

Figure 14.1. A layered approach to communication systems.

- 1) Address information field
 - physical address specifying the destination/source computers
 - logical address specifying the destination/source processes (e.g., users)
- 2) Synchronization or handshake field
 - Physical synchronization like shared clock, start and stop bits
 - OS synchronization like request connection or acknowledge
 - Process synchronization like semaphores
- 3) Data field
 - ASCII text (raw or compressed)
 - Binary (raw or compressed)
- 4) Error detection and correction field
 - Vertical and horizontal parity
 - Checksum
 - Block correction codes (BCC)

Lab 3g Distributed Data Acquisition

This lab must be performed in teams of 4 to 6 students. Approval of the team by the TA is required before the preparation is started. Existing lab groups can be merged together to form the team (no splitting of existing groups). Only one version of the software need be developed for each team, but the number of operational nodes in the network is equal to the number of students on the team minus one. For example a group of 6, should create a network with 5 nodes.

Goals

- Develop a layered communication system,
- Design and implement a hardware/software interconnect protocol.
- Use communication skills to work effectively as team.

Specifications

- Each node measures sound locally in real time using its microphone.
- Each node plays sound on its speaker that was measured at a different node.
- There are at least **n-1** nodes, where $4 \leq n \leq 6$ is the size of the group.
- The CAN interface is used.
- Interrupt synchronization is used in an appropriate manner.
- The hardware/software system is modular.
- The system uses FIFO queues.
- You run the system at various sampling rates in order to determine the maximum bandwidth.

Controller Area Network (CAN).

- high-integrity serial communications
- real-time applications
- 1 Mbits/second (you will run at 250,000 bits/sec)
- originally for use in automobiles,
- can have up to 112 nodes.

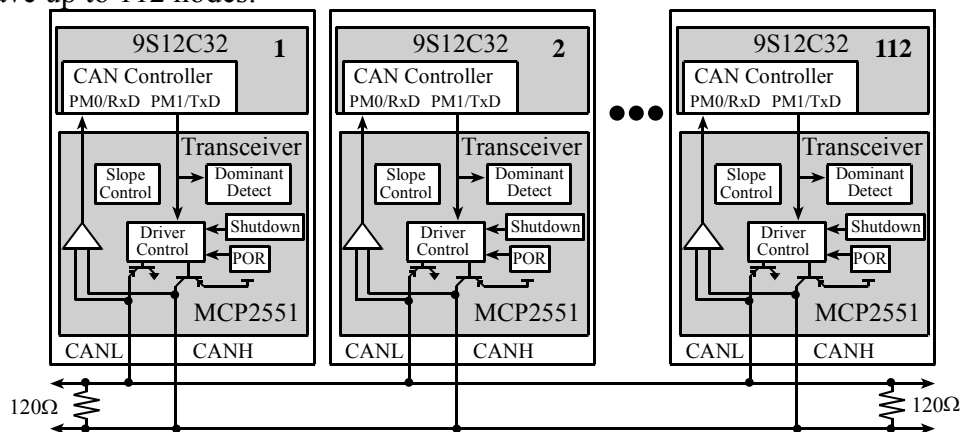


Figure 14.5. Block Diagram of a 9S12C32-Based CAN communication system

Similar to wire-or open collector logic

Dominant state is logic low

Recessive state is logic high

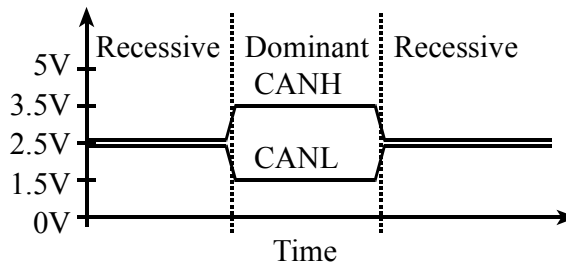


Figure 14.6. Voltage specifications for the recessive and dominant states.

Four message types or frames

- **Data Frame,**
- **Remote Frame,**
- **Error Frame,** and
- **Overload Frame.**

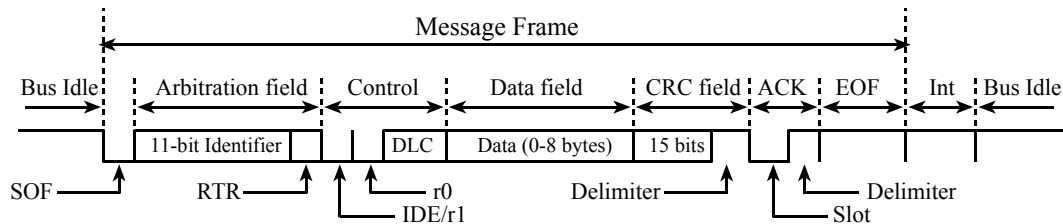


Figure 14.7. CAN Standard Format Data Frame.

Arbitration Field

11-bit identifier specifies data type (in our case sound destination)
 priority handled by dominate wins over recessive
 lower IDs are higher priority
 RTR=IDE=0 means 11-bit standard format data frame

Control Field

DLC, which specifies the number of data bytes (0 to 8)

Data Field

contains zero to eight bytes of data.

CRC Field

15-bit checksum used for error detection.

$$\text{Bandwidth} = \frac{\text{number of information bits/frame}}{\text{total number of bits/frame}} \cdot \text{baud rate}$$

Number of bits in a CAN message frame.

- ID (11 or 29 bits)
- Data (0, 8, 16, 24, 32, 40, 48, 56, or 64 bits)
- Remaining components (36 bits)
 - SOF (1)
 - RTR (1)
 - IDE/r1 (1)
 - r0 (1)
 - DLC (4)
 - CRC (15)
 - ACK/EOF/intermission (13)

How many bits in a frame:

- Standard CAN 2.0A frame with 4 data bytes?
- Extended CAN 2.0B frame with 8 data bytes?

Bit Stuffing

Where is the clock? (Answer: in the data)
 Data line needs edges so the receiver can synchronize
 A long sequence of 0's or a long sequence of 1's,
 Insert a complementary bit after five bits of equal value.
 CAN 2.0A may add 3+n stuff bits (n is number of bytes)
 CAN 2.0B may add 5+n stuff bits.
 Receiver has to un-stuff

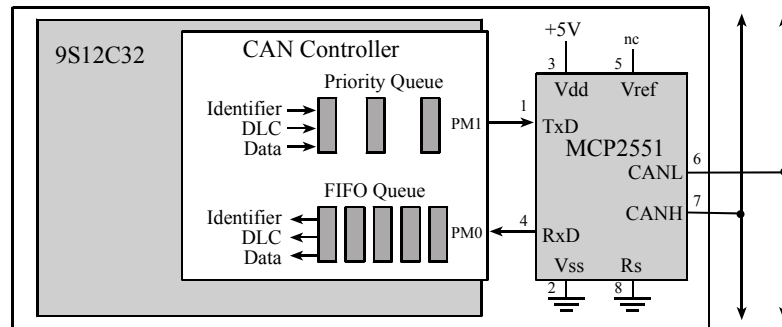


Figure 14.8. Data flow through the 9S12C32 CAN controller (corrected).

Leave pin 5, V_{REF}, not connected (typo in book).
 A resistor from pin 8, R_s, to ground can be added
 to restrict the slew rate on the channel.

Limiting the slew rate will reduce EMI emissions.

Address	Bit 7	6	5	4	3	2	1	Bit 0	Name
\$0140	RXFRM	RXACT	CSWAI	SYNCH	TIME	WUPE	SLPRQ	INITRQ	CANCTL0
\$0141	CANE	CLKSRC	LOOPB	LISTEN	0	WUPM	SLPAK	INITAK	CANCTL1
\$0142	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	CANBTR0
\$0143	SAMP	TSEG22	TSEG21	TSEG20	TSEG13	TSEG12	TSEG11	TSEG10	CANBTR1
\$0144	WUPIF	CSCIF	RSTAT1	RSTAT0	TSTAT1	TSTAT0	OVRIF	RXF	CANRFLG
\$0145	WUPIE	CSCIE	RSTATE1	RSTATE0	TSTATE1	TSTATE0	OVRIE	RXFIE	CANRIER
\$0146	0	0	0	0	0	TXE2	TXE1	TXE0	CANTFLG
\$0147	0	0	0	0	0	TXEIE2	TXEIE1	TXEIE0	CANTIER
\$014A	0	0	0	0	0	TX2	TX1	TX0	CANTBSEL
\$014B	0	0	IDAM1	IDAM0	0	IDHIT2	IDHIT1	IDHIT0	CANIDAC
\$0150-\$0153	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0	CANIDAR0 – CANIDAR3
\$0154-\$0157	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0	CANIDMR0 – CANIDMR3
\$0158-\$015B	AC7	AC6	AC5	AC4	AC3	AC2	AC1	AC0	CANIDAR4 – CANIDAR7
\$015C-\$015F	AM7	AM6	AM5	AM4	AM3	AM2	AM1	AM0	CANIDMR4 – CANIDMR7
\$0160	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	CANRXIDR0
\$0161	ID2	ID1	ID0	RTR	IDE=0	0	0	0	CANRXIDR1
\$0164-\$016B	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	CANRXDSR0- CANRXDSR7
\$016C	0	0	0	0	DLC3	DLC2	DLC1	DLC0	CANRXDLR
\$0170	ID10	ID9	ID8	ID7	ID6	ID5	ID4	ID3	CANTXIDR0
\$0171	ID2	ID1	ID0	RTR	IDE=0	0	0	0	CANTXIDR1
\$0174-\$017B	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	CANTXDSR0- CANTXDSR7
\$017C	0	0	0	0	DLC3	DLC2	DLC1	DLC0	CANTXDLR
\$017F	PRI07	PRI06	PRI05	PRI04	PRI03	PRI02	PRI01	PRI00	CANTXTBPR

Table 14.1. MC9S12C32 CAN ports.

CLKSRC is the CAN Clock Source bit, which defines the clock source for the CAN module. We set it to 1 to use the Bus Clock, and to 0 to use the Oscillator Clock.

```
void CAN_Open(void){
    asm sei                // make atomic
    CANFifo_Init();        // Initialize FIFO data structure (you write this)
// 1) put CAN into initialization mode
    CANCTL1 |= 0x80;       // CANEN=1, Enable CAN
    CANCTL0 |= 0x02;       // SLPRQ=1, go to sleep first
    while((CANCTL1&0x02)==0){}; // SLPK signifies Sleep Mode

    CANCTL0 &= ~0x02;     // SLPRQ=0, leave Sleep Mode
    CANCTL0 |= 0x01;       // INITRQ=1, Enter Initialization Mode
    while((CANCTL1&0x01)==0){}; // INITAK signifies Initialization Mode

    CANCTL1 &= ~0x10;     // LISTEN=0, get out of Listen-only mode
    CANCTL1 &= ~0x40;     // CLKSRC=0, use oscillator clock (8 MHz)

// 2) set the receive filters
    CANIDAC = 0x10;       // four 16-bit filters
    CANIDMR0 = 0xFF; CANIDMR1 = 0xFF; CANIDMR2 = 0xFF; CANIDMR3 = 0xFF;
    CANIDMR4 = 0xFF; CANIDMR5 = 0xFF; CANIDMR6 = 0xFF; CANIDMR7 = 0xFF;

// 3) specify baud rate clock
    CANBTR0 = 0x03;       // (x+1)=4, assume oscillator is 8 MHz
    CANBTR1 = 0x23;       // (3+y+z)=8, divide by 32 gives 250,000 bits/sec

// 4) active mode, armed for receive message interrupts
    CANCTL0 &= ~0x01;     // INITRQ=0, Leave Initialization mode
    while(CANCTL1&0x01){}; // wait for the end of initialization
    CANRIER |= 0x01;      // Arm RxF, interrupt on receive message
    asm cli                // Enable interrupts
}
}
```

Program 14.1. Initialization of the 9S12C32 CAN network.

There are two priorities

 Within the three blocks locally on the transmitting CAN (**CANTXTBPR**)

 On the network competing with other transmitters (11-bit **id**)

```
void CAN_Send(unsigned short id, char length, char *data, char priority) {
char *pt=(char*)&_CANTXDSR0;    // points to transmit message buffer;
// 1) wait for TxCAN to be ready to accept a frame
// replace this gadfly loop with a periodic polling ISR
    while((CANTFLG&0x07)== 0){}; // Wait for transmit buffer available

// 2) Configure the frame in the TxCAN
    CANTBSEL = CANTFLG;         // Request selection of empty transmit buf
    CANTXIDR0 = id>>3;          // Write Identifier into ID registers
    CANTXIDR1 = id<<5;          // with RTR and IDE=0
    CANTXDLR = length;          // 0 to 8 bytes
    //
    while(length){
        *pt++ = *data++;        // copy data into data registers
    }
}
```

```

    length--;
}
CANTXTBPR = priority;          // set priority of this message
// 3) Send the frame
CANTFLG = CANTBSEL;           // flag buffer as ready for transmission
}

```

Program 14.2. Transmit a message on the 9S12C32 CAN network.

There are three options for moving the CAN transmission into the background

Periodic interrupt with period polling

CAN transmitter interrupts

Arm when data to be sent

Disarm when no data to be sent

Call this existing **CAN_Send()** from the background

Very efficient because one less ISR, one less FIFO

Risky, might crash... why?

```

void CAN_Receive(char msg[13]) {
    while(CANfifo_Get(msg) == 0){}; // wait for incoming message
// removes 13 bytes from CANfifo
// msg[0-1] contain ID (left justified)
// msg[4-11] contain data
// msg[12] contains DLC (number of bytes)
}
interrupt 38 void CANInterruptHandler(void){
    char *msgPtr = (char*)&_CANRXIDR0;
    if(CANRFLG & RXF){
        CANCTL0 |= RXFRM; // clear Received frame flag (clears all flags)
        CANfifo_Put(msgPtr); // copy 13 bytes into CANfifo
        CANRFLG |= RXF; // clear RXF by writing a 1 (clears all flags)
    }
}

```

Program 14.3. Receive a message from the 9S12C32 CAN network.

CANfifo puts and gets an entire 13-byte message.

It does not put 1 byte at a time, called 13 times.

Rather, it is called once and either

successfully puts all 13 bytes or

fails (fifo full) and puts no bytes.

You have to write this FIFO

Why 13 bytes?

Need to make sure the ID+data+DLC are one non-divisible message

Could create a 11-byte buffer

Move ID from `CANRXIDR0` (\$0160-\$0161) into buffer

Move data from `CANRXDSR0` (\$0164-\$016B) into buffer

Move DLC from `CANRXDLR` (\$0165) into buffer

Then Put the 11-byte buffer into the Fifo

A trick to improve speed

Simply put 13 bytes directly from CAN into Fifo

First 2 bytes are ID
Next 2 bytes have no meaning (\$0162-\$0163)
Next 8 bytes are data
Last byte is DLC

Lab 3g options (your choice, assuming 4 nodes)

Round robin 1->2, 2->3, 3->4, 4->1

Speaker phone 1->(2,3,4), 2->(1,3,4), 3->(1,2,4), 4->(1,2,3)

Telephone Use buttons to dial and answer
 If node n calls node m, n->m, and m->n

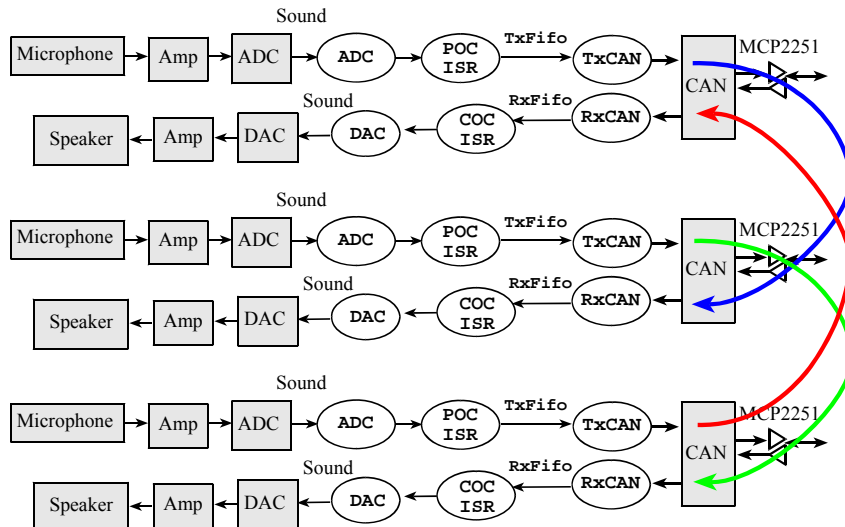


Figure 3.1. Data flow graph of the distributed data acquisition system.

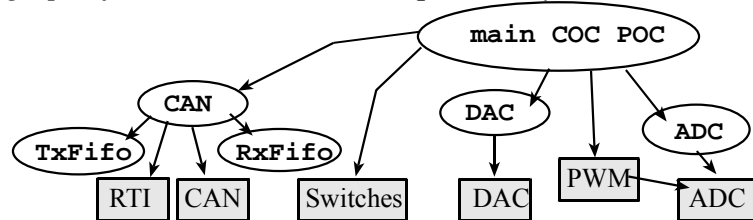
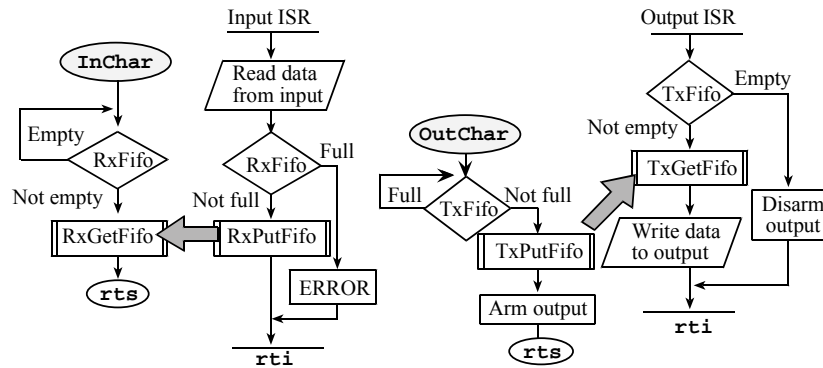


Figure 3.2. Call-graph of the distributed data acquisition system.

Be careful to consider what your software does if

- TxFifo** is full on a **TxFifo_Put**
- TxFifo** is empty on a **TxFifo_Get**
- RxFifo** is full on a **RxFifo_Put**
- RxFifo** is empty on a **RxFifo_Get**



FIFO queues used in SCIA to pass data between threads.

RxFifo becomes full

- data will be lost
- always if average input rate > average processing rate
 - must improve processing rate or slow down input rate
- temporary increase in arrival rate
- temporary decrease in process rate
 - could solve by increasing the size of the **RxFifo**.

TxFifo becomes full

- appropriate but inefficient to gadfly wait
- efficient design has no waiting on a full **TxFifo**

I/O bound (limited by I/O speed)

- RxFifo** has either 0 or 1 entry
- TxFifo** has many entries
- bandwidth improved by increasing speed of I/O devices

CPU bound (limited by software execution speed)

- TxFifo** has either 0 or 1 entry
- RxFifo** has many entries
- bandwidth improved by speeding up software execution

Synchronization issues

- How to connect transmitter/receiver threads?
 - How to start, handshake
 - Race conditions
- How to prevent streaming sound from stalling?
 - ADC sampling, FIFO full
 - DAC outputs, FIFO empty