EG3576

COMMUNICATIONS ENGINEERING I
- COMMUNICATIONS FOR CONTROL

GORRY FAIRHURST
RAFFAELLO SECCHI
SCHOOL OF ENGINEERING
UNIVERSITY OF ABERDEEN

HTTP://WWW.ERG.ABDN.AC.UK/~GORRY/EG3576/

CAN-RDM V09

1

DUPLEX SERIAL COMMUNICATIONS

2

REMOTE DEVICE MANAGEMENT (RDM)

RDM Standardised as E1.20 (2010)

- RDM physical layer
- Packet format for RDM and the UID
- Communicating with devices
- Discovering the UIDs of devices
- RDM repeaters

3

WHY RDM?

Before RDM, any change to a device meant actually setting switches/controls on the device itself.

Using RDM, devices can be monitored and configuration can be changed remotely using the bus.

4

WHAT IS RDM?

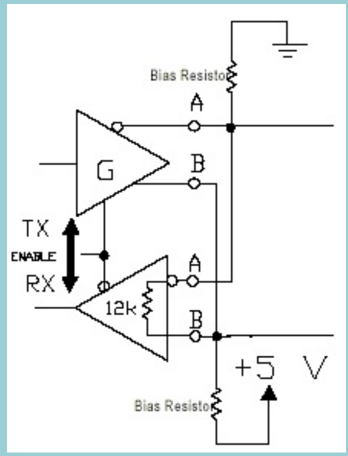
Remote Device Management

Allows bi-directional communication to/from a device using the DMX cable.

This can be used to:

- Build a list of all devices on a DMX bus
- Set a device's DMX base address (which slots to read)
- Set a device's DMX channel profile (what slots do)
- Monitor the status or faults reported by a device
- Download an upgrade to the device firmware

THE RDM PHYSICAL LAYER



RDM HISTORY

Work started 2001, main spec 2010, updated 2023

Should the standard use ***two*** wires or ***four*** wires?

Soon after 2001 it was decided to use just two wires

Two-wire DMX cable was then common

It uses a half-duplex bus (one transmitter active at any time)

Each RDM device also has a ***Unique ID*** (not DMX address)

http://tsp.plasa.org/tsp/documents/docs/E1-20RDM_2006.pdf

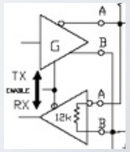
RDM BUS TERMINATION

A classical DMX sender is conected at one end of the DMX cable

In RDM, any of the 32 devices on the bus might send

The signal therefore travels in both directions along the cable

It is important to terminate BOTH ends of the cable with 120 Ohms

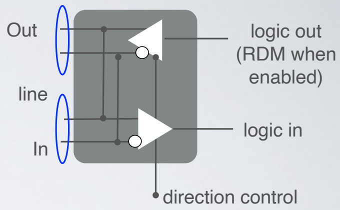


RDM PHYSICAL LAYER

RDM uses a Bi-directional EIA-485 bus

An RDM device uses ***tri-state*** drivers

- This uses ***Half Duplex***



Each device controls the direction of transmission:

(a) The master normally sends; Others normally listen.

(b) These roles can be reversed to allow equipment to send.

(c) There can be moments where there is no sender.

(d) There may be transients when more than one device tries to send (in half-duplex these result in signal corruption).

9

HALF DUPLEX OPERATION

There are two roles assumed to enable an equipment to send:

(1) One device is the ***master*** - usually the DMX sender.

The master controls who can transmit to the bus.

The master initiates a communications request to a "***slave***" by addressing the unit and then setting the transceiver to receive.

(2) The master listens for a response (receive mode).

The slave receiver recognises a control slot.

If the slot addresses the slave, it enables its own ***transmitter***.

(3) Once data sent, the slave ***reverts back to receive mode***.

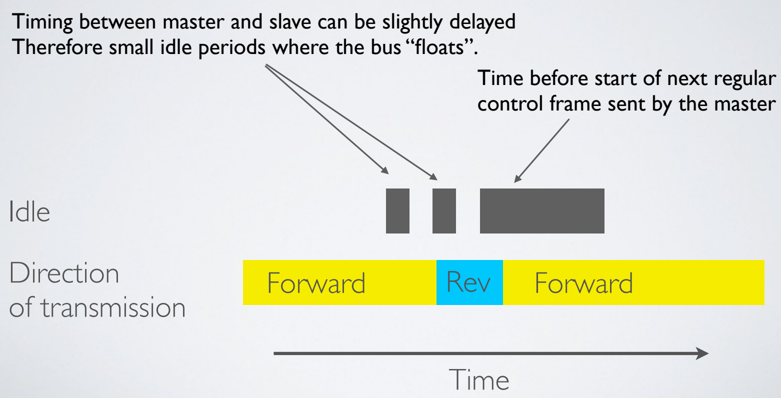
Master resumes control after reception from slave (or a timeout).

10

RDM - HALF DUPLEX

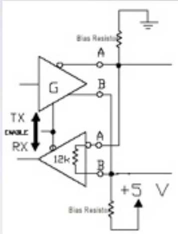
11

RDM - IDLE TIMES



12

THE CONTROLLER



When the line is idle, it “floats”

This makes a receiver vulnerable to noise

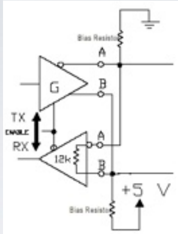
Instead, a bias network is added to ensure the line level > 245 mV

Line A is connected via bias resistance to GND

Line B isconnected via bias resistance to +5V

Of course, only do this once for each bus!

CALCULATING BIAS



Each EIA-485 node has an input impedance of 12K.

32 nodes in parallel present load of 376 ohms.

Two 120 Ohm terminators - a combined 60 Ohm load.

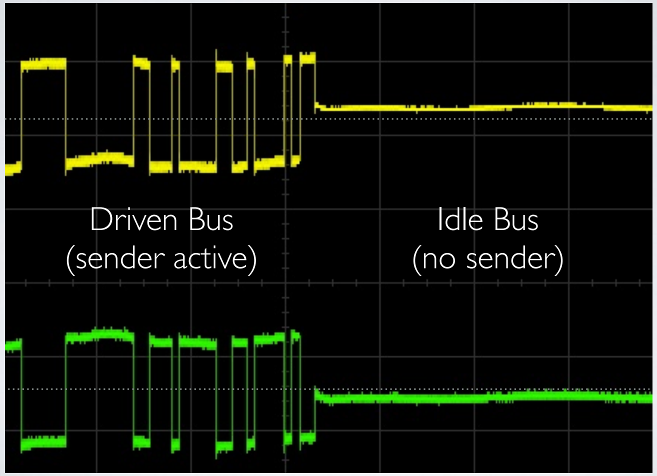
Total load is therefore ***51.8 ohms.***

To maintain ***at least 245 mV*** between B & A line, needs a bias current of ~ 4.7 mA to flow through this load.

A 5V supply needs a series resistance of 1063 Ohms, subtract 51.8 Ohms of bus loading, this leaves 1011 Ohms.

Placing half as a pull-up to 5V and half as a pull-down to ground gives a bias of 505 Ohms, ***510 Ohms*** to nearest preferred value.

RDM BIAS



THE PACKET FORMAT FOR RDM AND THE UID

An RDM packet is sent in a DMX frame with:

Start Code (value 0xCC, 204 decimal)

RDM Header (24 slots);

Message Length; Source; Dest.; Command; Param.; etc

RDM Data Area (variable)

Checksum (2 slots) - 16-bit sum of all slot values

IDENTIFYING RDM DEVICES

All RDM frames use a Start code of 0xCC

- “simple” devices already ignore non-zero start codes!

Each RDM device has a Unique ID (UID)

- The UID is assigned by a manufacturer
- This is not a DMX base address (position in the frame)
- The UID is a globally unique identifier

17

RDM PARAMETERS

Each device has:

- A UID (permanently set by the manufacturer)
- A flag to say whether the device is ***addressed***
- A flag to say whether the device is ***muted*** *(see later)*

A set of parameters stored in an EEPROM data (non-volatile):

- The device DMX base address
- The current profile (mapping slots to parameters)
- Other configuration parameters (defined by the profile)
- Other status parameters (e.g., temperature, current, time used)

18

RDM UNIQUE ID

All RDM equipment is uniquely identified:

- Manufacturers assign a ***unique 6 byte UID***
- FFFF: FFFF FFFF (Broadcast)
- A 2B Manufacturer ID is assigned to each manufacturer
- UID = 2B Manufacturer ID + 4B Serial Number (Flat address)
- 2B Manufacturer ID: FFFF FFFF (All manufacturer systems)

DMX base address can be changed depending on the use

- The ID is ***not*** the DMX base address
- An RDM Device is ***addressed*** irrespective of DMX address

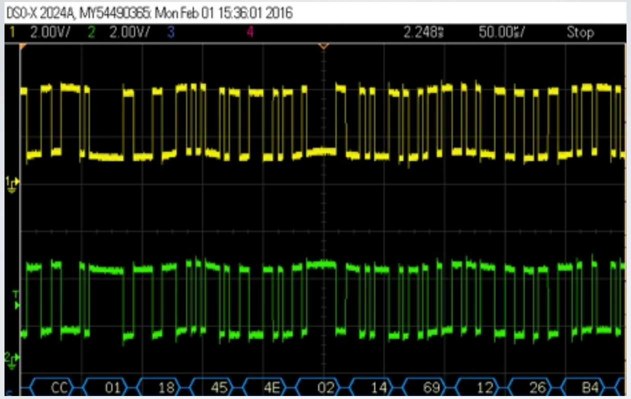
19

RDM CHECKSUM

- **Sender**:
  - Calculates the unsigned, modulo 0x10000, 16-bit ***additive checksum*** of the entire packet slot data (from START Code to end of frame)
  - Places result in the ***Checksum field*** of the frame
- **Receiver**:
  - Calculates the unsigned, modulo 0x10000, 16-bit additive checksum of the entire packet slot data (from START Code to end of frame)
  - Compares result with the Checksum field of the frame
  - Only if two match frame is ***OK***, otherwise frame is ***discarded***

20

RDM PACKET FORMAT



A

B

Decode

Start Code

RDM Header (24 slots), RDM Data, and Checksum

COMMUNICATING WITH RDM DEVICES

RDM COMMANDS

RDM devices do not respond to commands unless addressed

- They do read DMX data sent with a start code of 0x00

To communicate with a specific device using RDM:

1. Address the device using the UID ("Listen" sent to the UID)
2. Write (set) or read (get) information stored in the set of parameters
3. Then the device is released ("Quiet")

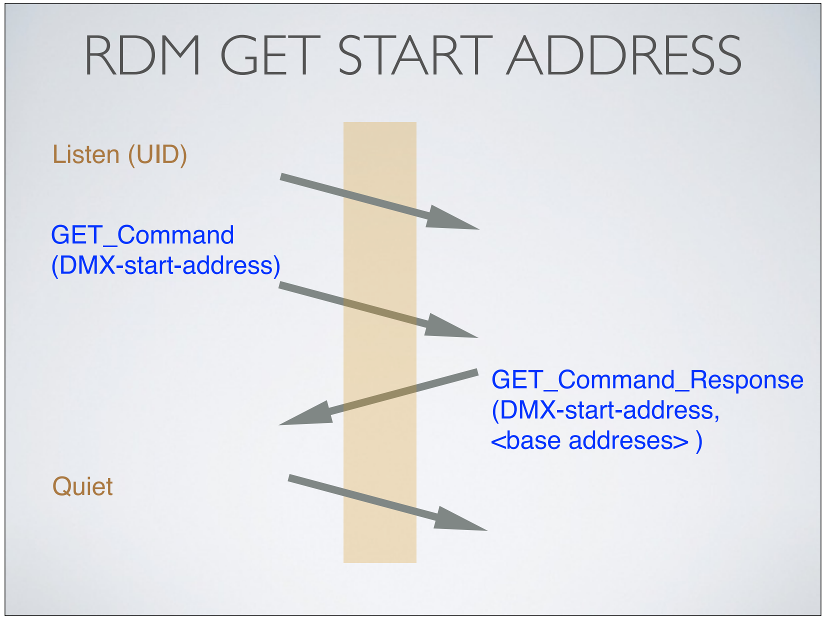
MASTER MUST KNOW UIDS

The master needs to know the UID of ***each*** receiver

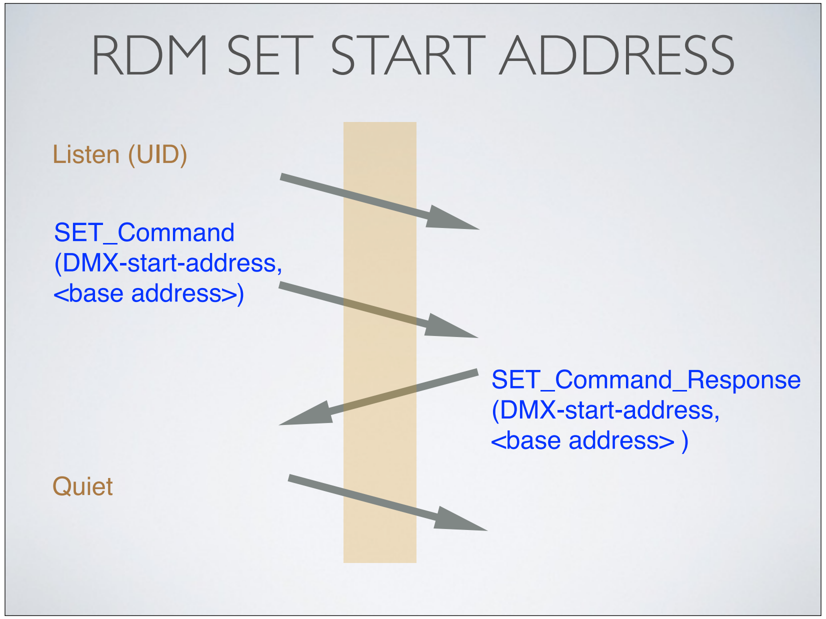
- Important to address each device
- Important to know what equipment is on the bus.
- i.e. parameters need to be interpreted in context.

Key question is how to find out what is connected!

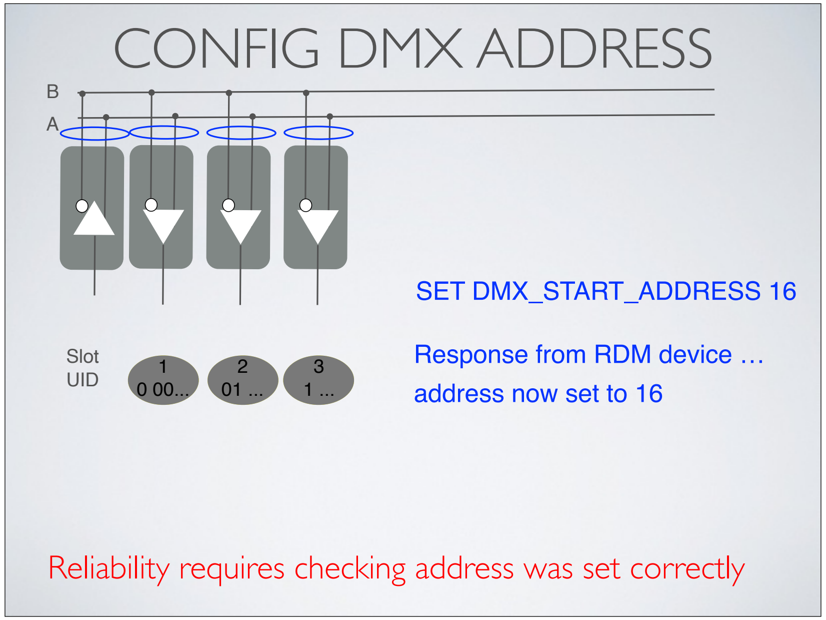
281,474,976,710,656 UID values!


RDM GET START ADDRESS
Listen (UID)
GET_Command
(DMX-start-address)
GET_Command_Response
(DMX-start-address,
<base addreses>)
Quiet

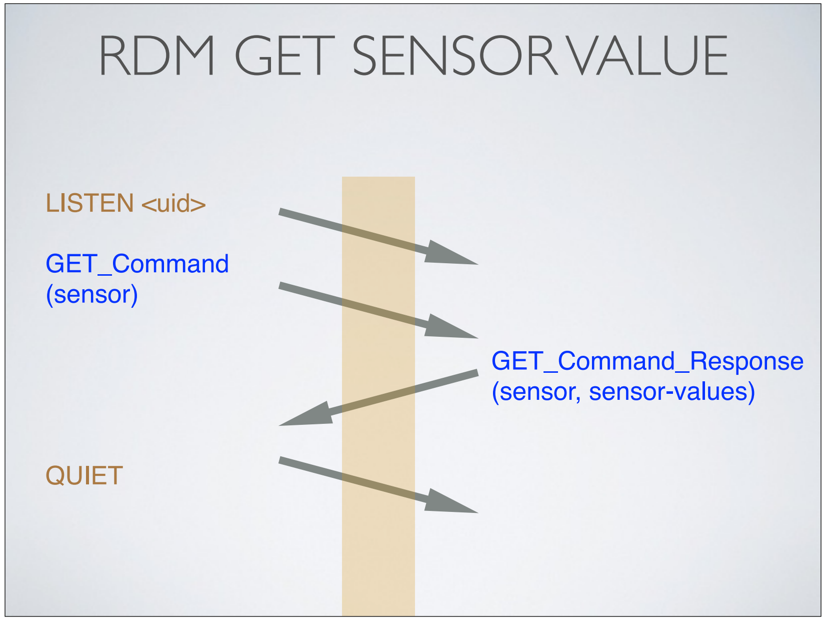
25


RDM SET START ADDRESS
Listen (UID)
SET_Command
(DMX-start-address,
<base address>)
SET_Command_Response
(DMX-start-address,
<base address>)
Quiet

26

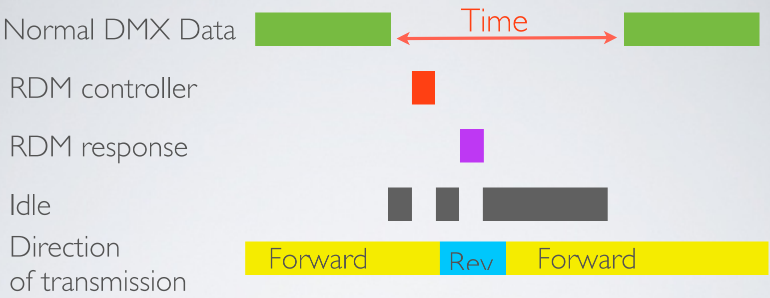

CONFIG DMX ADDRESS
B
A
Slot
UID
1
0 00...
2
01 ...
3
1 ...
SET DMX_START_ADDRESS 16
Response from RDM device …
address now set to 16
Reliability requires checking address was set correctly

27


RDM GET SENSOR VALUE
LISTEN <uid>
GET_Command
(sensor)
GET_Command_Response
(sensor, sensor-values)
QUIET

28

TIMING OF RESPONSES

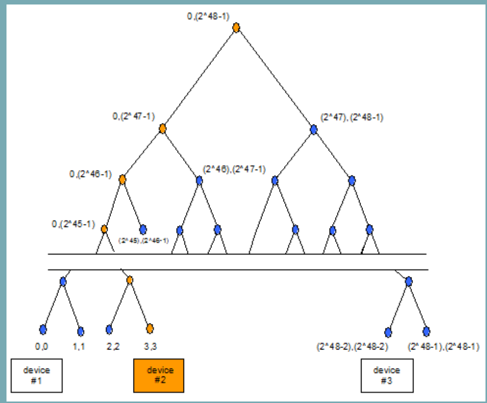


A "simple DMX" device ignores start code > zero

RDM takes time, this limits the **maximum** frame rate of sender.

- Commands typically require only one receiver to respond
- RDM not recommended during time critical communications

RDM DISCOVERY



RDM MASTER

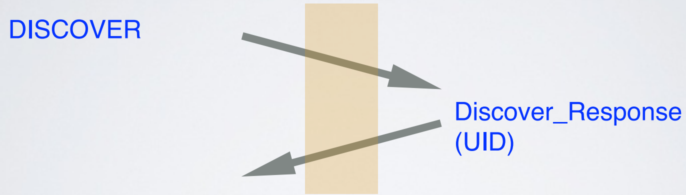
The RDM master (controller)

Needs to find a list of the devices that respond to RDM

***Discovery** is used to ask devices to respond*

Devices respond to discovery messages by sending their UID

DISCOVERY - ONE DEVICE



Once the UID is discovered the controller can address the device.

When more than one device responds, the Discover_response will be corrupted by multiple devices sending at the same time!

RDM DEVICE MUTE FLAG

Each RDM device has a ***MUTE*** Flag

The RDM bus controller can set or clear this MUTE Flag

DISC_UNMUTE (UID)

DISC_UNIQUE_BRANCH (UID-range)

Once set, the device does ***not*** respond to Discovery messages

This is used in the discovery algorithm in tow ways:

To resolve collisions (avoiding two replies at the same time)

To avoid discovered devices responding, once found.

RDM - UID DISCOVERY

Master discovers UID of each device on network.

Starts with DISC_UNMUTE FFFF: FFFF FFFF

- Tells all muted devices to respond

- Master clears its list of responders

RDM ***discovers*** devices ***polling***

DISC_UNIQUE_BRANCH [0000: 0000 0000 - FFFF: FFFF FFFF]

- Tells all devices to respond: Range to respond

No response? ... then there are no responders.

One response ... we've found the only responder (add to list).

Collision ... there is more than one responder!

RDM - UID DISCOVERY

RDM then starts a ***binary search***

- divides the search space into two halves:

DISC_UNIQUE_BRANCH [0000: 0000 0000 - 7FFF: FFFF FFFF]

- Do these devices have the first bit unset?

No response? ... there are no responders in bottom half.

One response ... we've found a responder, add to list.

Tell responder to mute, and expand the search range.

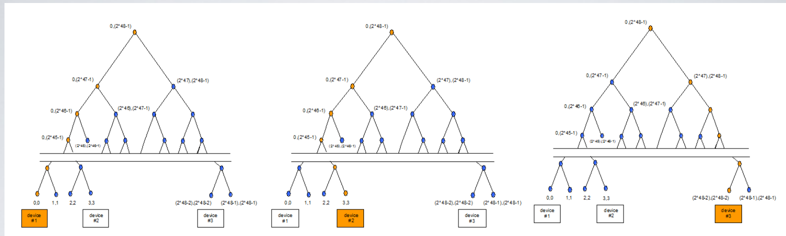
Collision ... there is more than one responder:

divide the range by two and loop...

Repeat for other ***half*** of space:

DISC_UNIQUE_BRANCH [8000: 0000 0000 - FFFF: FFFF FFFF]

RDM DISCOVERY

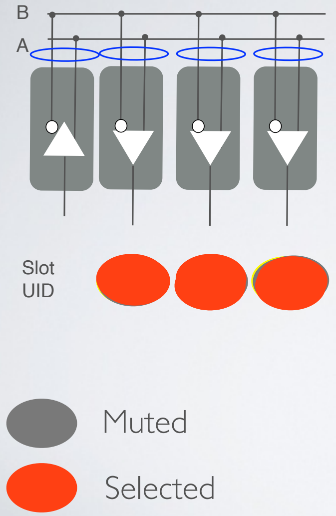


Isolate parts of the tree using a ***Binary Search***

Discovery finishes when there are no more devices to MUTE

At this stage, the master has a list of all device UIDs

DISCOVERY OF DEVICE UID



UMUTE ALL: ???? ?
DISC_UNIQUE_BRANCH ???? ?
Multiple response
DISC_UNIQUE_BRANCH 0??? ?
- 1 response, slot 1 = 00001
DISC_MUTE 0000 1
DISC_UNIQUE_BRANCH 1??? ?
Multiple response
DISC_UNIQUE_BRANCH 10?? ?
1 response, slot 2 = 10011
DISC_MUTE 10011
DISC_UNIQUE_BRANCH 1??? ?
- 1 response slot 3 = 11000
DISC_MUTE 11000
???? ? - No response
All devices have been found!

DISCOVERING CHANGES

After discovery the controller ought to know the UID of every device

It can then retrieve the DMX base address, equipment profile, and ar other required parameters

What happens when a new RDM device is added to the bus?

....Or a discovered device its removed?

The RDM Master controller could use the ***discovery algorithm***

... This can require many commands and take a long time

Instead, a RDM Master controller could be smarter

Incremental discovery uses the already discovered list of devices

CHECKING DISCOVERED LIST OF DEVICES

First step: Check the list of responders in the list.

Send a command to each UID

If the device ***responds***, then it is still there.

If it it ***does not respond***, remove the UID from the list

DISCOVERING NEW DEVICES

The second part of incremental discovery is checks for new devices

Send DISC_UNMUTE FFFF: FFFF FFFF

Send DISC_MUTE each previously discovered slot in list

See if any new responders have appeared

i.e. DISC_UNIQUE_BRANCH [0000: 0000 0000 - FFFF: FFFF FFFF]

- After this, the RDM Master controller knows all devices on the bus

LOSS OF COMMANDS

What happens when a responder ***misses*** a command?

Missing a MUTE or UNMUTE breaks the protocol!

- it is helpful to repeat all critical commands

- also helps to add delay between repeated commands.

41

DISCOVERY PROBLEM

The initial design had a problem:

The lights "***flickered***" in the first design.

... because more than one device could respond

... the collision signal ***could look like a start code of zero***!

... other devices would read this as data

The solution came in two parts:

1) Do not send a Break/MAB for RDM responses, instead respond using a special pre-amble sequence

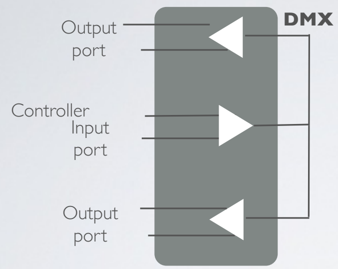
2) Encode data so it is highly unlikely that a "combined" signal is wrongly interpreted as actual data.

42

RDM SPLITTERS

43

DMX REPEATER (RECAP)



A DMX repeater is designed for a ***simplex link***

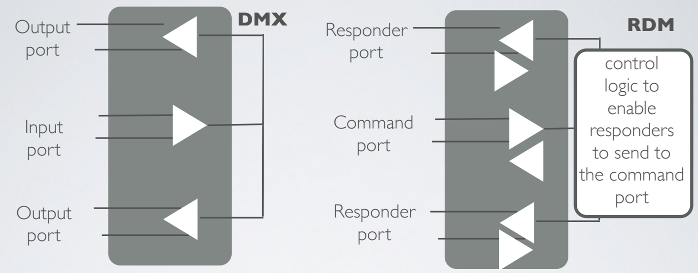
All DMX frames originates at the control

The repeater/splitter copies the DMX frames to all the output ports

A DMX repeater will never repeat RDM responses from output ports back to the controller

44

RDM SPLITTERS/REPEATERS

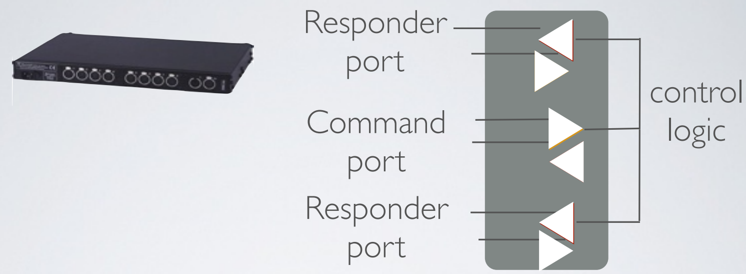


An RDM repeater/splitter needs to be different to support ***half-duplex***.

The repeater/splitter configures the transceivers at the ports so a responder can send a frame to the command port, when it needs to.

This frame only needs to be sent to the ***command port*** (i.e. master). (A slave never needs to send frames to other slaves).

RDM REPEATERS DETAIL

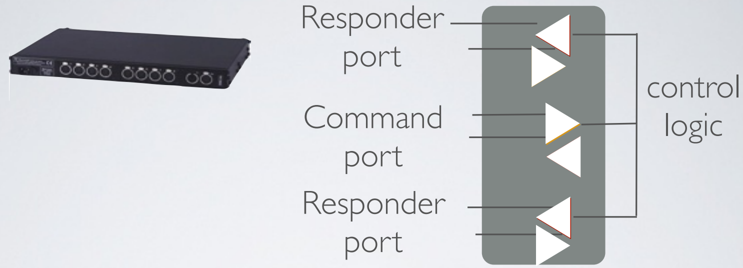


Two types of port

Responder Ports receive commands, and transmit responses towards controller

Command Ports sends commands and can receives responses

RDM REPEATERS DETAIL



All ports ***can*** be enabled to send or receive

Normally, the command port is in receive mode, other ports in send

When a ***break*** is received on a responder port.

A frame is received by the repeater on a responder port

The frame is repeated towards the master using the ***command port***

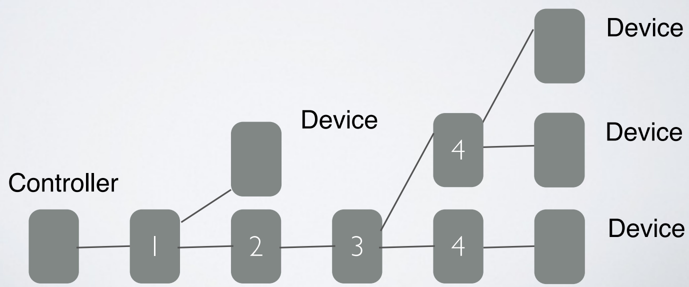
The repeater returns the command port back to receive mode

RDM REPEATER NETWORKS

RDM Repeaters need to support half-duplex

Overall network timing important for half duplex

No more than 4 repeaters in series (timing constraint)



RDM QUESTIONS

- **Does RDM slow-down the speed of DMX update?**
- **Why did the first version of RDM make DMX equipment "flicker"?**
- **Does RDM replace DMX?**
-

49

FURTHER DMX READING

- "Control Freak - A real world guide to DMX-512 and Remote Device Management", Wayne Howell, 2010
- "Recommended Practice for DMX 512: A Guide For users and Installers", Adam Bennette, (PLASA) *
- ANSI E1.11, Asynchronous Serial Digital Data Transmission Standard for Controlling Lighting Equipment and Accessories, USITT DMX512-A, American National Standards Institute, 1990 (PLASA) *
- ANSI E1.20, Remote Device Management, over USITT DMX 512 Networks, 2003 (PLASA) *

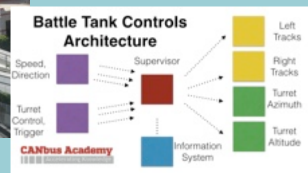
* Free download at tsp.plasa.org

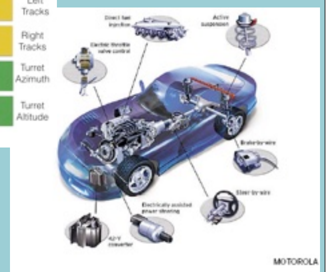
50

SYNCHRONOUS CONTROL

51

CAN

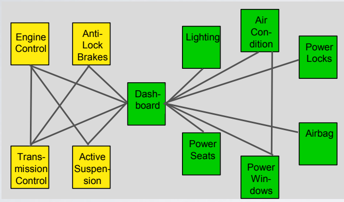




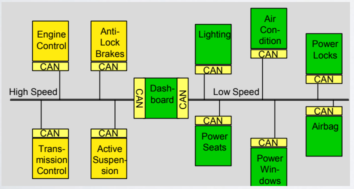
Controller Area Network
G Fairhurst

52

POINT-TO-POINT WIRING



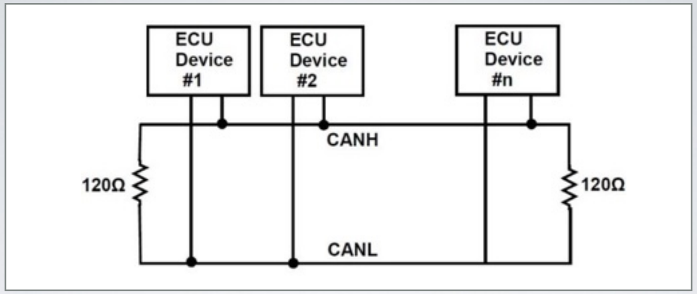
Traditional car wiring loom can be several miles of cable!!



A bus significantly reduces cable & cost

53

CAN BUS



120 Ohm shielded twisted pair cable

Specified as 108 - 132 Ohms

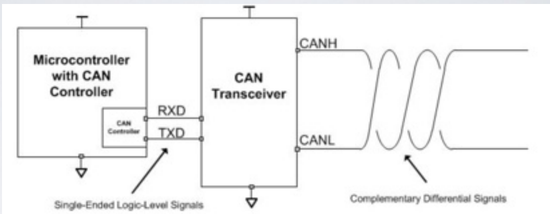
The conductors in the pair are labelled CANH and CANL

A shield reduces EMI

Bus terminated at each end with 120 Ohm resistor

54

CAN TRANSMISSION

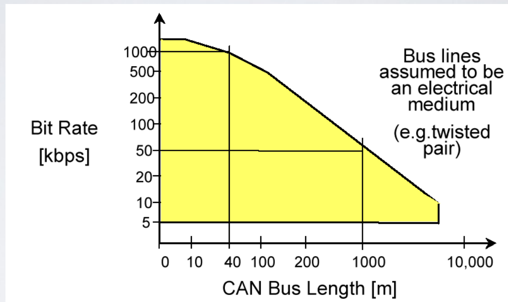


Max 1 Mbps data transmission

(CAN-FD is compatible and works at 5 Mbps)

55

CAN BUS LENGTH



MAximum bus length is a function of bus speed

1 Mbps <= 40m

125 kbps <= 500m

56

CAN TRANSCEIVER

CAN transceivers use Open-Collector (O/C) logic to connect to the bus

Logic 1 (recessive): No signal sent

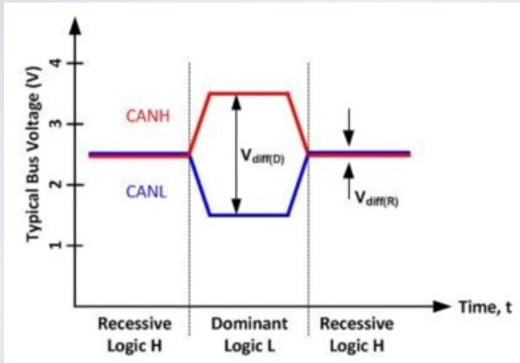
- Output at CAN_L floats to 2.5V
- Output at CAN_H floats to 2.5V
- i.e. there is a no voltage difference between the conductors

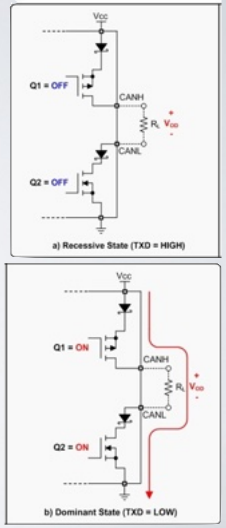
Logic 0 (dominant): Forces bus to a zero level

- Output at CAN_L driven to 1.5V
- Output at CAN_H driven to 3.5V
- i.e. there is a 2V voltage difference between the conductors

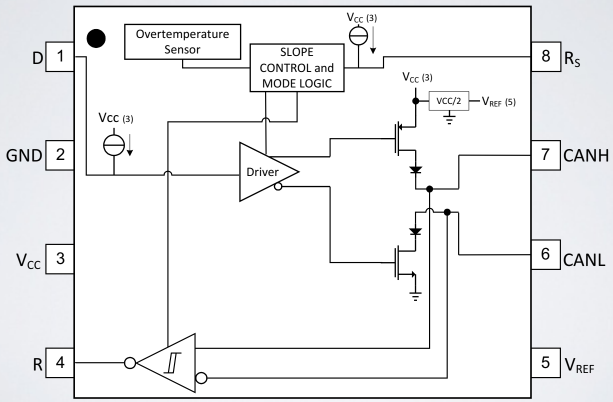
A receiver detects a 0 when CAN_H-CAN_L > 0.9V

CAN CABLE VOLTAGE

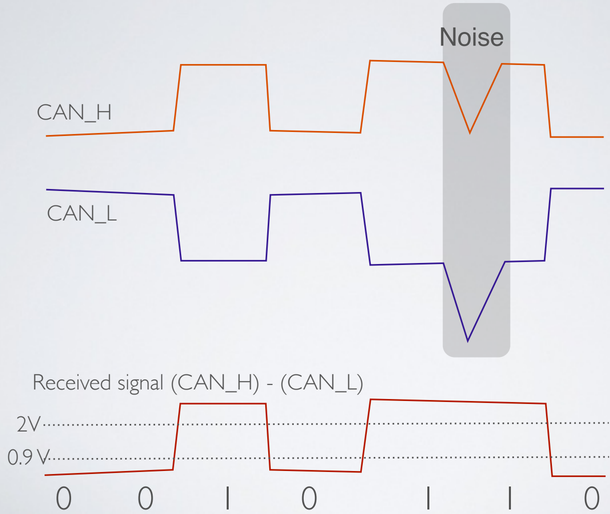




TI SNX5HVD251 INDUSTRIAL CAN BUS TRANSCEIVER



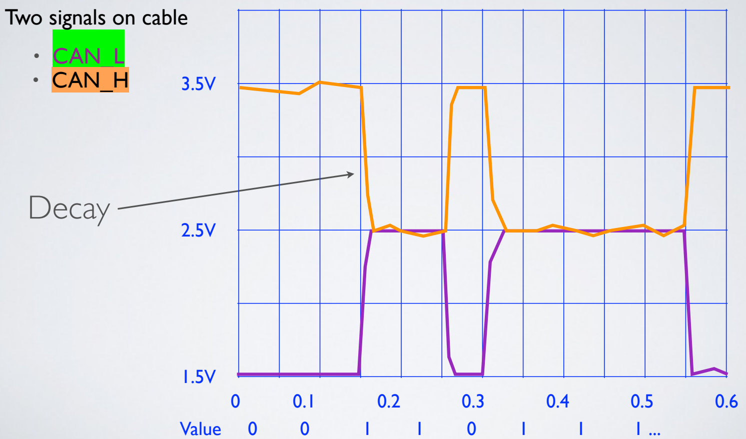
DIFFERENTIAL RECEPTION



CAN SIGNAL

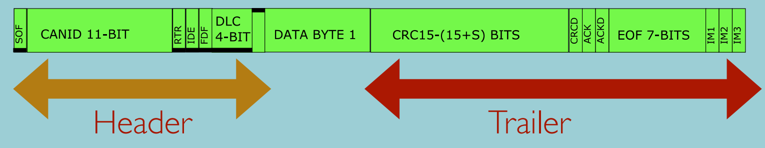
Two signals on cable

- CAN_L
- CAN_H



61

CAN FRAME



There is no bus master

All frames have a format defined by the header

Each frame ***may*** carry some data

Each frame ends with a common trailer

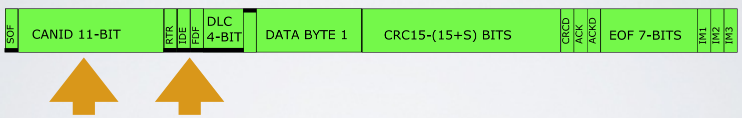
62

CAN ID

Every frame has a CAN_ID - this is ***NOT*** an address.

CAN_IDs are unique (centrally assigned in a network), lowest has highest priority

Nodes can send any CAN_ID, but usually use one CAN_ID for each event



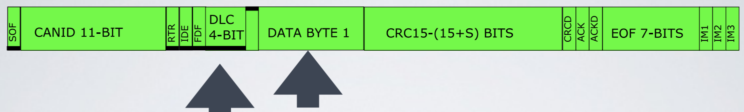
11-bit ID IDE flag indicates if 18 more address bits directly follow the IDE

If IDE = 0, the CAN-ID is 11 bits (CAN 2.0A)

If IDE = 1, the CAN-ID are 29 bits (CAN 2.0B - with 18 bit extension)

63

CAN FRAME FORMAT



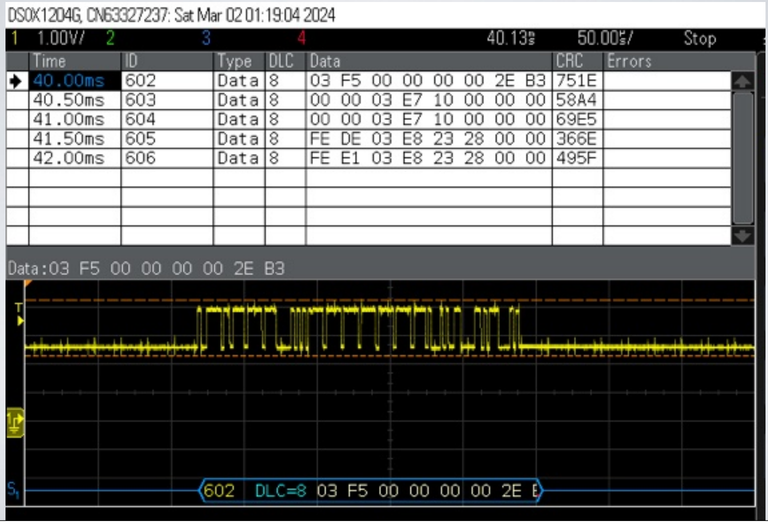
Data 0-8 bytes (0-64b), sent msb first

DLC = Data Length Code 0-8 bytes

- Start of Frame (1b) = 0 - dominant bit!
- Control fields (3b) {RTR; ID (long of short); Reserved/FDF}
  - Data length (4b)
  - Data (0-64b)
  - CRC (15b)
  - CRC delimiter (1b) = 1
- ACK field (2b)
- End of Frame Delimiter (7b) = 1

64

CAN FRAMES



CAN ACK FIELD

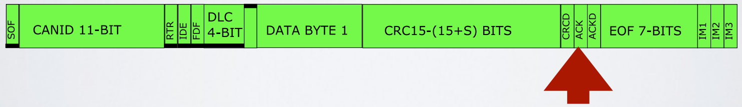
Senders monitor the bus while transmitting...

The sender sends the ACK (recessive) at the end of each frame

- When a receiver sees the end of the message, it sets the ACK bit to dominant

The sender now knows that message has actually been sent by the bus

- If the sender does not see this bit set, it knows there was an "ACK ERROR"!



ACKD = 1

EVERY WORKING BUS >= 2 NODES!

END OF FRAME

Valid frames finish with a series of seven recessive bits, i.e. "idle"

Followed by a 3-bit inter-frame space

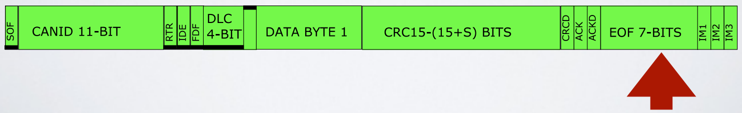
Senders monitor the bus while transmitting...

CRC, DEL, ACK, EOF all need to be seen correctly

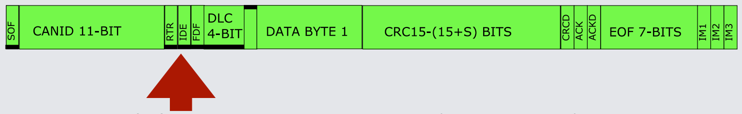
Otherwise the frame is in error

An ERROR FRAME is sent to force all nodes to see the fault

This typically causes the frame to be resent



4 CAN FRAME TYPES



- DATA - Broadcasts data to the bus (most common)
- REMOTE - Request data from a node (see later)
- ERROR FRAME - Reports an error by a node
- OVERLOAD FRAME - Flow control to delay transmission
-

CAR ELECTRICAL SYSTEM

Car electrical system components:

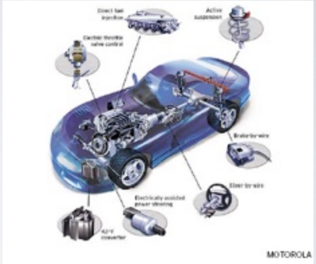
produce and/or consume

- Dashboard
- Engine Control Units (ECUs)
- Anti-lock Braking System (ABS)
- Active Suspension
- Transmission Control

produce

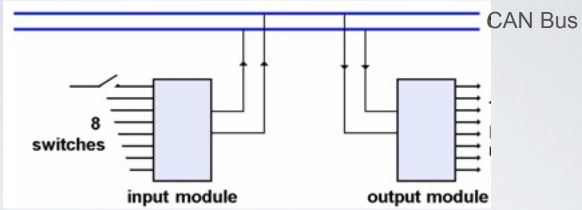
- Lighting
- AirCon
- AirBags
- Power Windows; Power seats; Power Locks; etc

consume



Each component can produce and/or consume CAN frames

USING CAN FRAMES



The CAN ID identifies the message/event

It is not the address of a sender or the receiver

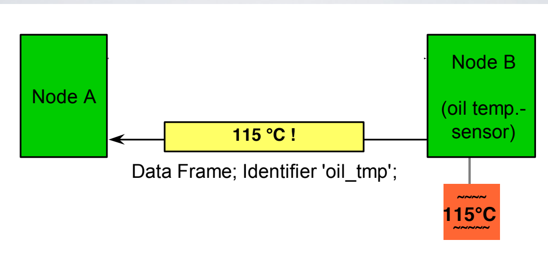
An input module ***produces*** CAN frames

An ID is assigned to each event

An output module ***consumes*** one or more CAN frames

For each configured ID sets an appropriate output

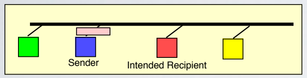
RECEIVING FRAMES/EVENTS



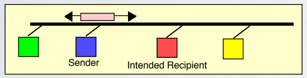
Any node can receive any data (event)

Nodes simply select which messages are of interest and receive them

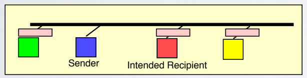
CAN FRAME PROCESSING



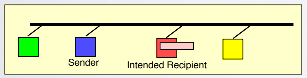
Frames sent with an ID



Frames propagate to all nodes



Nodes sees all frames

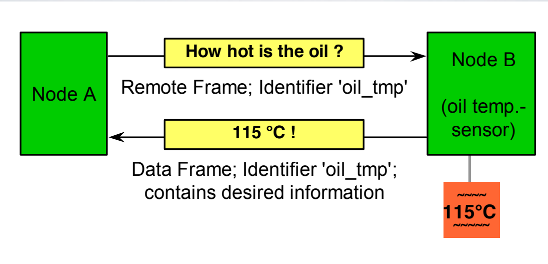


Nodes filter only wanted set of IDs

Some frames are of interest to no nodes at all!

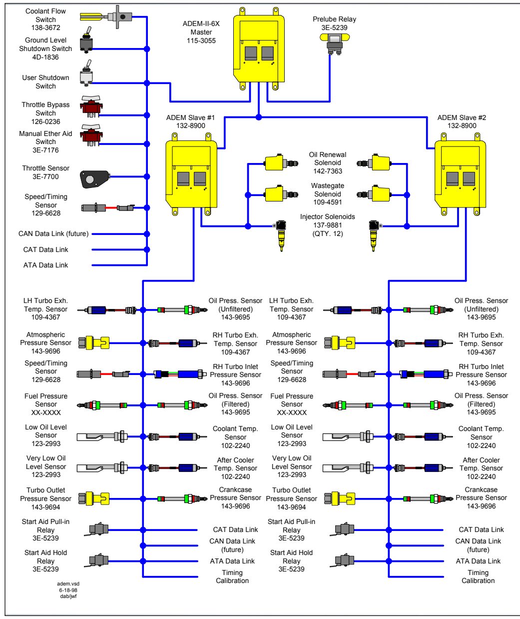
The same frames could be of interest to more than one node

REMOTE FRAMES



Remote Frames are sent in two stages:

- **Remote Frame** sent to ask for a data frame
- **Data Frame** is sent to the CAN bus



Caterpillar
Adem II engine control

3 separate CAN busses

CAN APPLICATIONS

History

- 1983 Original application was for car electrical systems (Robert Bosch)
- 1987 First CAN controllers by Intel and Philips
- 1993 ISO 1198
- 1995 Standards developed from CAN: CANopen; DeviceNet; J.1939

Original applications (~85% market)

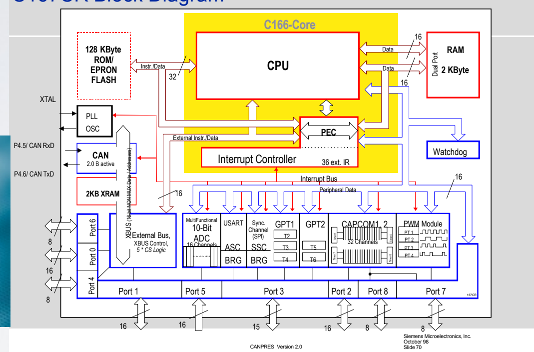
- Cars, trucks, agricultural equipment, etc

Other applications (~15% market)

- Trains, Planes (non safety-critical - e.g. aircon)
- Medical equipment, (XRay, CAT scanners, etc)
- Building automation (e.g. lifts), Office automation
- Household appliances (including coffee makers), Stage control (Chillinet)
- Military vehicles, MILCAM (combines CANopen & J.1939)

ELECTRONIC CONTROL UNIT

C167CR Block Diagram



CAN controllers integrated in a range of microcontrollers (ECU)
- usually use an external transceiver

INDUSTRIAL AUTOMATION

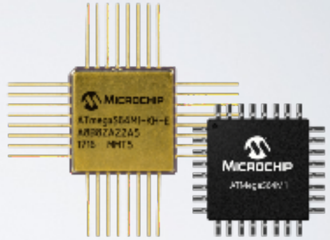
SAM3X 84MHz ARM

- ARM CORTEX M3 Processor
- 2 x CAN 2.0B,
- 10/100 Mbps Ethernet, USB 2.0, I2C, UARTs
- 103 I/O pins

CAN IN AEROSPACE/SPACE

ATmegaS64M1 8-bit megaAVR® MCU

- Operating temperature -55° C to +125° C
- Supports CAN 2.0
- 8-bit UART & SPI
- 11 Channels ADC

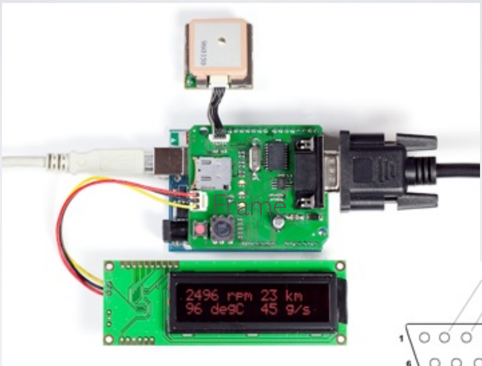


Package

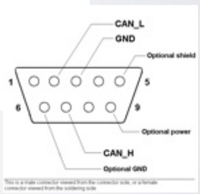
- Plastic aerospace applications
- Ceramic radiation-tolerant for space applications
- Same pinout as automotive-qualified AVR

ARDUINO CAN SHIELD

USB interface



CAN interface (DB-sub9)



Microchip MCP2515 CAN controller
Microchip MCP2551 CAN transceiver
EM406 GPS Interface (asynchronous serial)

BIT STUFFING

CAN bus uses ***synchronous*** transmission

There are no start and stop bauds to frame each byte (e.g., slots in DMX)

There is a transparency problem when sending the same level for many bit periods

- There would be no timing at the receiver to discover sample time for bauds
- CAN uses bit stuffing to prevent this

BIT STUFFING

Senders and receivers count runs of bits sent at the same level

A ***sender*** that sends 5 bits of same polarity, inserts one stuffing bit (of the opposite polarity) before sending the next bit.

These bits are not part of message.

Does not apply to CRC or ACK fields

A ***receiver*** that receives 5 bits of same polarity, deletes the following bit:

The removed stuffing bit must be the opposite polarity (or a STUFF ERROR)

Note this happens ***automatically*** and ensures receivers always see transitions

81

BIT STUFFING EXAMPLES

Examples - can you encode these using bit-stuffing?

- Original data:1010101001
- Original data: 1010000001
- Original data: 1010000011 1

Examples - can you decode these using bit-stuffing?

- Sent on cable as **1010111101**
- Sent on cable as **1010111111**
- Sent on cable as **10101101101**

82

BIT STUFFING II

Examples - can you encode these using bit-stuffing?

- Original data:1010101001
  - sent on cable as 1010101001, received as 1010101001 (not stuffed)
- Original data: 1010000001
  - sent on cable as 10100000 **(1)** 01, received as 1010000001
- Original data: 1010000011 1
  - sent on cable as 10100000 **(1)** 1111, received as 1010000011 1

Examples - can you decode these using bit-stuffing?

- Sent on cable as **1010111101**
  - This was stuffed as 10101111**(0)** 1, received as 101011111
- Sent on cable as **1010111111**
  - This was stuffed as 10101111**(1)** 1, this is a ***stuffing error***
- Sent on cable as **10101101101**
  - This was not stuffed, received as 10101101101

corrected!

83

MAXIMUM LENGTH

The size of a CAN frame is:

- 44 (header size) + 8n (n bytes of payload data)

Bit-stuffing can increase the size of a frame

- (44+8n) <= size after stuffing <= (44+8n)+(34+8n-1)/4

Bit stuffing in CAN ensures there are always some bit transitions

- Bit stuffing adds extra bits before sending and removes them before processing
- Can add up to one bit in five, maximum 20% additional overhead

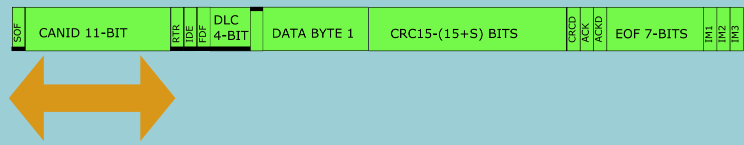
84

ERROR FRAME

When the error flag is set, an Error Frame is sent

- This is six dominant bits followed by eight recessive bits
- This is of course illegal (due to the stuffing rules)
- All nodes recognise this as a fault condition

IDS & CAN ARBITRATION



Arbitration Period*

During first part of message (arbitration period) ***each sender*** monitors bus

If two nodes attempt to simultaneously transmit arbitration rules select lowest message ID, which continues to be sent.

After the arbitration period there can be only one sender!

* Note: When the IDE indicates a long ID,
the arbitration period is extended to cover the entire ID

ARBITRATION PERIOD

The "dominant" values replaces the "recessive" value

- A node continues if it does not see a dominant (0) when it sends a recessive (1)

Other nodes become idle:

- If a node sees a dominant (0) when it wanted to send a zero, it backs-off:
  - It then repeats transmission as soon as idle (CSMA/CD)
  - After arbitration one message is always correctly received

The need for bus monitoring limits the maximum propagation time

This limits the maximum ***allowed bus length***

ARBITRATION EXAMPLE 1

Consider two nodes with two message IDs:

- Node A sends 15 (00000001111)
- Node B sends 16 (00000010000)

A												
B												
bus												

ARBITRATION EXAMPLE 1

Consider two nodes with two message IDs sent at the same time::

- Node A sends CANI-D 15 001111
- Node B sends CAN-ID 16 0010000

Note: Logic 0 is dominant

SFD B backs off

A	0	0	0	0	0	0	0	0	1	1	1	1
B	0	0	0	0	0	0	0	1	-	-	-	-
bus	0	0	0	0	0	0	0	0	1	1	1	1

ARBITRATION EXAMPLE 2

Consider three nodes with three message IDs sent at the same time:

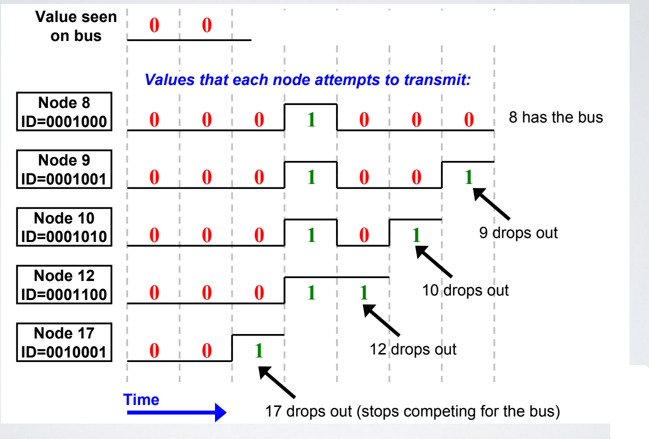
- Node A sends CANI-D 14 0001110
- Node B sends CAN-ID 24 0011000
- Node C sends CAN-ID 7 00000111

Note: Logic 0 is dominant

SFD A,B backs off

A	0	0	0	0	0	0	0	0	1	-	-	-
B	0	0	0	0	0	0	0	1	-	-	-	-
C	0	0	0	0	0	0	0	0	0	1	1	1
bus	0	0	0	0	0	0	0	0	0	1	1	1

LOWEST ID WINS ARBITRATION



High priority messages are assigned lower IDs

ARBITRATION EXAMPLE 2

Consider two nodes with two message IDs:

- Node A sends 685 (01010101101)
- Node B sends 655 (01010001111)

SFD A backs off

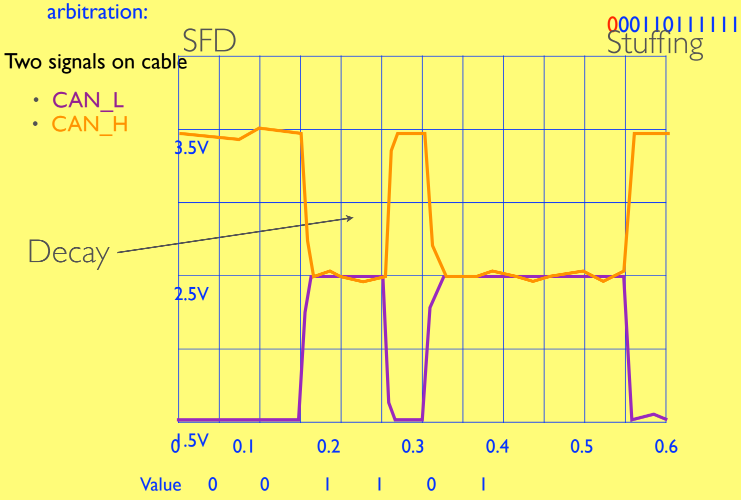
A	0	0	1	0	1	0	1	-	-	-	-	-
B	0	0	1	0	1	0	0	0	1	1	1	1
bus	0	0	1	0	1	0	0	0	1	1	1	1

WAVEFORM

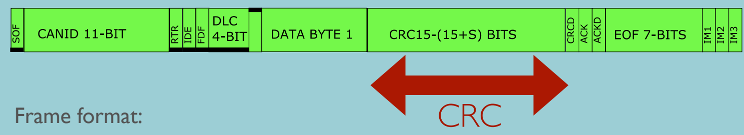
- Sketch the waveform for first 12 bit periods for the sequence that wins the arbitration:

Two signals on cable

- CAN_L
- CAN_H



CAN FRAME CRC



Frame format:

- Start of Frame (1b) = 0 - dominant bit!
- Message ID (11b for CAN 2.0A) - Identifies one of 2048 message
- Control fields (3b) {RTR; ID (long of short); Reserved}
- Data length (4b)
- Data (0-64b)
- CRC (15b)
- CRC delimiter (1b) = 1 - recessive
- ACK field (2b)
- End of Frame Delimiter (7b) = 1
- 1 bit

Discard frames with any formatting errors and/or CRC errors

Cyclic Redundancy Check (CRC)

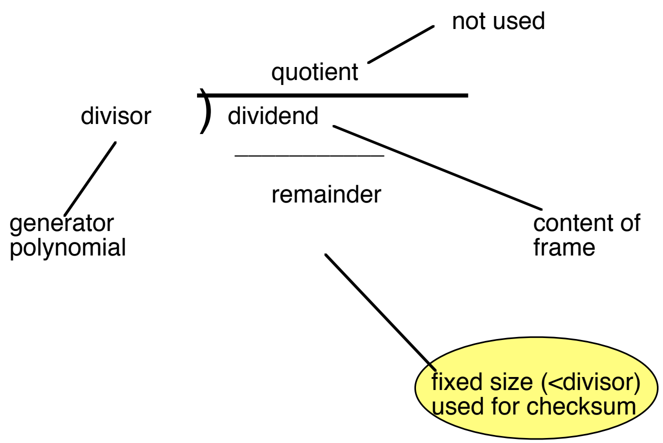
CRC is a form of digital signature (15 bit hash)

- Calculated at the sender & sent
- Re-calculated at the receiver
- Two values compared at receiver
- Able to verify the integrity of the frame

CRC detects:

- Frames that have been corrupted
- Bit timing errors

Galois Field 2 Division



You do not need to reproduce this long division in an exam!

Why Modulo 2 Division?

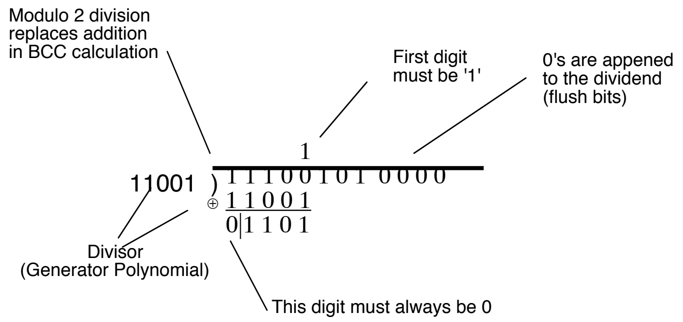
Because the hardware solution is simple!!!!!

Truth Table for Modulo-2 Division (XOR)

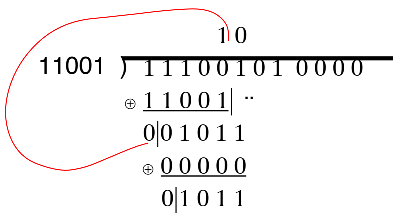
$$0 \oplus 0 = 0$$
$$0 \oplus 1 = 1$$
$$1 \oplus 0 = 1$$
$$1 \oplus 1 = 0$$

All CRC calculations ignore the carry

Example simplified to generate a short (4 bit) CRC

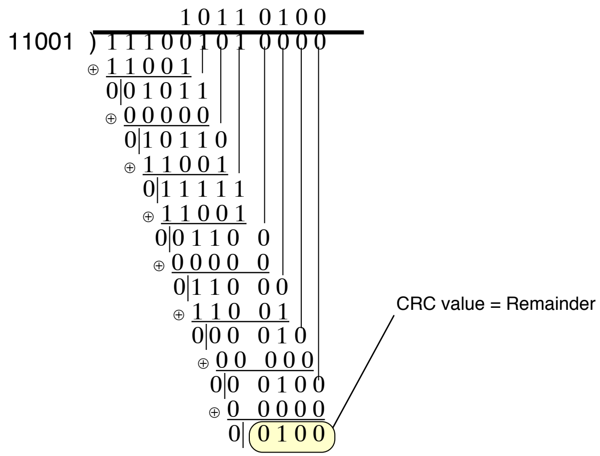


Example simplified to generate a short (4 bit) CRC

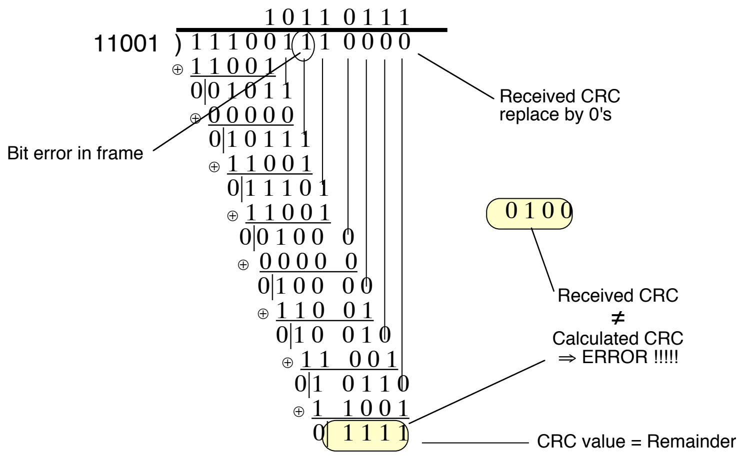


1. Bring next digit of dividend down
2. Copy msb of value to quotient
3. Insert 0 (if quotient 0) or divisor (if quotient 1)
4. Calculate XOR sum
5. Discard msb of value (always 0)

The CRC Value



CRC Value when there was an Error



You do not need to reproduce this long division in an exam!

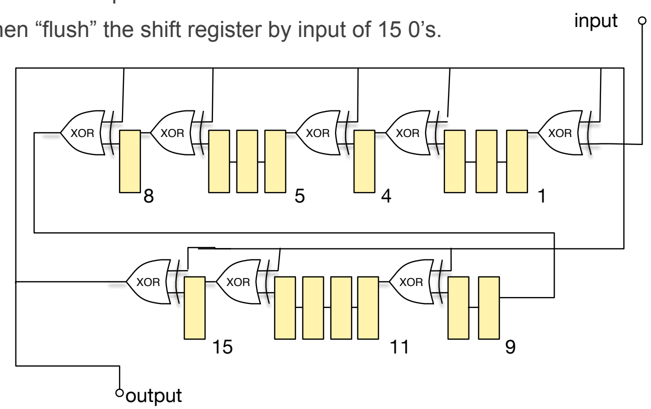
Hardware Example: CRC-15

$x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + x^{0}$

A CRC-15 requires a 15-stage shift register and X-OR gates

Clock each input bit

Then “flush” the shift register by input of 15 0’s.



CRC-15 properties

x^0 Is a parity bit that detects all odd numbers of errors

Consider this CRC-15:

$x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + x^{0}$

The final code has a ***Hamming Distance*** of **six**

This means that **five** randomly distributed bit failures are detectable.

The probability of undetected multiple bit-errors is very low

CRC-15 and CAN

Many systems detect errors using a CRC to and discard corrupted frames.

$x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + x^{0}$

The CAN bus uses the CRC to verify each message

Each message where the received and calculated CRCs do not match causes the CAN receiver to send an ***Error Frame***

HOWEVER although the code has a *Hamming Distance* of six it is less strong than it seems when used with CAN!

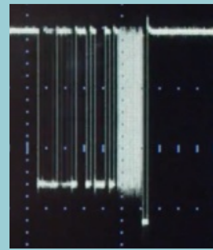
Corruption of a single stuffing bit leads to shifting of the data, effectively inducing a 0.5 error rate, which reduces the power of CRC-15!!

Still, good enough for most applications.

Comparison of Integrity Checking Methods

	Longitudinal Parity	Checksum	CRC
Example	NMEA GPS	DMX SIP Frames	CAN, USB
Hardware Imnplementation	1 XOR gate per bit	Adder per byte	XOR gates and shift register
Software Implementation	XOR instruction + register	Add instruction + register	maths, lookup table + register
Detection of multiple errors	Poor	Better	Good

CAN-FLEXIBLE DATA (FD)

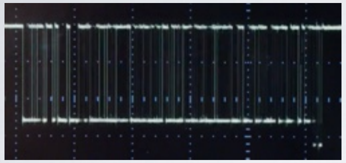


CAN-FD adds new formats

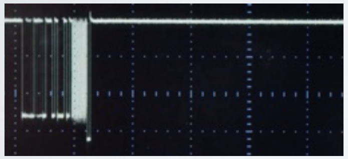
- Extends ***frame size*** up to 64B of data
- Increases ***transmission speed*** *of data*

CAN-FD HIGH RATE

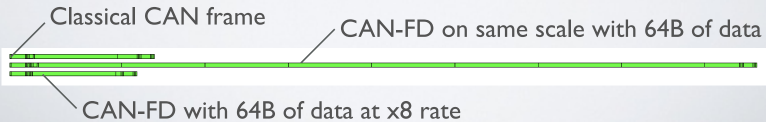
[ISO 11898-1 and ISO 16845-1]



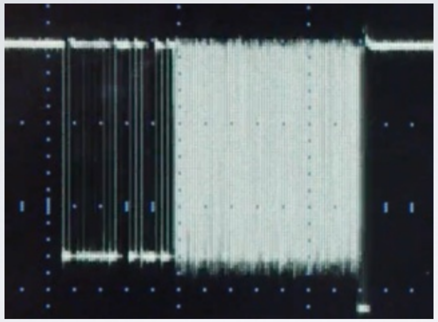
Classical CAN frame with 8 B of data



CAN-FD with 8 B of data



CAN-FD LARGE FRAMES



CAN-FD with 64B of data at x8 rate

Higher baud rate results in lower Eb/No
- and hence more stringent cabling/transceiver design

CAN SUMMARY

High speed control bus

- Supports multiple senders with arbitration
- Supports real-time applications

Low cost chips and cable

- High Reliability
- Plug and Play operation

Extensible

- CANopen extends CAN for other applications
- CAN-FD increases data rate to ~ 5-8 Mbps

109

COMPARE DMX & CAN

	CAN	DMX	RDM
PHY		RS-485 Async	RS-485 Async
Cable		120R STP	120R STP
Direction		Simplex	HDX
Levels		A inverse of B	A inverse of B
Inter-Byte Gap		Idle	Idle
Senders		1	Any with Master
Frame SFD		92 µS Break	92 µS Break
Frame Data Size		1-512B	1-512B
Frame EOF		Idle	Idle

110

COMPARE DMX & CAN

	CAN	DMX	RDM
PHY	RS-485 Sync	RS-485 Async	RS-485 Async
Cable	120R STP	120R STP	120R STP
Direction	HDX	Simplex	HDX
Levels	2.5V for 1 1.5, 3.5 for 0	A inverse of B	A inverse of B
Inter-Byte Gap	No	Idle	Idle
Senders	Any	1	Any with Master
Frame SFD	0	92 µS Break	92 µS Break
Frame Data Size	0-8B	1-512B	1-512B
Frame EOF	111 1111	Idle	Idle

111

Not 2018

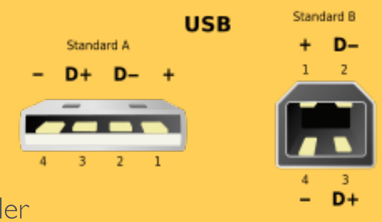
USB

112

UNIVERSAL SERIAL BUS

- About 10,000,000,000 USB ports in use
  - USB 1.1 (1996)
    - Low-speed devices (1.5 Mbps)
    - Full-speed devices (12 Mbps)
  - USB 2.0
    - High-speed devices Up to 480 Mbps
    - Uses same connectors, Speed negotiated device-by-device
  - USB 3
    - Up to about 4 Gbps

USB



- ≤ 127 devices per controller
- Interface:
  - +Data (3), -Data (2) - twisted pair, 90 Ohm
  - Ground (4)
  - +5V Power (1), 500mA (USB2), 900mA (UBS3)

USB SIGNALING

1	0	1	1	0	0	1	0
J	K	K	K	J	K	K	J

- Uses two line **NRZI levels**:
  - J signaled by 0-0.3V; K signaled by 2.8-3.6V
- **Differential**: 0 is signaled by a change in J-K or vice versa

USB FRAMES

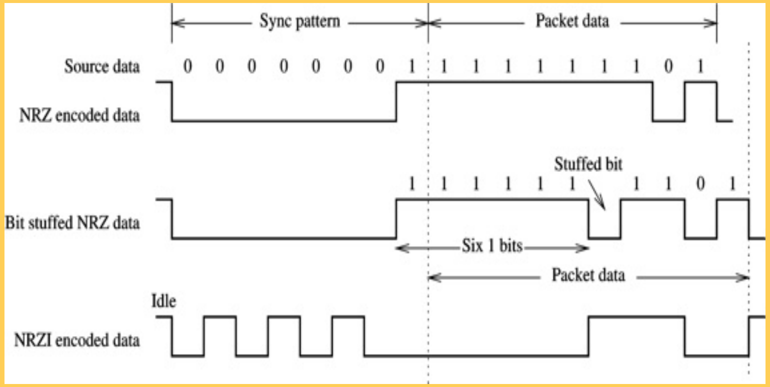
SYNC	PID	DATA	CRC	EOP

- Data formatted in **frames**
  - Controller determines which device transmits
  - Each frame starts with an all '0' Sync Field
    - (8bits low speed, 32 bits high speed)
  - Frame has a packet ID
    - Includes a CRC-16
  - End of packet (EOP_ signaled by 2-bit exception sequence

BIT STUFFING

- **0-bit insertion** (stuffing) used after 6 1's
  - Needed to allow any bit sequence within a frame.
  - More efficient than using start/stop bauds for bytes!
- ***Sender*** physical layer monitors transmission
  - Automatically injects a 0 after 6 1's
- ***Receiver*** physical layer monitors reception
  - Automatically removes a bit after 6 1's
  - If the removed bit is NOT a '0' then the receiver has detected an error condition.

BIT STUFFING



A zero is inserted after every six consecutive 1s

USB (BIT STUFFING)

1) What is the maximum and minimum overhead when using bit stuffing?

2) Determine the sequence of bits when the following data pattern is received over a USB cable: 0111111110100000

3) Explain the implication of bit-errors (inversion) on a stream that uses bit-stuffing. How may the problem that arises be detected?