the identifiers 700 h to 73F h

```
can_rx:
acc_filter:
      ld    A, RID
      and  A, #b'01111100
      ifne  A, #b'01111100
      jp    end_can
```

3.0 USAGE OF THE SLIO

3.1 SLIO Registers

In the SLIO module concept the SLIO memory contains several parameter defined registers (see *Table 3*), which can be configured by messages sent over the CAN bus or by an external EEPROM during the initialization phase. The configuration of the Multi-I/O Function Block, the power save conditions and the bus mode can be set and altered by the parameters in the Register Block. The Identifiers and the CAN prescaler are configured via the Identifier port.

TABLE 3. SLIO Register Block

Register Marker (hex)	Name	Function	Message type	config over CAN	config over EEPROM
0x00	1N1	read status P0 to P7	r	read only	no
0x01	PE	config P0 to P7 positive edge	r/w	yes	yes
0x02	NE	config P0 to P7 negative edge	r/w	yes	yes
0x03	OD1	write data to P0 to P7	r/w	yes	yes
0x04	DD1	config P0 to P7 data direction	r/w	yes	yes
0x05	IN2	read status P8 to P13	r	read only	no
0x06	OD2	write data to P8 to P13	r/w	yes	yes
0x07	DD2	config P8 to P13 data direction	r/w	yes	yes
0x08 to 0x0E	ADC	read specified analog input	r	read only	no
0x0F	IN1	reset status register marker point to IN1	r	read only	no
0x10 to 0x13	DAC1/DAC2	config analog output	r/w	yes	yes
0x1C	ACT	analog output control	r/w	yes	yes
0x1E	CCT	comparator control	r/w	yes	yes
0x1F	CTR	configuration register	r/w	yes	yes

3.2 CAN Message Format

CAN messages to and from SLIO are limited to two byte messages. The first databyte is reserved for the register marker and system information. The register marker can be considered as a pointer of the specialized SLIO register, which should be changed through the data of the second databyte. The upper 3 bits of the first databyte include information about the bus mode of the SLIO and give information about the CAN Error status of the SLIO. The content of the two data bytes and from the control field is shown in *Table 4*.

TABLE 4. SLIO Frame Format

DLC = 2	ST	BM		RM					data
		1	0	4	3	2	1	0	

ST CAN Error Status of the SLIO

0 = error active

1 = the device became error passive since the last frame transmitted by the SLIO.

BM[1:0] Current Bus mode

0 = dual wire

1 = single wire RX0

2 = single wire RX1

3 = not allowed

RM[4:0] Register marker bits

Example: Status message

Status messages are created from SLIO without any demand messages from CAN. These messages are transmitted after the following actions:

1. initialization is finished
2. external event on the pins P0 to P7 (if they are enabled by PE or NE register)
3. awakening from NAP/SLEEP mode

This message contains the status of the pins P0 to P7. The content of *Figure 8* describes a status message.

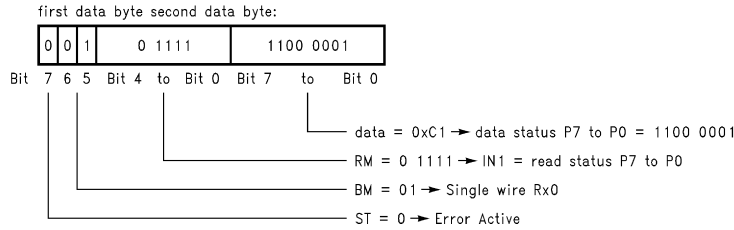


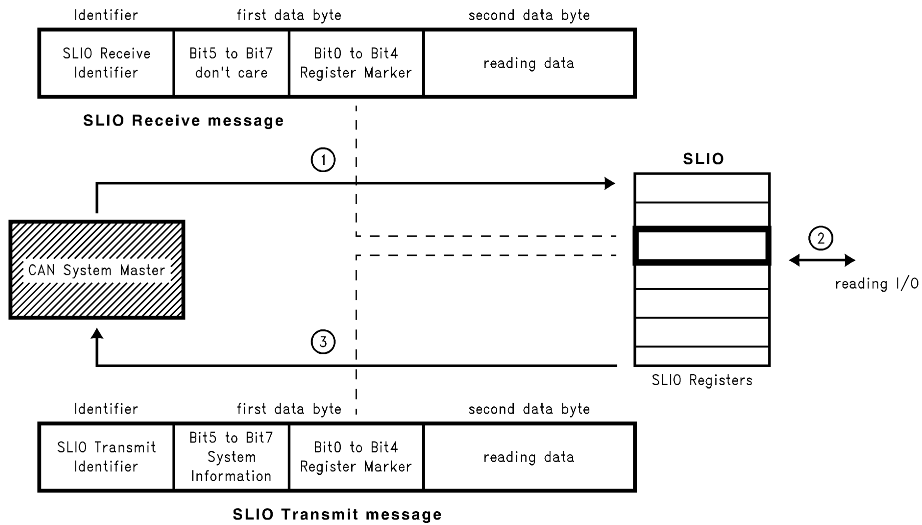
FIGURE 8. Status Message

3.3 CAN System Master Communication

Communication between the SLIO and the CAN System Master is achieved through query and response in addition to those messages which are initiated from the SLIO as a result of interrupts and wake-up conditions. There are two different message types, read only and read/write.

3.3.1 Read Message Transfer

Read Only messages demand the status of digital and analog pins (see *Table 4* SLIO register block). *Figure 9* shows the data transfer of read only messages between CAN System Master and the SLIO device.



The following steps are executed:

1. SLIO receives a read message from System Master with correct Receive Identifier and with the correct Read register marker.
2. The SLIO reads the port pins or the analog input pin and write the reading data in the specified SLIO Register.
3. After the reading process the SLIO creates a message with its transmit ID and the reading data.

FIGURE 9. CAN Communication with Read Messages

Example: Read Pin P0 to P7

This example describes reading the digital status of pin P0 to P7 over CAN by the following configuration. The SLIO is configured in Single Wire Rx0/TX0 Bus mode and the error state of the SLIO is Error Active. The Identifiers are configured in Pin Mode as follows:

ID0 = GND; ID1 = GND; ID2 = GND; ID3 =GND

ID configuration: Transmit ID = 0400 H (from the SLIO)

Receive ID = 0401 H (to the SLIO)

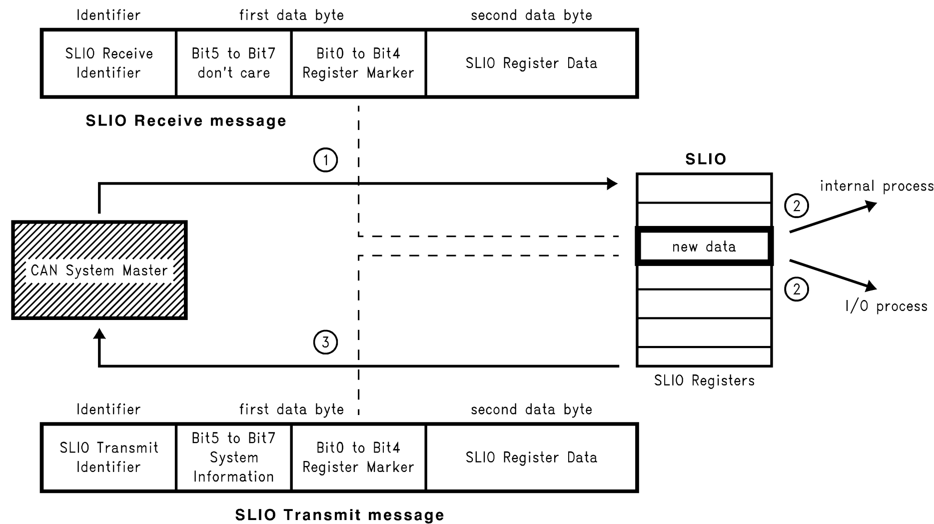
The data of the SLIO pins are 0xC0. In *Table 5* the data transfer between CAN System Master and the SLIO is monitored.

TABLE 5. Read Transfer Example

Name	ID	databyte1	databyte2
receive message	0401 H	00 H	don't care
transmit message	0400 H	20 H	C0 H

3.3.2 Read/Write Message Transfer

Read/write transfer updates the configuration data within the SLIO register block by writing data into the specified register as indicated by the register marker in the first data byte. The response is a read from the specified data register subsequent to the update process. The data transfer is shown schematically in *Figure 10*.



The following steps are executed:

1. The SLIO receives a read/write message from System Master with correct Receive Identifier, with the correct Read register marker and the new data placed in the second databyte.
2. The SLIO changes the configuration of the internal condition or, optionally, of the I/O pin configuration.
3. After this process the SLIO creates a response message with its transmit ID and the new status of the specified SLIO register. This informs the System Master that the message data transfer was executed correctly.

FIGURE 10. CAN Communication with Read/Write Messages

Example: Configuration of Data Direction P0 to P7

The data direction of the SLIO pins P0 to P7 should be changed via CAN. It will be assumed that the configuration of the Identifier and Bus mode is the same as in the Read Only example.

The data direction of the pins should be changed as follows:

P0 to P3 as input

P4 to P7 as output

In *Table 6* the read/write data transfer between CAN System Master and SLIO is monitored.

TABLE 6. Read/Write Data Transfer Example

Name	ID	databyte1	databyte2
receive message	0401 H	00 H	F0 H
transmit message	0400 H	20 H	F0 H

3.4 CAN System Master in CAN - SLIO Network

In the CAN network at least one node is assigned as the CAN System Master. The system master handles the communication with the connected SLIO nodes. Each of these nodes must have two sequential IDs, an even one for transmit and an odd one for receive. In addition, when using EEPROM mode, a global receive ID may be defined.

An example of one CAN-message address space is shown in the example *Figure 11*.

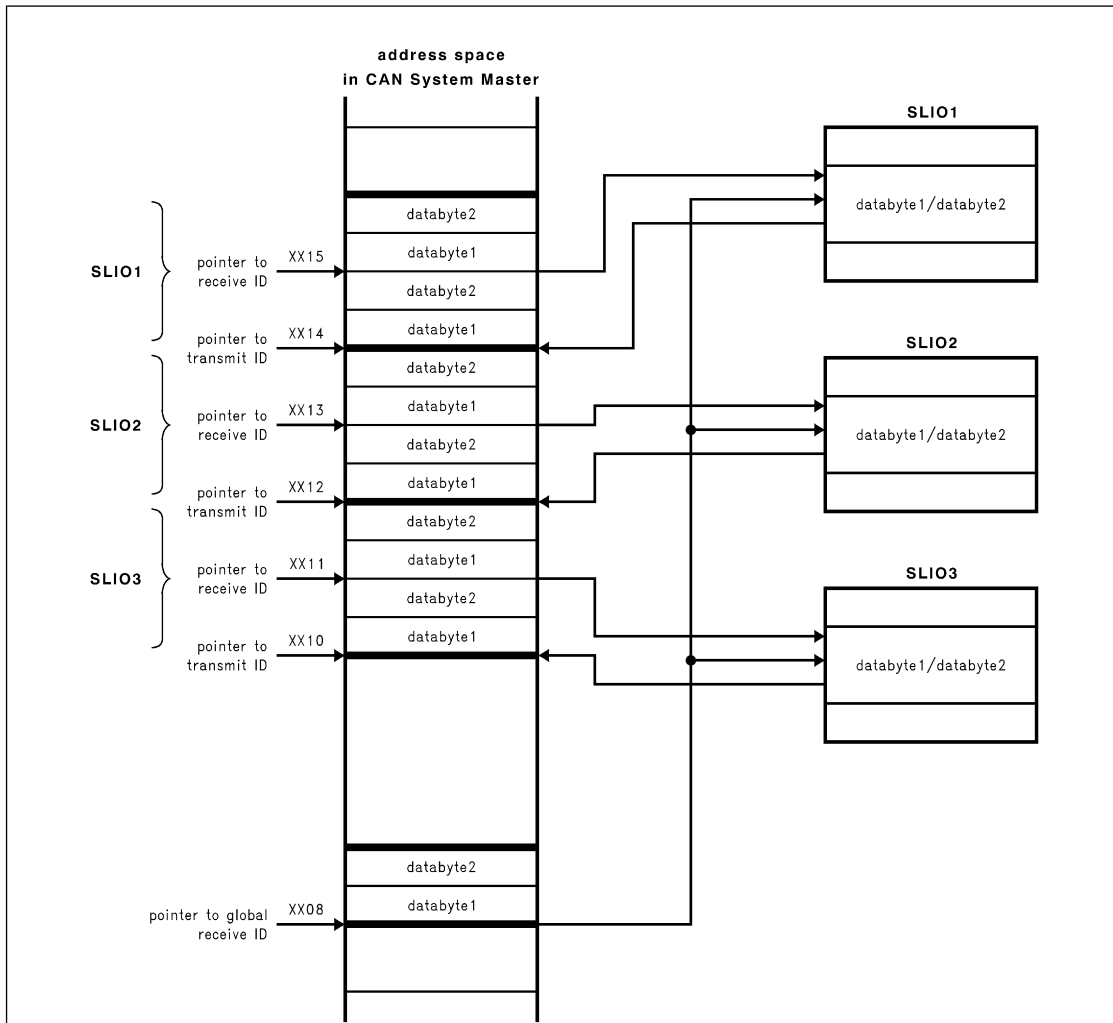


FIGURE 11. Example - CAN System Master Address Space

3.5 SLIO System

Figure 12 shows the schematics of a CAN SLIO node. The pins P0 to P13 can be connected with Sensors or Actuators over the I/O Feature Connector. The Power Supply circuit with LM2925 generates first, V_{CC} = 5V and second, the external RESET. V_{IN} can be obtained from an external source of 6V < V_{IN} < 26V or from the CAN wiring system. Because of the external crystal oscillator on the pins CKI and CKO, the SLIO does not need synchronization messages from the System Master. This device has Master capabilities, which means that it can synchronize itself to the CAN bus.

The two signal wires of CAN, BUS_H and BUS_L, are connected with the integrated CAN interface of the SLIO over the Physical Bus interface. The bus timing programmability of the SLIO CAN interface is limited with the exception of the CAN prescaler and CKI. Refer to section one for circuit description of the interfaces.

During the initialization phase the SLIO application reads the identifiers from the identifier circuit over the pins ID0 to ID3. There are two different capabilities to read the identifier. Two means of determining the identifier exist; direct pull up/down of the ID pins or via EEPROM.

AN100026-16

FIGURE 12. SLIO CAN Node

3.6 Initialization

3.6.1 General Description

There are two different modes of reading the CAN identifier. In Pin mode four, the 11 available identifier bits may be read from ID0–3, the remainder of the bits are predetermined, thus 16 different SLIO nodes may be connected to one CAN network system. To avoid this restriction, an EEPROM(NMC93C06) mode may be used on the ID port allowing programming of 11 of the identifier bits. This means that the ID port is internally connected to the MICROWIRE interface. Furthermore, the use of an EEPROM allows the programming of the Register Block and avoids the necessity of having the system master transmit the setup configuration when the node is brought on line.

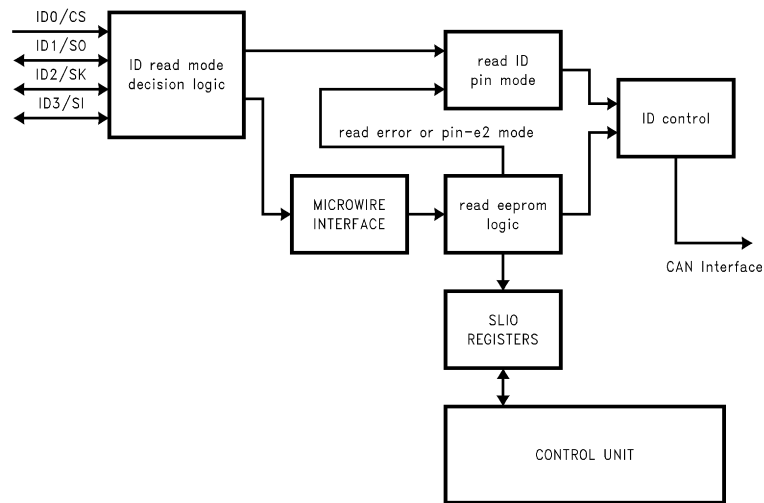


FIGURE 13. Read CAN Identifier Port

A mixture between EEPROM mode and pin mode was implemented as Pin_E2 mode. In this Pin_E2 mode, the identifier are read in pin mode and the configuration of the SLIO registers can be read from the external EEPROM. In this case, the information to go in Pin_E2 mode is given from EEPROM in *Figure 13*. At the end of the initialization phase the SLIO will transmit a status message. After this message the SLIO is ready to communicate with the CAN bus.

3.6.2 Initialization Example in Pin Mode

In this section an example to initialize the SLIO in Pin Mode is shown. Initialization means on the one hand to program the CAN Identifier and on the other hand the configuration of the SLIO Registers.

Example: CAN Identifier Programming - Pin Mode

In Pin Mode, the CAN Identifier is programmed through pull up/pull down resistors on the Identifier Port pins ID0 to ID3. This means that 4 CAN Identifiers can be programmed and so 16 different SLIO Nodes can be connected on the CAN bus. *Figure 14* shows a connection example.

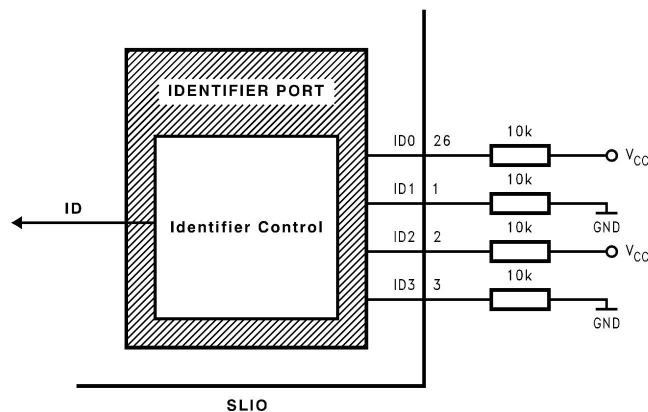


FIGURE 14. Identifier Configuration in Pin Mode

The result of this programming is shown in *Table 7*, columns ID3, ID2, ID1 and ID0.

TABLE 7. SLIO CAN Identifiers in Pin Mode

ID - Name	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	DIR
transmit ID (0x414 h)	1	0	0	0	0	0	1	0	1	0	0
receive ID (0x415 h)	1	0	0	0	0	0	1	0	1	0	1

In Pin Mode, the SLIO Registers are fixed to default values. They cannot be configured during initialization phase. The bus rate is fixed to CKI/40 and the bus mode is automatic. That means that the device cycles through all bus modes if no message is received for 8*Bt.

In *Table 8* the default SLIO Register values are shown.

TABLE 8. Configuration of the SLIO Registers in Pin Mode

Name	Function	configured after Initialization (hex)
IN1	read status P0 to P7	cannot be configured
PE	config P0 to P7 positive edge	0x00
NE	config P0 to P7 negative edge	0x00
OD1	write data to P0 to P7	0xFF
DD1	config P0 to P7 data direction	0x00
IN2	read status P8 to P13	cannot be configured
OD2	write data to P8 to P13	0xFF
DD2	config P8 to P13 data direction	0x00
ADC	read specified analog input	cannot be configured
DAC1	config analog output	0x00
DAC2	config analog output	0X00
ACT	analog output control	0x00
CCT	comparator control	0x00
CTR	configuration register	0x00

3.6.3 Initialization Example in EEPROM Mode

In EEPROM mode, all bits of the CAN standard identifier are programmable and each of the SLIO register, as well as the CAN prescaler register, may be configured separately. Prior to reading the EEPROM the CS(ID0) pin must be held low to prevent interference with any other microwire users available to the node. If an EEPROM is connected to the SLIO for purposes of programming the identifier and registers, the first location must read an AA hex value. If the value is other than AA hex, the device assumes the EEPROM is for purposes other than of programming the SLIO. When AA is detected in EEPROM location E2-MASK, the data contained in the EEPROM is transferred to the internal registers of the SLIO.

Example: CAN Identifier Programming EEPROM Mode

In *Figure 15* the connection between EEPROM NMC93C06 and the Identifier port is shown.

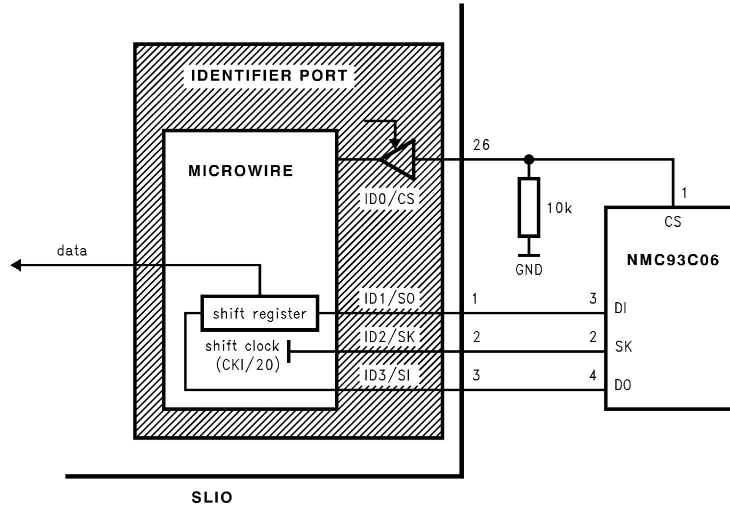


FIGURE 15. EEPROM Connection

The programming of the identifier ID0 to ID6 will be done by the EEPROM register RXIDL. DIR (Bit0) of this register can not be configured, it is don't care. The data direction will be configured automatically. ID7 to ID9 are configured with the EEPROM register RXIDH. A configuration example of the identifier through RXIDH/RXIDL are shown in *Table 9*.

TABLE 9. Receive/Transmit Identifier Programming with EEPROM

CID10	CID9	CID8	CID7	CID6	CID5	CID4	CID3	CID2	CID1	CID0
ID9	**ID8**	**ID7**	**ID6**	**ID5**	**ID4**	**ID3**	**ID2**	**ID1**	**ID0**	**DIR**
1	1	1	0	0	0	0	1	0	1	-
RXIDH			RXIDL							

The result of programming receive and transmit Identifiers, which are programmed by RXIDH and RXIDL, are shown below:

Transmit ID→0x70A Receive ID→0x70B

The programming of the global identifier ID0 to ID6 will be done by the EEPROM register RXIDGL. DIR (Bit0) of this register is set to 1 (receive data direction) automatically. ID7 to ID9 are configured with the EEPROM register RXIDGH. A configuration example is shown in *Table 10*.

TABLE 10. Global Identifier Programming

CID10	CID9	CID8	CID7	CID6	CID5	CID4	CID3	CID2	CID1	CID0
ID9	**ID8**	**ID7**	**ID6**	**ID5**	**ID4**	**ID3**	**ID2**	**ID1**	**ID0**	**DIR**
1	1	1	0	0	0	0	0	0	0	-
RXIDGH			RXIDGL							

The result of programming global receive Identifiers, which are programmed by RXIDGH and RXIDGL, are shown below:

Global Receive ID→0x701Transmit ID→0x70A

In *Table 11* a configuration example of the external EEPROM register settings are shown.

TABLE 11. Example of Configuration of SLIO Registers in EEPROM Mode

E2-address	EEPROM Registers	SLIO Registers
0x00	**E2-MASK** 0xAA	
	PIN-E2-MASK 0x00	
0x01 0x02	**RXIDH/RXIDL** **RXIDGH/RXIDGL**	
0x03	**PEDGE** 0xF0	**PE** 0xF0
	NEDGE 0x0B	**NE** 0x0B
0x04	**ODATA1** 0x01	**OD1** 0x01
	ODATA2 0x00	**OD2** 0x00
0x05	**DATADIR1** 0x00	**DD1** 0x00
	DATADIR2 0x00	**DD2** 0x00
0x06	**DACH** 0x01	**DAC2** 0x01
	DACL 0xB0	**DAC1** 0xB0
0x07	**ACR** 0x03	**ACT** 0x03
	CCR 0xE0	**CCT** 0xE0
0x08	**DCR** 0x08	**CTR** 0x08
	CAN_PSC 0x03	**CAN Prescaler** 0x03

3.6.4 Initialization Example in Pin_E2 Mode

If the E2 register PIN_E2 MASK is programmed with 0x55, the Pin_E2 mode is enabled. This allows the reading of the SLIO default values from the EEPROM and the Identifiers ID1 to ID3 in Pin mode. The pin ID0/CS can not be used for Identifier programming, because this pin needs a pull down resistor for the reading process of the EEPROM. Therefore, in Pin_E2 mode, only eight different Identifier can be configured.

Example: Initialization in Pin_E2 mode

In *Figure 16* the connection between EEPROM NMC93C06, pull up/pull down resistors and Identifier port are shown.

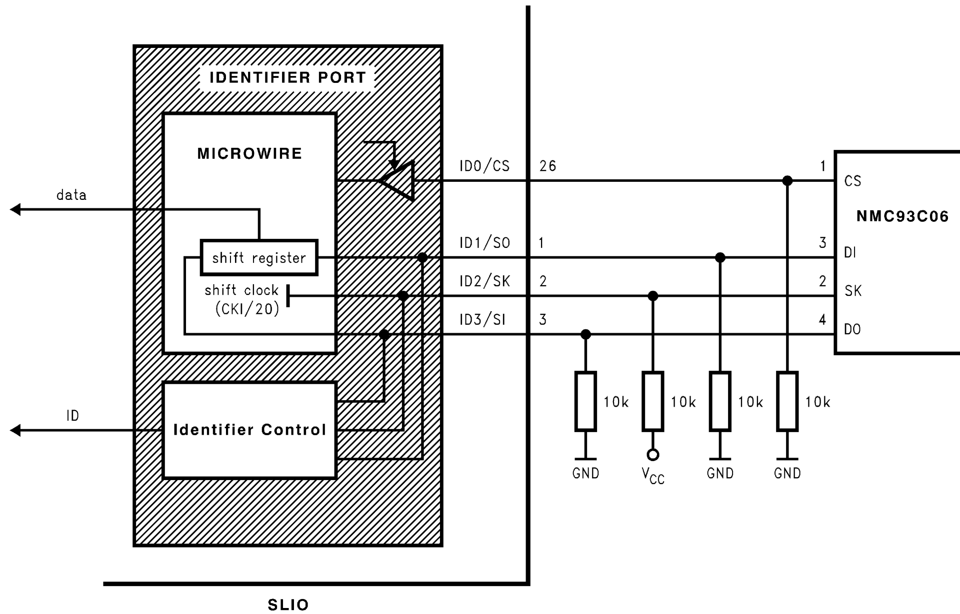


FIGURE 16. Pin_E2 Configuration

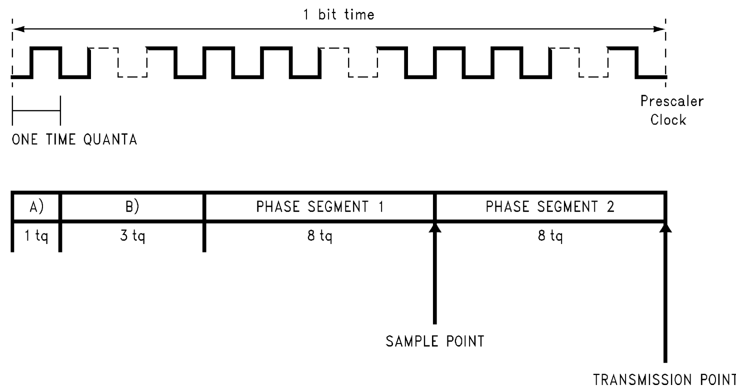
The result of this programming is shown in *Table 12*. Hereby the identifier ID0 has always low level. Therefore in Pin_E2 mode only ID1 to ID3 can be configured over pull up/down resistors. ID4 to ID9 can be configured over EEPROM.

TABLE 12. SLIO CAN Identifiers in Pin_E2 mode

ID - Name	ID9	ID8	ID7	ID6	ID5	ID4	ID3	ID2	ID1	ID0	DIR
transmit ID (0x408 h)	0	1	1	0	0	0	0	1	0	0	0
receive OD (0x409 h)	0	1	1	0	0	0	0	1	0	0	1
	configurable over EEPROM						configurable over resistors			fix to 0	

3.6.5 CAN Bus Rate Configuration

In EEPROM Mode and in Pin_E2 Mode the bus rate of the SLIO can be configured by EEPROM register CAN_PSC and CKI. In pin mode the bus rate is fixed to CKI/40. Furthermore the segments of one bit time are predefined as described in *Figure 17*. This means, that the sample point is fixed to 60% up to 500 kbit/s bus rate and to 80% using 1 Mbit/s. Hereby the synchronization jump width is configured to 4 time quanta up to 500 kbit/s and 2 time quanta using 1 Mbit/s.



AN100026-21

A) synchronization segment
B) propagation segment

FIGURE 17. Bit Timing up to 500 kbit/s

Example: bus time configuration - EEPROM Mode/ Pin_E2 Mode

If EEPROM Mode/Pin_E2 Mode is used, the bus rate can be configured with the CAN_PSC register during initialization phase.(see NM93C06 memory map-datasheet). Configuration formula: bus rate = CKI/(10 * (CAN_PSC+1)) In *Table 13*, some examples of initialization are shown.

TABLE 13. Examples Bus Rate (CKI = 10 MHz)

CAN_PSC (dez)	Bus Rate (kbit/s)
01	500
03	250
04	200
07	125
09	100
19	50

3.7 Usage of Analog Input

The analog input is not intended to be a high performance A/D-conversion, but provides the capability of reading up to 16 different voltage levels with any of seven I/O pins. *Figure 18* shows an example by reading different voltage levels of a resistor array. This is done by measuring the charge or discharge time of an external capacitor. The internal construction of an I/O pin (see *Figure 18*) will support the analog input. At first, the level of the Schmitt Trigger Input is measured. Depending on the result, low or high, the internal driver, which is controlled by the charge/discharge logic, charges or discharges the external capacitor.

Schmitt Trigger level = low→charge capacity

Schmitt Trigger level = high→discharge capacity

The charge/discharge control is then disabled and the time to get the original digital (after Schmitt Trigger) state is measured by a counter register. This counter values consider different input voltages.

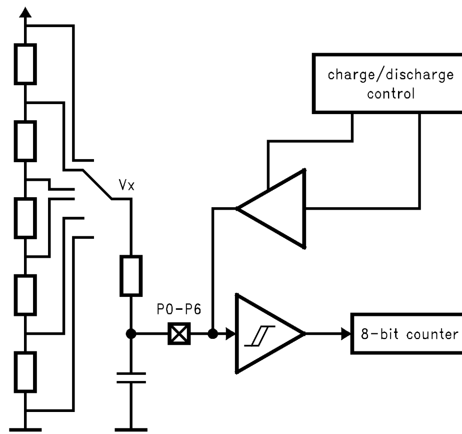


FIGURE 18. Analog Input

Example: Read 16 different voltages on pin P0 using R/C

The restriction of this A/D conversion is shown in *Figure 19*, because the charge or discharge time of the capacitor is dependent on the current and this current is not linear. Especially voltages near the Schmitt Trigger level the 8-bit counter value is overflowed and no measurement is possible. This measurement is dependent upon the CPU speed, hence the R/C values may have to be adjusted to accommodate a change in CKI value from 10 MHz. The external components, which are connected to the pin P0, are shown in *Figure 20*.

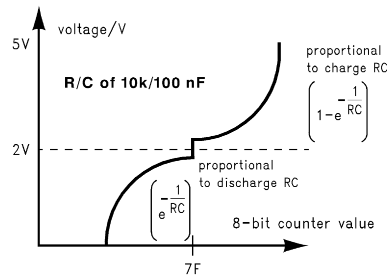


FIGURE 19. Input Voltage Depending on Counter Value

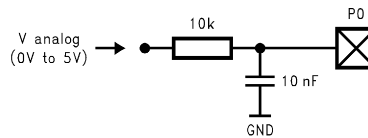


FIGURE 20. Example of Analog Input Components

Before the analog input Register marker can be executed, the pin has to be configured as High-Z input. This means that DD1 and OD1 have to be configured to low for the pin P0. The following CAN frame examples assume that the SLIO is configured to SINGLE WIRE RX0 bus mode, the error condition is error active and the receive ID = 0021. The data frames for the P0 configuration are shown in *Table 14* The pin configuration frames have to be transfered one time only.

TABLE 14. Read/Write Data Transfer Example

Direction	Name	ID	first databyte	second databyte
→	conf SINGLE WIRE RX0	0021 H	1F H	08 H
←	answer from SLIO	0020 H	3F H	08 H
→	OD1 to 11110000	0021 H	03 H	F0 H
←	answer from SLIO	0020 H	23 H	F0 H
→	DD1 to 11110000	0021 H	04 H	F0 H
←	answer from SLIO	0020 H	24 H	F0 H
→	analog input from P0	0021 H	08 H	XX H
←	answer from SLIO	0020 H	28 H	13 H (0.0V)

The values of the 16 different values are shown in *Table 15*.

TABLE 15. Reading of 16 Different Analog Voltages

Voltage Input (V)	counter value (hex)	counter range (+ 4 counter steps)	range number
0.0	13	0F to 17	0
1.3	23	1F to 27	1
1.5	2D	29 to 31	2
1.8	3D	39 to 41	3
1.9	46	42 to 4A	4
2.0	53	(±8) 4B to 5B	5
2.1	6C	(±8) 64 to 74	6
2.5	7F	(±8) 77 to 87	7
2.9	90	(±8) 88 to 98	8
3.0	9D	99 to A1	9
3.1	AB	A7 to AF	A
3.3	B7	B3 to BB	B
3.5	C1	BD to C5	C
3.8	CB	C7 to CF	D
4.2	D4	C1 to D8	E
5.0	DF	DB to E3	F

The different ranges of the example in *Table 15* are shown in *Figure 21*.

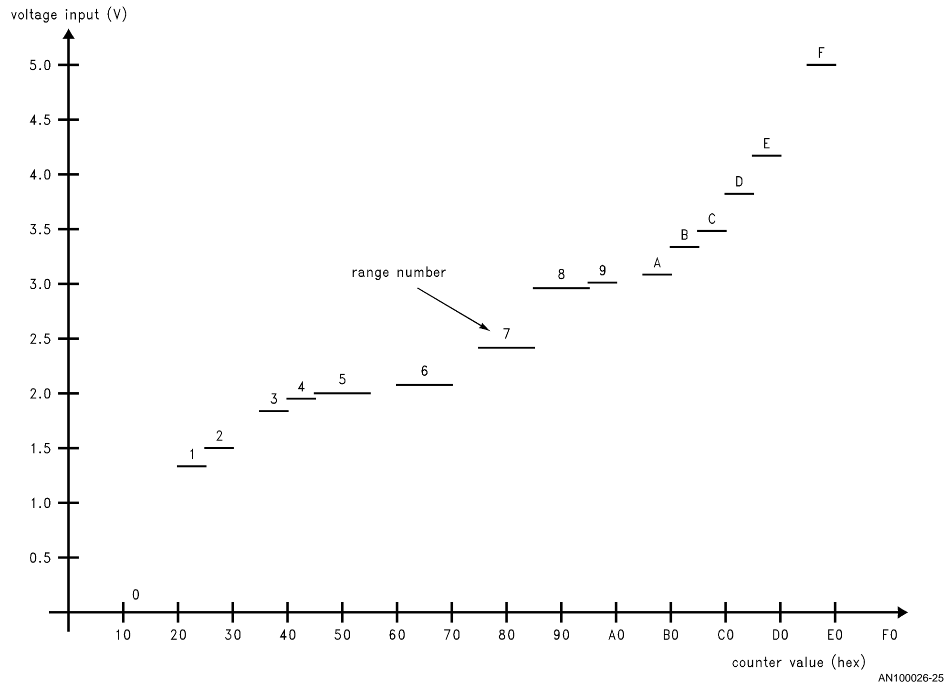


FIGURE 21. Graph of the Different Ranges

3.8 Usage of D/A Output

A user programmable PWM signal is provided on pin P9. This signal may be configured to either a 10-bit or 8-bit resolution. This PWM signal is CKI dependent. For example, by using CKI =10 MHz, one PWM cycle is 255 µs (8-Bit) or 1023 µs (10-Bit). In order to calculate the cycle time of the PWM using the following formula.

$$T_{pwm} = \frac{10 \times (2^{resolution} - 1)}{CKI}$$

By using an external low pass filter, analog voltages can be generated. An example of the RC is shown in *Figure 22*. The analog output will be configured with the SLIO registers DAC1, DAC2 and ACT.

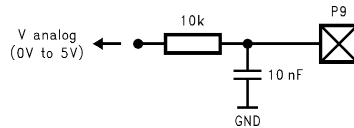


FIGURE 22. Example of Analog Output Components

To generate PWM signals on P9 the following steps have to be executed:

- configure P9 as output (over E2 or over CAN)
- configure High/Low Time of the PWM signal with the registers DAC2 and DAC1 (over E2 or over CAN)
- configure 8-Bit or 10-Bit PWM signal with the DAR Bit of the register ACT (over E2 or over CAN)
- enable PWM output with DACEN of ACT (over E2 or over CAN)

10-Bit PWM Configuration

The configuration of the SLIO registers DAC2/DAC1 via CAN is shown in *Table 16*.

TABLE 16. 10-Bit D/A Output Examples

Register Marker					second databyte (hex)	10-Bit Format D/A	
Bit4	Bit3	Bit2	Bit1	Bit0		DAC2	DAC1
1	0	0	0	0	B0	00	B0
1	0	0	0	1	B0	01	B0
1	0	0	1	1	B0	03	B0
1	0	0	1	1	B0	03	B0

Example: 10 Bit PWM over CAN

Table 17 summarizes all messages which are necessary to configure pin P9 (as a ten bit PWM output). It is assumed that the SLIO is configured in Single wire RX0 bus mode and the error mode is error active. The CKI is configured with 10 MHz. In *Figure 23* the PWM output resulting from the configuration of *Table 17* is shown.

TABLE 17. Data Transfer Example for 10-Bit D/A

Direction	Name	ID	first databyte	second databyte
→	conf SINGLE WIRE RX0	0401 H	1F H	08 H
←	answer from SLIO	0400 H	3F H	08 H
→	OD2 to 00000000	0401 H	06 H	00 H
←	answer from SLIO	0400 H	26 H	00 H
→	DD2 to 00000010	0401 H	07 H	02 H
←	answer from SLIO	0400 H	27 H	02 H
→	DAC2/DAC1 to 02 B0 H	0401 H	12 H	B0 H
←	answer from SLIO	0400 H	32 H	B0 H
→	ACT to 00000011	0401 H	1C H	03 H
←	answer from SLIO	0400 H	3C H	03 H

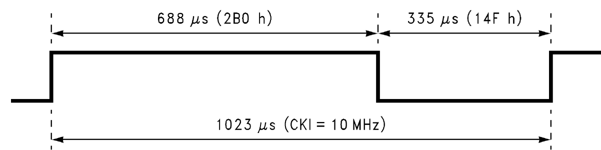


FIGURE 23. 10-Bit PWM Output

8-Bit PWM Configuration

The 8-Bit configuration of the SLIO registers DAC1 via CAN is shown in the *Table 18*. In this case Bit0/Bit1 of the register marker are don't care. That means that all register marker bits, which are reserved for DAC, can be used for 8-Bit PWM configuration.

TABLE 18. 8-Bit D/A Output Examples

Register Marker					second databyte (hex)	8-Bit Format D/A
Bit4	Bit3	Bit2	Bit1	Bit0		DAC1
1	0	0	x	x	B0	B0

Example: 8-Bit PWM configuration

In *Table 19* all CAN messages are summarized, which are necessary to configure pin P9 as 8-Bit PWM output. Hereby it is assumed that the SLIO is configured in Single wire RX0 bus mode and the error mode is error active. Moreover the CKI is configured with 10 MHz.

TABLE 19. Data Transfer Example for 8-Bit D/A

Direction	Name	ID	first databyte	second databyte
→	conf SINGLE WIRE RX0	0401 H	1F H	08 H
←	answer from SLIO	0400 H	3F H	08 H
→	OD2 to 00000000	0401 H	06 H	00 H
←	answer from SLIO	0400 H	26 H	00 H
→	DD2 to 00000010	0401 H	07 H	02 H
←	answer from SLIO	0400 H	27 H	02 H
→	DAC1 to 10110000	0401 H	12 H	B0 H
←	answer from SLIO	0400 H	32 H	B0 H
→	ACT to 00000010	0401 H	1C H	02 H
←	answer from SLIO	0400 H	3C H	02 H

3.9 Handling of External Events

Pins P0 to P7 provide monitoring of external events through detection of rising or/and falling edge transition. The configuration is done through the SLIO registers PE and NE. A one in a given bit of these registers enables the external event mode for the corresponding pin.

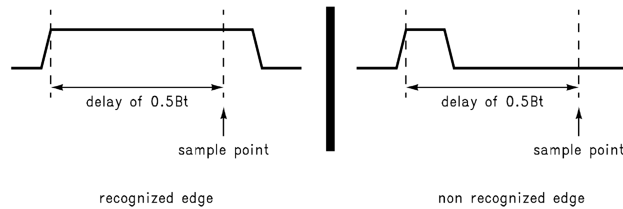
Example: configuration P0 - pos. edge and P1 - pos./ neg. edge

Table 20 depicts the configuration of the status of P0–P7 via the CAN bus. Subsequent to this configuration a matching edge on the port will result in a transmission of P0–P7 status on the bus from the SLIO. In order to eliminate the possibility of noise or switch bounce, the port is resampled after a time period of Bt. Note that this period is dependent on the CPU clock frequency. If an event occurs during a bus transaction the reporting of the event will be delayed until the bus is clear.

During the receive/transmit phase of the SLIO the process caused through event is delayed until CAN communication is finished.

TABLE 20. Configuration of PE and NE via CAN

Direction	Name	ID	first databyte	second databyte
→	conf SINGLE WIRE RX0	0401 H	1F H	08 H
←	answer from SLIO	0400 H	3F H	08 H
→	PE to 00000011	0401 H	01 H	03 H
←	answer from SLIO	0400 H	21 H	03 H
→	NE to 00000010	0401 H	02 H	02 H
←	answer from SLIO	0400 H	22 H	02 H



Note: 1Bt = 40960/CKI

FIGURE 24. Delay Time External Rising Event

3.10 Power Save Mode Examples

The SLIO device supports two different power save modes, SLEEP mode and NAP mode. SLEEP mode stops all activities and clock. NAP mode stops all activities but the clock and an internal counter. This counter will wake-up the device every Bt time *(Figure 25)*. The device will wake-up from both modes by an external signal applied on one or more of the port pins P0 to P6, by a recessive to dominate transition on the CAN bus and by pulling the $\overline{\text{RESET}}$ pin low. Waking-up triggers and automatic wakeup in NAP mode through the internal counter cause the transmission of a status message. If the device wakes up from SLEEP mode, it will stay in active mode *(Figure 25)* and all previous settings of the registers are valid again.

The power mode bits PO0 to PO2 in the control register CTR set up the power saving modes SLEEP and NAP. The different configurations are summarized in *Table 21*.

TABLE 21. Power Modes Configuration (CTR Register)

PO2	PO1	PO0	power mode
0	0	0	active
0	0	1	NAP: 1 * Bt
0	1	0	NAP: 2 * Bt
0	1	1	NAP: 4 * Bt
1	0	0	NAP: 8 * Bt
1	0	1	NAP: 16 * Bt
1	1	0	NAP: 32 * Bt
1	1	1	SLEEP

Example: SLEEP mode

The CAN messages, described in *Table 22*, enables the SLEEP mode.

TABLE 22. Configuration of SLEEP Mode

Direction	Name	ID	first databyte	second databyte
→	conf SINGLE WIRE RX0 and enable SLEEP mode	0401 H	1F H	E8 H
←	answer from SLIO	0400 H	3F H	E8 H

After this data transfer the device enters SLEEP mode, all activities including the CKI clock are stopped. The SLIO will wake up on a rising/falling edge on any enabled pin PO - P6 or upon a recessive/dominant transition on the CAN bus. *Table 23* gives an example of a wake-up transaction from SLEEP mode over CAN.

TABLE 23. Example of Wake-Up SLEEP Mode

Direction	Name	ID	first databyte	second databyte
→	wake-up message (SOF=rec./dom. transition)	0401 H	xx H	xx H
←	status message answer	0400 H	20 H	00 H

After wake-up the clock is running and the SLIO will stay in active mode.

Example: configuration NAP mode - 16*Bt

The CAN message in *Table 24* enables the 16*Bt NAP mode.

TABLE 24. Configuration 16*Bt NAP Mode

Direction	Name	ID	first databyte	second databyte
→	conf SINGLE WIRE RX0 and enable NAP mode	0401 H	1F H	A8 H
←	answer from SLIO	0400 H	3F H	A8 H

After this data transfer, the device is in NAP mode, all activities excluding the internal timer are stopped. This internal timer was configured through the second data byte of the previous message (*Table 24*) that after every 16*Bt, the device wakeup for 1*Bt (see also *Figure 25*). If during the NAP condition a wakeup is coming, the device will be active during the next 16*Bt, period. If during this period the power mode is not changed, the NAP mode is entering again.

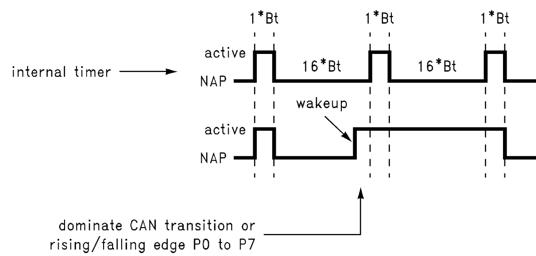


FIGURE 25. Timing NAP-mode (16*Bt)

4.0 CAN SYSTEM EXAMPLE

4.1 Start Up Consideration

In this section an example is shown to start a first CAN application. Before starting the following steps have to be checked:

- **At least two CAN nodes have to be connected on the CAN Bus, because every message on the bus needs an acknowledge**
- **Usage of the same physical bus interface as described in section 1**
- **Usage of the same bus mode (differential/single wire)**
- **Configuration of the termination on the two CAN bus endings depending on the physical bus interface**
  - — ISO High Speed (ext. Transceiver): 120Ω between CAN_H and CAN_L
  - — ISO Low Speed: voltage divider 1.75V/ 3.25V recessive levels
- **Usage of the same bus timing (described in section 1) for all CAN nodes**
- **Consideration of length/frequency and the number of CAN nodes**
- **Consideration of the number of SLIO nodes depending on the SLIO Identifier mode**
  - — Pin mode: connected SLIO < 16 (only 4+1 Identifier can be configured)
  - — EEPROM mode: quasi no limit (all ID in standard CAN format are used)

4.2 Network Description

These CAN communication examples between COP884BC and the SLIO describe the basis of an application with National CAN interface. The COP884BC software controls the CAN data transfer, which means that the counter value of a decrement 8-Bit counter is transmitted to the SLIO pins P0 to P7. In order to control the CAN data, the status of the counter is also given out to L_port of the COP884BC. The communication is restricted to SLIO CAN format. The schematics of COP884BC node and SLIO node are shown in *Figure 26* and *Figure 27*.

To start the application, the following steps have to be executed:

- **Reset COP884BC**
- **create a rising edge to the port pin G0 for COP884BC**
- **Reset the SLIO**

After the Controller receives the Status Message of the SLIO, the counter will be enabled and the data transfer begins. Next, all CAN frames from COP884BC will be requested from the SLIO by an answering message. The software of COPCAN waits for this message and will generate the next data frame after a delay caused through the IDLE Timer pending flag T0PND.

The physical features are summarized in the next points:

- **CKI = 10 MHz**
- **Bus Rate = 250 kbit/s**
- **external transceiver chip connection (ISO High Speed)**

- **usage of the external EEPROM NMC93C06**

The EEPROM configures the receive/transmit ID′s to/from the SLIO (0023/0022), the bus mode and the data direction of P0 to P7. The configuration of the EEPROM registers are shown in *Table 25*.

TABLE 25. Example of Configuration of SLIO Registers in EEPROM Mode

E2-address	EEPROM Registers	SLIO Registers
0x00	**E2-MASK** 0xAA	
	PIN-E2-MASK 0x00	
0x01	**RXIDH** 0x00	
	RXIDL 0x22	
0x02	**RXIDGH** 0x00	
	RXIDGL 0x00	
0x03	**PEDGE** 0x00	**PE** 0x00
	NEDGE 0x00	**NE** 0x00
0x04	**ODATA1** 0x00	**OD1** 0x00
	ODATA2 0x00	**OD2** 0x00
0x05	**DATADIR1** 0xFF	**DD1** 0xFF
	DATADIR2 0x00	**DD2** 0x00
0x06	**DACH** 0x00	**DAC2** 0x00
	DACL 0x00	**DAC1** 0x00
0x07	**ACR** 0x00	**ACT** 0x00
	CCR 0x00	**CCT** 0x00
0x08	**DCR** 0x08	**CTR** 0x08
	CAN_PSC 0x03	**CAN Prescaler** 0x03

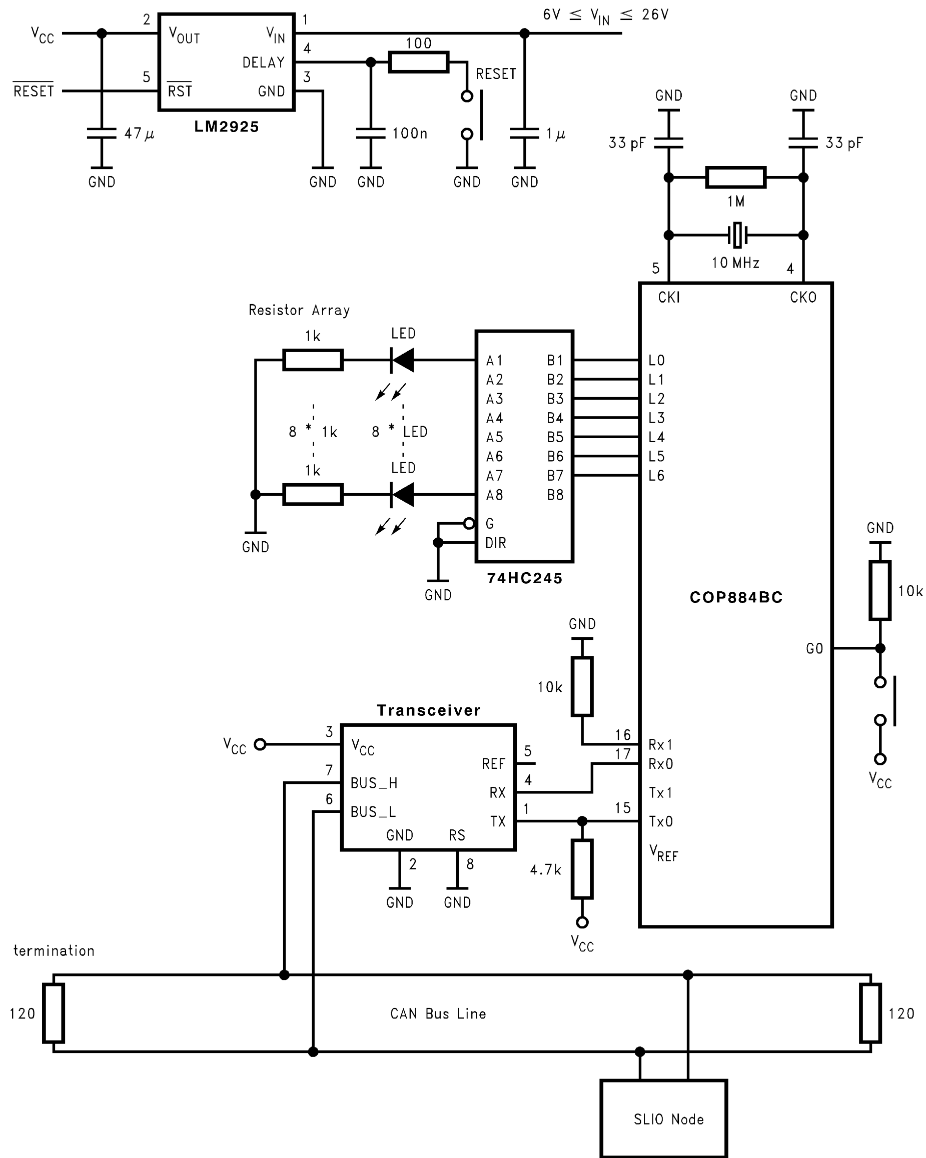


FIGURE 26. COP884BC Node

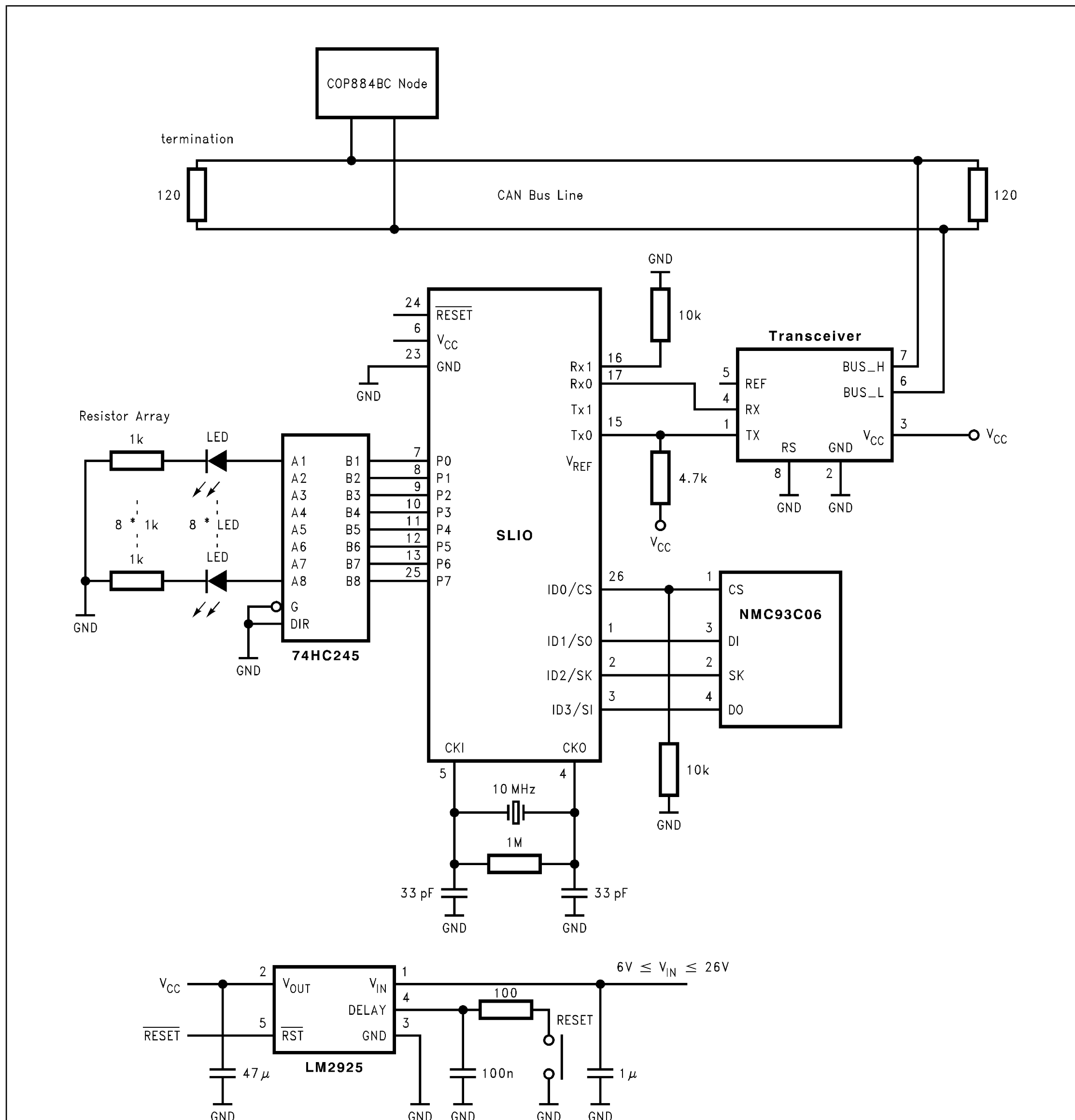


FIGURE 27. SLIO Node Schematic

4.3 Software Structure

The Block Diagram in *Figure 28* describes the software process. The software can be separated into the following modules.

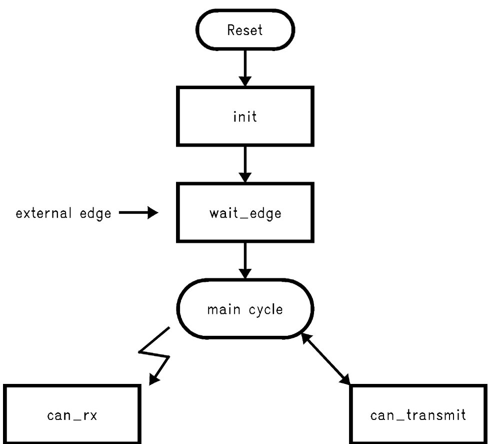


FIGURE 28. Software Block Diagram

- **initialization (init)**

After Reset, the software will execute the initialization routine. Within this routine the various interrupts and the initialization of the CAN interface will be configured.

- **rising edge wait (wait_edg)**

Next, the software waits for the rising edge on pin G0. If this rising edge is received, the interrupt routine enables the access of the application.

- **main cycle (main)**
- **CAN receive interrupt routine (can_rx)**

This is the same receive interrupt routine as described for 2 bytes or less in section 2. It processes the answering message from the SLIO and saves the data in the receive object rx_obj. Then the control bit is set and the COP884BC can transmit the next data message with the next counter value after a self defined delay.

- **CAN Transmit routine (can_tx)**

This is the same transmit interrupt routine as described for 2 bytes or less in section 2.

4.4 Source Code

```
.incld cop888bc.inc
tx_cnt = 0    ; flag equations for the control register
tx_dly = 1    ; flag equations for the control register
action = 2    ; flag equations for the control register
lo =   00a    ; delay (lo * 40960/CKI)
.sect msg_buf, base
    tx_obj:  .dsb 4
    ; transmit object format:
    ; tx_obj[0] = trtr, tid[10:4]
    ; tx_obj[1] = tid[3:0], tdlc[3:0]
    ; tx_obj[2] = txd1 !
    ; tx_obj[3] = txd2 !
    rx_obj0: .dsb 4
    ; rx_obj[0] = lock, rid[10:4]
    ; tx_obj[1] = rid[3:0], rdlc[3:0]
    ; tx_obj[2] = rxd1 !
    ; tx_obj[3] = rxd2 !
.endsect
;==========================================================================
.sect base,base
        control:    .dsb 1    ; allocation of flag control register
.endsect
;==========================================================================
.sect register,reg
        counter:    .dsb 1
        light:      .dsb 1
.endsect
;==========================================================================
.sect code,rom,abs=0
main:
```

```
reset:
        ld sp,#02f        ;load stack pointer
;----------------------------------------------------------------------
; clear ram from 0x00 to 0x2f
; stack area will overwrite as well→don't use as a subroutine
;----------------------------------------------------------------------
clr_ram:
        ld      b,#02f           ; pointer to the last ram location
clr_loop:
        ld      [b],#0           ; clear ram byte
        drsz    b                ; decrement and "skip if zero"
        jp      clr_loop         ; ..counter>0
        ld      [b],#0           ; clear first ram byte
;----------------------------------------------------------
init:
        ld      counter,#000     ; reset counter
        ld      light,#000       ; reset light counter
init_prt_l:
        ld      portlc,#0ff
init_G0:
        rbit    iedg,cntrl       ; →rising edge
        sbit    exen,psw         ; enable extrn int
        rbit    expnd,psw        ; clear extern int pending
init_can:
        jsr     can_init
conf_rx_obj0:
        ld      b,#rx_obj0       ; configure receive message box
        ld      [b+], #082       ; with ID 0022 , 2 byte messages
        ld     [b], #022
enable_can:
        ld      cbus,#058        ; conf single wire rx0
                                 ; RIAF enabled→compare with higher id's
                                 ; enable CAN
enable_int:
        ld      b,#tcntl
        sbit    rie,[b]          ; enable can receive int
        sbit    gie,psw          ; enable global interrupt
;-------------------------------------------------------------------
;main cycle
;-------------------------------------------------------------------
start:
wait_begin:
        ifbit   action,control   ; wait until rising edge is comming
        jp      start_loop       ; yes.. process
        jp      wait_begin
start_loop:
        ifbit   tx_cnt,control   ; transmission
        jsr     cantx
        jp      start_loop
;-------------------------------------------------------------------
cantx:
        jsr     delay            ; delay routine
        jsr     action_count     ; count lights
        jsr     can_tx           ; transmit
        sbit    7,rx_obj0        ; enable receive buffer 0
        ret
;-------------------------------------------------------------------
delay:
        ld      counter,#lo      ; conf t0pnd_counter
        ld      b,#icntrl        ; point icntrl
dlay:
        rbit    t0pnd,[b]        ; reset t0 pending flag
loop_w:
        ifbit   t0pnd,[b]        ; wait unti t0pnd is set
        jp      count            ;
        jp      loop_w           ;
count:
```

```
        drsz    counter          ; count x*(40960/CKI)
        jp      dlay
        rbit    t0pnd,[b]        ;
        ret
;-------------------------------------------------------------------
action_count:
        rbit    tx_cnt,control   ; reset
        drsz    light
        nop
        ld      a,light
        x       a,portld
conf_tx_obj:
        ld      b, #tx_obj
        ld      [b+], #002       ; tid,#002
        ld      [b+], #032       ; tdlc,#032
        ld      [b+], #003       ; rxd1, #003
        ld      a,light
        x       a, [b]           ; rxd2, #count value light
        ret
;==========================================================================
.sect code_can_init, rom
; this code initializes the CAN with minimun
; possible instuctions/rom space
can_init:
        ld      b, #cscal
        ld      [b+], #3         ; CAN prescaler
        ld      [b+], #00f       ; ctim (BTL)
        ld      [b], #0          ; TCNTL ; don't point to RTSTAT
                                 ; clear RERR, TERR, etc..
        ret
.endsect
;==========================================================================
.sect code_can_tx, rom
; this code transmits a 0 to 2 byte or remote CAN message
; from a transmit buffer tx_obj[0:3]
; this code intentionally does not check for remote or
; DLC (data length code) as the COPCAN interface will
; automatically transmit no data bytes in a remote frame
; and not more than DLC data bytes
can_tx:
        rc                       ; (*) reset error flag
        ifbit   TXPND, RTSTAT    ; check if transmit busy
        jp      tx_busy          ; .. yes then exit
        ld      b, #tx_obj       ; point to tx_obj[0]
        ld      x, #TID          ; point to TID
        ld      a, [b+]          ; get tx_obj[0]; point to tx_obj[1]
        x       a, [x-]          ; .. and save
        ld      a, [b+]          ; get tx_obj[1]; point to tx_obj[2]
        x       a, [x-]          ; .. and save
        ld      a, [b+]          ; point to tx_obj[3]
        ld      a, [b-]          ; get tx_obj[3]; point to tx_obj[2]
        x       a, [x-]          ; save to TXD1
        ld      a, [b]           ; get tx_obj[3]
        x       a, [x]           ; save to TXD2
tx_done:
        sbit    txss, tcntl      ;set pending transmission
                                 ;automatic reset of txss after transmission
        ret                      ; exit without error
tx_busy:
        sc                       ; (*) indicate tx_busy
        ret                      ; (*) exit with error
        ; retsk                  ; optional use retsk instead
                                 ; 1st and last 2 lines to skip next
.endsect
;==========================================================================
.sect    int,rom,abs=0ff
interrupt:
```

```
        push    a
        ld      a,b
        push       a
restore:
        vis
int_end:
        pop     a
        x       a,b
        pop     a
        reti
.endsect
;===========================================================================
.sect   inttab, rom , abs=01E0
        .addrw restore           ; default VIS
        .addrw restore           ; PortL interrupt/wake-up
        .addrw restore           ; reserved
        .addrw restore           ; reserved
        .addrw restore           ; reserved
        .addrw restore           ; PWM Timer
        .addrw restore           ; MicroWire/Plus
.        addrw restore           ; T1B
        .addrw restore           ; T1A
        .addrw restore           ; Idle Timer
        .addrw int_g0            ; Pin G0
        .addrw restore           ; CAN Transmit
        .addrw restore           ; CAN Error
        .addrw can_rx            ; CAN Receive
        .addrw restore           ; reserved
        .addrw reset             ; Opcode 00 Software-Trap
.endsect
;===========================================================================
.sect   code_can_rx, rom         ; from interrupt
can_rx:
        ; this interrupt is triggerd by RBF, RRTR or RFV
        ; RRTR and RBF are cleared by reading or b's pointing to RXD1
        ; RFV is cleared by reading RTSTAT to A
        ; or executing the equiv. of LD B, #RSTAT; LD A, #xx
;------
        sbit    tx_cnt,control
;------
        ld      b, #rx_obj0      ; (*) receive id hi ; * only with RIAF = 0
        ifbit   7, [b]           ; buffer free
        jp      receive_msg      ; .. yes then receive
        ld      b, #rx_obj1      ; next buffer
        ifbit   7, [b]           ; buffer free
        jp      receive_msg      ; .. then receive msg
        jp      can_rx_exit      ; else exit
receive_msg:
        rbit    7, [b]
        ld      a, rid           ; (*) get received id
        ifne    a, [b]           ; (*) check if accept
        jp      can_rx_exit      ; (*) .. no then exit
        x       a, [b+]
        ld      a, ridl          ; get received IDLC
        x       a, [b]           ; save message
        ifbit   RRTR, RTSTAT     ; received frame remote frame?
        jp      can_rx_rtr       ; yes
        jp      save_data        ; no
can_rx_rtr:
        ld      a,[b]            ; remote frame is signed
        or      a,#0F            ; through rdlc = F
        x       a,[b]            ;
        jp      wait_rx          ;
save_data:
        ld      a,[b+]           ;dummy read→point rx_data register
        ld      a, RXD1          ;
        x       a, [b+]
```

```
        ld      a, RXD2
        x       a, [b]
        ld      b, #RTSTAT
wait_rx:
        ifbit   RFV, [b]
        jp      rx_done
        ifbit   RERR, TCNTL
        jp      rx_error
        jp      wait_rx
; this is the error routine error interrupt must not be enabled
rx_error:
        ld      b, #rx_obj1
        ifbit   7, [b]
        jp      check_obj0
        jp      end_error
check_obj0:
        ld      b, #rx_obj0
end_error:
        sbit    7, [b]          ; free buffer
        rbit    RERR, TCNTL
rx_done:
can_rx_exit:
        ld      a, RXD1         ; dummy read to clear RBF, RTR
        ld      a, RTSTAT       ; dummy read to clear RFV
        jp      int_end
.endsect
;======================================================================
.sect   rom,rom
int_g0:
        sbit    action,control
        rbit    expnd,psw       ;clear extern int pending
        jp      int_end
.endsect
.end main
```


LIFE SUPPORT POLICY

NATIONAL'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS WRITTEN APPROVAL OF THE PRESIDENT OF NATIONAL SEMICONDUCTOR CORPORATION. As used herein:

1. Life support devices or systems are devices or systems which, (a) are intended for surgical implant into the body, or (b) support or sustain life, and whose failure to perform when properly used in accordance with instructions for use provided in the labeling, can be reasonably expected to result in a significant injury to the user.
2. A critical component in any component of a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system, or to affect its safety or effectiveness.

National Semiconductor Corporation
Americas
Tel: 1-800-272-9959
Fax: 1-800-737-7018
Email: support@nsc.com

www.national.com

National Semiconductor Europe
Fax: +49 (0) 1 80-530 85 86
Email: europe.support@nsc.com
Deutsch Tel: +49 (0) 1 80-530 85 85
English Tel: +49 (0) 1 80-532 78 32
Français Tel: +49 (0) 1 80-532 93 58
Italiano Tel: +49 (0) 1 80-534 16 80

National Semiconductor Hong Kong Ltd.
13th Floor, Straight Block,
Ocean Centre, 5 Canton Rd.
Tsimshatsui, Kowloon
Hong Kong
Tel: (852) 2737-1600
Fax: (852) 2736-9960

National Semiconductor Japan Ltd.
Tel: 81-3-5620-6175
Fax: 81-3-5620-6179

National does not assume any responsibility for use of any circuitry described, no circuit patent licenses are implied and National reserves the right at any time without notice to change said circuitry and specifications.