Omnibus Protocol

Nicholas Simmonds

8th October 2019

The Omnibus protocol is intended to allow a master device (generally assumed to be, but not necessarily, a computer interface device) to communicate with up to 256 slave devices. Communication is always initiated by the master device and is in effect a Remote Procedure Call protocol consisting of an arbitrary length outgoing data segment followed by an arbitrary length incoming data segment. Certain commands are expected to be implemented by all devices irrespective of type. Devices are expected to return a four byte device type identifier in the response to the Get Info standard command which indicates which other commands the device supports.

1 Physical Layer

At present the only physical layer defined uses RS-485 (EIA-485) serial communication however there is no restriction on adding support for other physical media (e.g. fiberoptic cable) in the future. A candidate physical layer requires to be capable of transmitting any 8 bit value as well as one other signal which is distinct from all possible 8 bit values. The RS-485 transport uses two consecutive `BREAK` conditions for this purpose. Physically a three way screw terminal connector should be used labelled `A`, `B` and `G` where `A` and `B` represent the signals of the same name in the RS-485 specification and `G` represents ground. Data is transmitted with eight data bits, one stop bit and no parity bits. The least significant bit is transmitted first in each byte.

2 Timing

Times in the Omnibus protocol are calculated as multiples of the time required to transmit one bit. There are two places in the protocol where timing is important: First when transmitting a multi-byte packet a device should permit no more than 15 bit times between transmitted bytes and the receiving device should discard the packet if there are more than 20 bit times between transmitted bytes (alternatively 25 and 30 bit times respectively from reception of one byte to reception of the next). Second when switching from receiving to transmitting a device should ensure at least 10 bit times after reception of the last byte before transmitting (to give the other device an opportunity to switch from transmitting to receiving).

Additionally a master device should permit 60 bit times when waiting for the start of a packet from a slave device before assuming that a packet has been lost and retransmission is necessary. This permits a device at most 50 bit times after receiving a packet before it must begin transmitting a response.

3 Packet Formats

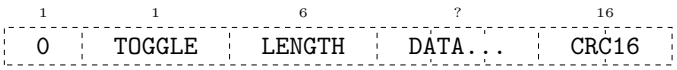
There are four types of packet used in the Omnibus protocol:

Reset

: Each command begins with a reset signal. In the RS-485 physical layer this is accomplished using two consecutive `BREAK` conditions. A device receiving two consecutive `BREAK` conditions should reset packet processing immediately and prepare to receive a new command.

Data

A data packet has the following format:



where `TOGGLE` alternates between `0` and `1` in each successive packet and `LENGTH` stores the length of the packet contents minus 1. As such possible packet lengths range from 1 to 64 bytes. Packets are protected by a CRC which is described in Appendix A. The length/toggle byte is included in the CRC calculation.

Acknowledgement

There are two acknowledgement packets. Both are one byte long and are used in response to a data packet or an end of transmission packet. The acknowledgement must match the toggle bit of the data or end of transmission packet it is sent in response to.

Acknowledgement(0)	`0x85`
Acknowledgement(1)	`0xA3`

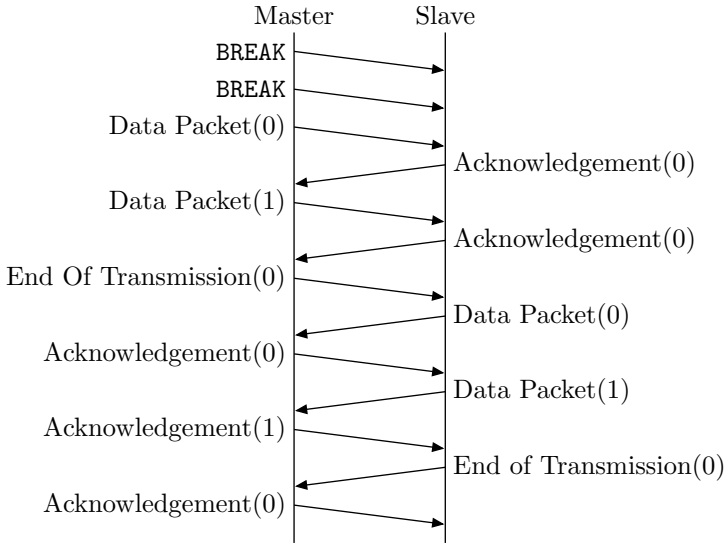
End Of Transmission

There are two end of transmission packets. Both are one byte long and are used instead of a data packet to indicate that this data segment is complete. The end of transmission used must match the toggle bit of the data packet it is replacing.

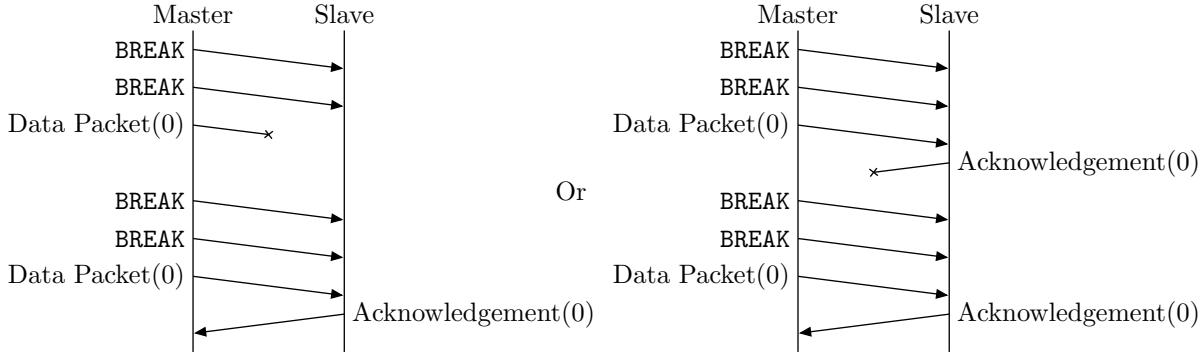
End Of Transmission(0)	`0xD4`
End Of Transmission(1)	`0x8E`

4 Command Transmission

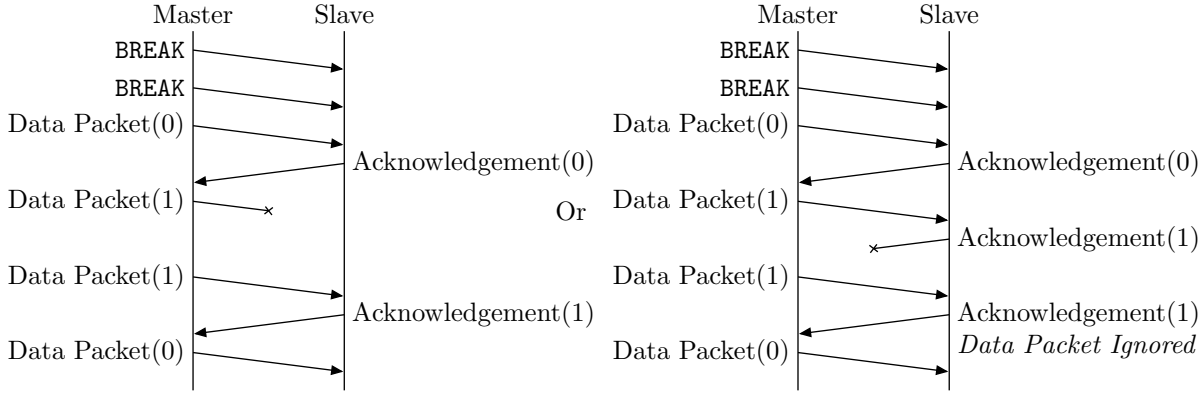
In the ideal case a command consists of two major phases: the Command and Response phases. Each consists of a sequence of data packets to and from the slave device respectively. At the start of each phase the toggle bit is initialised to `0`. A complete transmission looks like:



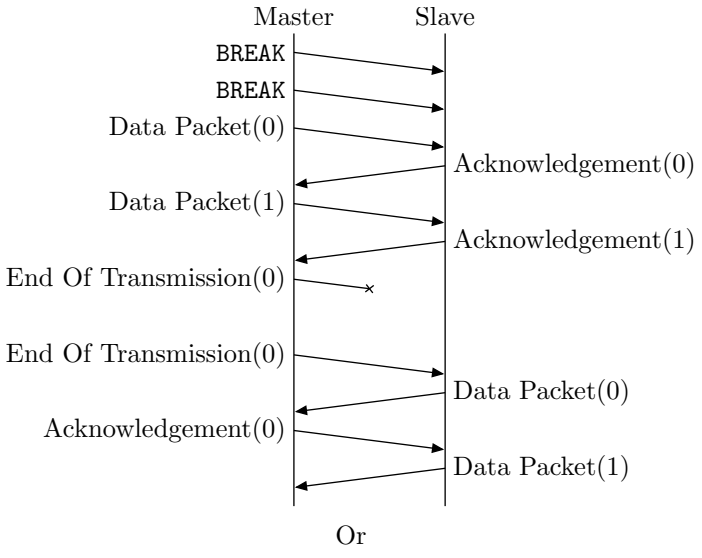
In the event of an error transmitting the first data packet or receiving the first acknowledgement retransmission begins from the initial `BREAK` conditions. This is to handle the case where the initial `BREAK` is corrupted.

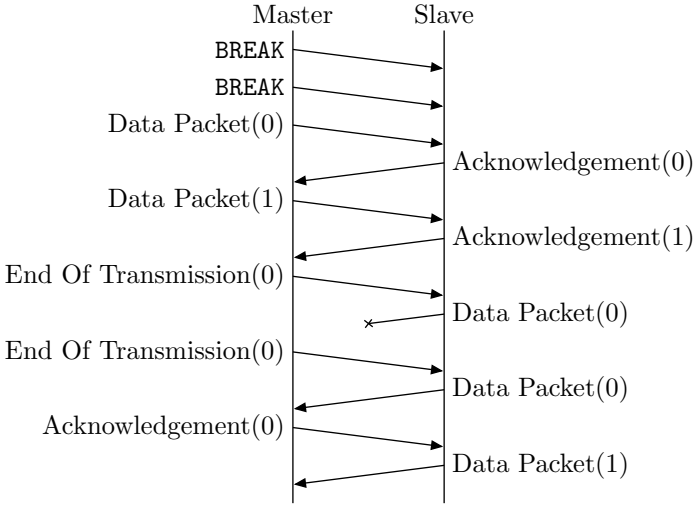


For subsequent packets in the Command Phase loss of the data packet or acknowledgement is handled by a timeout and retransmission by the master device. As such a device receiving a data packet with a valid CRC but the incorrect toggle bit should transmit an appropriate acknowledgement and otherwise ignore the packet. Data packets with incorrect CRC should be ignored in either case.

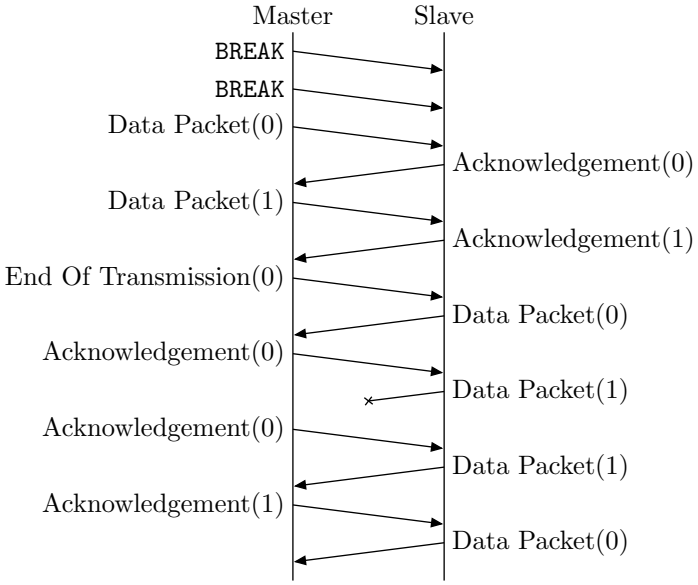


The Command Phase ends when the master device transmits an end of transmission packet instead of a data packet. On receipt of an end of transmission packet the slave device acknowledges by sending the first data packet of the Response Phase. If the end of transmission or the data packet go missing then the end of transmission is resent by the master device. Consequently the slave device must recognise an end of transmission after sending the first data packet of the Response Phase as a request for retransmission.

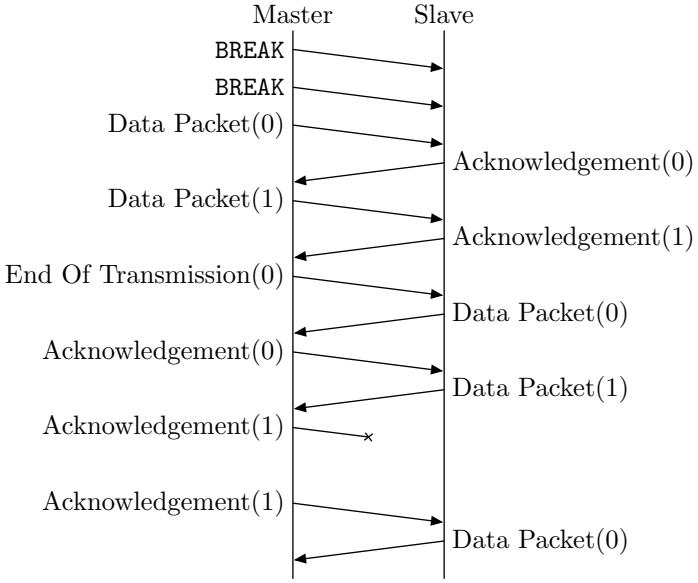




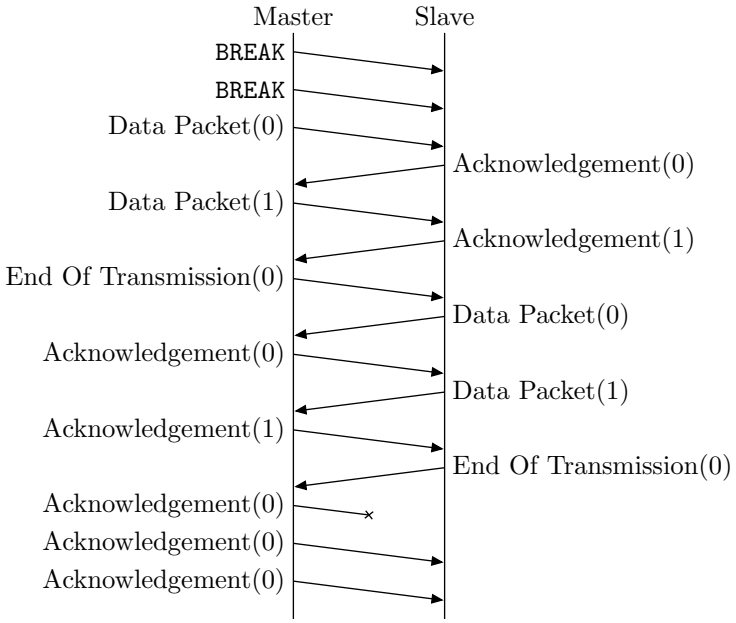
In both phases the master is responsible for requesting retransmissions after packet loss. This reduces the complexity of timeouts in slave devices and permits the master device to abort a transaction at any time without the device continuing to resend packets blindly. Consequently packet retransmission in the Response Phase is handled in a slightly different manner than the Command Phase. The use of two separate acknowledgement packets is required to permit this scheme.



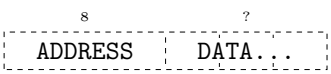
Or



The end of the Response Phase is indicated by the slave device sending an end of transmission packet. The master device responds by sending three consecutive acknowledgement packets. This avoids the situation where the acknowledgement is lost. The master device considers the transaction to be successfully completed after sending the acknowledgement packets. The slave device considers the transaction to be successfully completed when it receives an acknowledgement of the end of transmission packet. At this point the slave device can perform any pending actions required (e.g. changing the address or speed in response to a Set Address or Set Speed command should take place at this point).



5 Command Format

The basic format for commands on the Omnibus is 

though all the commands defined in this specification further restrict the format to 

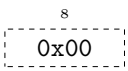
. It is recommended that new commands follow this scheme however the Omnibus specification itself doesn't require this beyond the fact that the mandatory commands use it and all devices must support the mandatory commands. In the following command descriptions the commands are written including `COMMAND` and excluding `ADDRESS`. (e.g. to send a Get Info command to device 2 send 

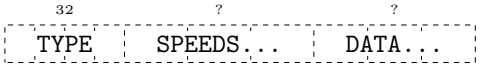
.)

6 Mandatory Commands

All devices must support the following commands:

0x00: Get Info

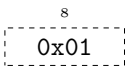
Command: 

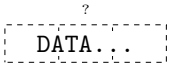
Response: 

`TYPE` is a four byte identifier indicating the device type as well as the format of the remaining data. `SPEEDS` is a variable length bitfield of supported speeds. One bit is set for each supported speed. The bitfield is represented as a sequence of one or more big-endian two byte values. In each value the most significant bit is set to one if the bitfield should be extended by a further two bytes. The remaining 15 bits represent potentially supported speeds. At present only 15 speeds are defined therefore `SPEEDS` will be precisely two bytes long and bit 15 will always be clear. Each device must additionally support a speed of 8kbps (the default speed at startup). The currently defined speeds are:

Bit 0: 16kbps Supported	**Bit 8:** 500kbps Supported
Bit 1: 32kbps Supported	**Bit 9:** 750kbps Supported
Bit 2: 48kbps Supported	**Bit 10:** 1Mbps Supported
Bit 3: 62.5kbps Supported	**Bit 11:** 2Mbps Supported
Bit 4: 100kbps Supported	**Bit 12:** 3Mbps Supported
Bit 5: 125kbps Supported	**Bit 13:** 4Mbps Supported
Bit 6: 250kbps Supported	**Bit 14:** 6Mbps Supported
Bit 7: 375kbps Supported	**Bit 15:** Reserved

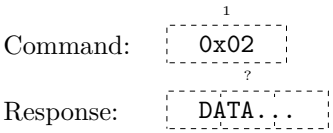
0x01: Get Description

Command: 

Response: 

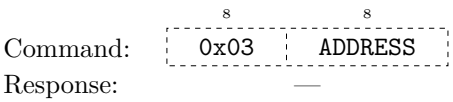
`DATA` represents a variable length string in the UTF-8 encoding. This string's main purpose is to describe to a user the nature of the device in more detail than the Get Info response provides. It should be human readable and should ideally include the manufacturer's name and the product's name/code. Additionally a URL where more information can be obtained about the device and some form of version indication are sensible elements to include. This string is also intended to be used by firmware update tools to match firmware files with devices. An example description for a RailCom reader might be: "`DCC4PC RailCom Reader 1.0 (www.dcc4pc.co.uk)`".

0x02: Get Serial Number



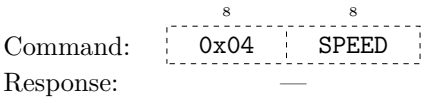
This command (which is optional but must always return a serial number or be ignored) should return a unique string in `DATA` which may be used to differentiate between otherwise identical devices. Software should not rely on the serial number for persistent device identification (the address serves that purpose) but should make it available to the user. This is an optional mechanism which a manufacturer may choose to provide to aid in tracking the history of specific hardware (e.g. if a particular board has been sent in for repair before). If absent there should be no impact on the user.

0x03: Set Address



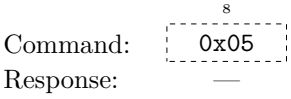
`ADDRESS` is the new address the device should use. The address takes effect on successful completion of the transaction, before the next transaction begins.

0x04: Set Speed



`SPEED` is one of the supported speeds identified in the response to the Get Info command. The `SPEED` value is equal to the bit position of the desired speed plus one (e.g. a `SPEED` value of 7 would indicate a desired speed of 250kps). To select 8kbps use a `SPEED` value of 0. This command always succeeds if the `SPEED` is listed as supported in the response to the Get Info command. If an invalid `SPEED` value is sent then the device should ignore the remainder of the transaction. The new speed setting will take effect on successful completion of the transaction, before the next transaction begins.

0x05: Begin Download

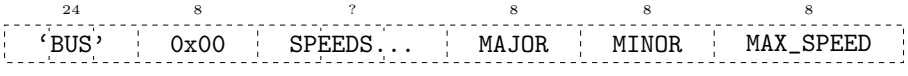


It is permissible for a device not to support this command (in which case it should be ignored). Otherwise it should put the device into Download Mode. Download Mode is indicated by Get Info returning '`DNLD`' and the device supporting the commands described in the '`DNLD`' Device Type Description. If a device is already in Download Mode then this command should be ignored.

All other commands are free for use by other device types.

7 `BUS` Device Type Definition

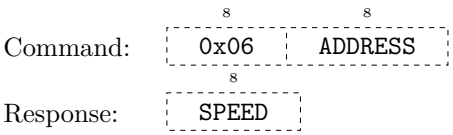
7.1 Get Info Response



The `MAJOR` and `MINOR` parameters specify the version of the `DNLD` specification supported. The version described by this document is 1.1. The `MAX_SPEED` parameter indicates the highest speed supported by the bus.

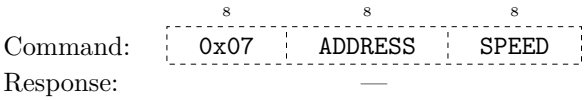
7.2 Additional Commands

0x06: Get Speed



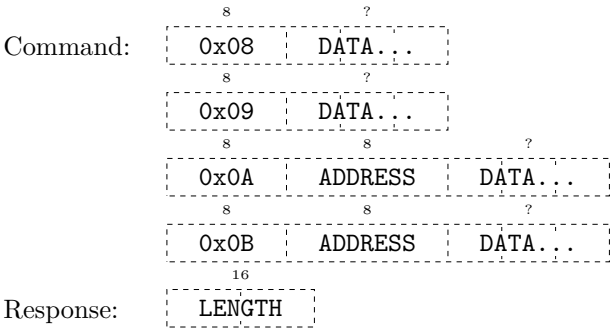
Returns the `SPEED` used when sending commands to `ADDRESS`.

0x07: Set Speed



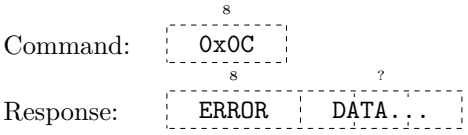
Sets the speed used when sending commands to `ADDRESS` to `SPEED`. It is an error for `SPEED` to be greater than `MAX_SPEED`.

0x08: Send Command



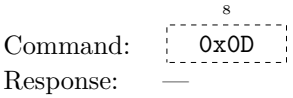
These commands are used to send a command to a child device. The commands return the length accepted by the command. The `0x0A` command starts a command and will cancel a previous command in progress. `0x08` continues the command and `0x09` ends the command. `0x0B` both starts and ends a command.

0x0C: Get Response



Fetches the response returned by the previous command. If `ERROR` is "Incomplete Response" then more data is coming. It is not an error for `DATA` to be empty.

0x0D: Reset Speeds (Version 1.1)



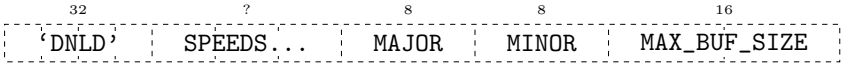
Sends a global 'Reset Speed to 0' sequence. This will reset the speeds of all attached devices to a known state. This makes probing for attached devices quicker as there is no longer a need to test each possible speed. In practice it is seldom necessary to send this command explicitly, as `BUS` devices supporting this command send the sequence automatically at start-up. Consequently, when testing for the presence of a device, it is sufficient to use the current speed of the device and, if the current speed is not zero, retry with speed zero.

7.3 Error Codes

0 Success
1 Incomplete Response
2 Command Failed

8 DNLD Device Type Definition

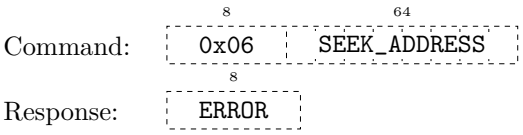
8.1 Get Info Response



The MAJOR and MINOR parameters specify the version of the DNLD specification supported. The version described by this document is 1.0. The MAX_BUF_SIZE parameter indicates the largest data buffer which can be written in a single operation.

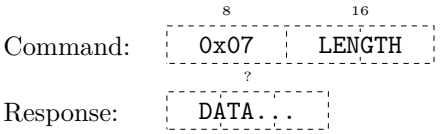
8.2 Additional Commands

0x06: Seek



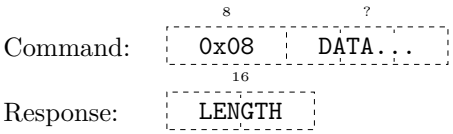
Sets the current read/write pointer to SEEK_ADDRESS. Prior to executing a Seek command the position of the read/write pointer is undefined. The Seek command also flushes any pending writes. There is no way to request from a device the address ranges which are programmable as a firmware developer would require access to this information in order to write a firmware update.

0x07: Read



LENGTH is the length (in bytes) of the data to return. Data will be returned starting from the current location of the read/write pointer which will be updated upon successful completion of the transaction. The data returned may be shorter than requested but never longer. Returning a zero-length response indicates reaching the end of the current memory region.

0x08: Write



This command writes the byte string DATA to the device at the current position of the read/write pointer. The command will buffer as much of the data as possible and return the length of the data actually used in LENGTH. The MAX_BUF_SIZE parameter in the Get Info response is an indicator of the largest possible Write which could be accepted completely. It should be noted that Write may still return a LENGTH less than MAX_BUF_SIZE if there is already buffered data waiting to be written. Once the buffer is full the device should begin a flash write operation and return a LENGTH of 0 in response to any further Write commands until the flash write is complete.

0x09: Erase

Command: | 8 | 0x09 |

Response: | 8 | ERROR |

Erases the device in preparation for writing.

0x0A: Exit Download Mode

Command: | 8 | 0x0A |

Response: —

After this command the device should leave download mode and resume normal operations. If the code has been modified then any final actions required to mark the code as complete should be performed now. The device should perform a reboot in response to this command, consequently the device speed will be reset to 8kbps.

8.3 Error Codes

0 Success
1 Busy (e.g. flash write/erase in progress)
2 Invalid (e.g. seek to non-writable address)

9 RCRD Device Type Definition

9.1 Get Info Response

32	?	8	8	8	8
'RCRD'	SPEEDS...	MAJOR	MINOR	NR_INPUTS	ENCODINGS

MAJOR and MINOR represent the version of the RCRD specification the device supports. This specification defines version 1.0. NR_INPUTS is one less than the number of track sections that the device can monitor and ENCODINGS indicates the device's capabilities:

Bit 0: Device supports the Raw RailCom encoding.
Bit 1: Device supports the Cooked RailCom encoding.

The remaining bits are reserved and will be used to define other encoding schemes in the future. A device which doesn't set bit 0 is unsupported by this version of the RCRD specification.

9.2 Additional Commands

0x06: Select Encoding

	8	8
Command:	0x06	ENCODING

Response: —

ENCODING identifies the desired data format. At present only two encodings are defined: Raw (0) and Cooked (1). If ENCODING is not supported the command should be ignored.

0x07: Get Enabled Inputs

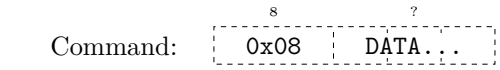
Command: | 8 | 0x07 |

Response: | ? | DATA... |

This command will return the state of each of the inputs the device possesses. Each input is encoded as a single bit where 1 indicates the input is enabled and will contribute data to the

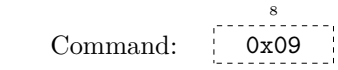
output of subsequent Get Data calls and `0` indicates the input is disabled and contributes nothing. The first byte will hold information on the first eight inputs (with the first input in bit 0 and the eighth in bit 7), the second byte the next eight inputs and so on. This command should always return precisely the number of bytes required to encode all the inputs the device supports (unused bits are set to `0`).

0x08: Set Input States



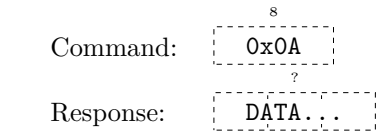
`DATA` represents the desired state of the inputs in the same format as is returned by Get Input States. There is no mechanism to update just a subset of states and as with Get Input States `DATA` should have precisely the length required for the number of inputs specified in the response to Get Info. Extra bits should be set to `0` however devices will ignore the value of these bits.

0x09: Reset Encoding State



This command will reset any saved state used by an encoding and force the next packet of RailCom data to be transmitted in full. The nature of the saved state is dependent on the nature of the encoding however in all cases calling this command will make all data returned by subsequent Get Data calls independent of data returned before Reset Encoding State was called. Calling this command before calling Get Data for the first time is advised.

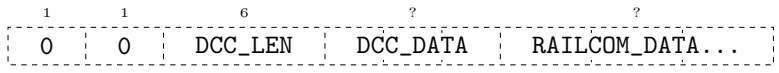
0x0A: Get Data



This command will return a block of data in the selected RailCom encoding. The specifics of the returned data are dependent on the encoding in question.

9.3 Raw RailCom Encoding

RailCom Data Packets come in three forms:

1. 

   This packet type indicates the RailCom reader received a DCC packet `DCC_DATA` (excluding the XOR error detection byte) of length `DCC_LEN` + 1. The RailCom data received immediately following the DCC packet is encoded in `RAILCOM_DATA` in the format described below. Data is only included for those inputs whose state (one of Unoccupied, Occupied With No RailCom Data, RailCom Data In Direction A or RailCom Data In Direction B) indicates RailCom data is expected. The state of the inputs is assumed to be unchanged from the previous packet.

2. 

   This packet type is similar to the above packet type however explicitly encodes the state of each input in `STATES` rather than inheriting the information from previous packets. Each enabled input contributes two bits which encode whether the input is Unoccupied (`00`), Occupied With No RailCom Data (`01`), RailCom Data With Orientation A (`10`) or RailCom Data With Orientation B (`11`). The first enabled input is stored in bits 1:0 of the first byte, the next enabled input is stored in bits 3:2 of the first byte, the fifth enabled input is stored in bits 1:0 of the second byte, etc. Inputs which are not enabled contribute nothing. The length of `STATES` can be extrapolated from the number of currently enabled inputs as returned by Get Enabled Inputs.

3.

1	7	?
1	CMD	DATA...

This packet type indicates a special command. The special command is indicated by the 7 bit value `CMD`. The only supported special command at present is `0x81` which takes no extra `DATA`. This command indicates that the device's internal buffer has overflown and therefore some packets may have been lost. As the majority of RailCom packets are self-contained this can generally be ignored however an Address Part 1 packet before an overflow should not be paired with an Address Part 2 packet after an overflow.

The `RAILCOM_DATA` in the above mentioned packet types is encoded using a scheme designed to minimise the length of each RailCom packet. A key element of this is a repetition encoding mechanism where the device can indicate that the content of the RailCom data for an input is a repeat of the first, second or third previous RailCom data for that input. As such it is necessary for a master device to keep track of this data in order to successfully interpret a packet.

First for each input which has RailCom data a two bit field is encoded in a manner similar to the `STATES` field of the second packet type described above and padded to a whole number of bytes. The possible values are No Duplicate (`00`), Duplicate Of Previous Packet (`01`), Duplicate Of Second Previous Packet (`10`) and Duplicate Of Third Previous Packet (`11`).

Next for each input which is marked as No Duplicate a four bit value is encoded in a manner similar to the `STATES` field however with the first input encoded in bits 3:0, the second in bits 7:4, the third in bits 3:0 of the second byte, etc. padded to a whole number of bytes. This field stores the length of RailCom data for the input.

Lastly for each input marked as No Duplicate the RailCom data is stored.

Example: Assume a RailCom Reader with 16 inputs, all enabled, returns the following packet in response to a Get Data command:

8	8	8	8	8	8	8	8	8
0x42	0x01	0x3F	0xA8	0x90	0x90	0xC0	0x00	0x00
0x42	0x02	0xA3	0xAC	0x8B	0x69	0x4E	0xA5	0x99
0x9A	0x01	0x01	0x91	0x04	0x22	0x99	0xA5	0xA3
0xAC	0x81	0x42	0x03	0x3F	0xBC	0x80	0x90	0xC0
0x00	0x20	0x24	0x8E	0xD4	0x4E	0x59	0xA3	0xAC

Decode Procedure:

1. Examine first byte: `0x42`. Matches

1	1	6
0	1	DCC_LEN

$\therefore$ is type 2 packet.

2. Decode packet:

1	1	6	?	?	?
0	1	DCC_LEN	DCC_DATA	STATES	RAILCOM_DATA...

```
DCC_LEN    = 2
DCC_DATA   = 0x01 0x3F 0xA8
STATES     = 0x90 0x90 0xC0 0x00
```

3. Decode States:

```
          0x90
        = 10 01 00 00
Input 0  = 00 (Unoccupied)
Input 1  = 00 (Unoccupied)
Input 2  = 01 (Occupied With No RailCom Data)
Input 3  = 10 (RailCom Data With Orientation A)
          0x90
        = 10 01 00 00
Input 4  = 00 (Unoccupied)
Input 5  = 00 (Unoccupied)
Input 6  = 01 (Occupied With No RailCom Data)
Input 7  = 10 (RailCom Data With Orientation A)
          0xC0
        = 11 00 00 00
Input 8  = 00 (Unoccupied)
Input 9  = 00 (Unoccupied)
Input 10 = 00 (Unoccupied)
Input 11 = 11 (RailCom Data With Orientation B)
          0x00
        = 00 00 00 00
Input 12 = 00 (Unoccupied)
Input 13 = 00 (Unoccupied)
Input 14 = 00 (Unoccupied)
Input 15 = 00 (Unoccupied)
```

4. Decode Duplicate information for 3 inputs (3, 7 and 11):

```
          0x00
        = 00 00 00
Input 3  = 00 (No Duplicate)
Input 7  = 00 (No Duplicate)
Input 11 = 00 (No Duplicate)
```

5. Decode Length information for 3 inputs (3, 7 and 11):

```
          0x42
Input 3  = 2
Input 7  = 4
          0x02
Input 11 = 2
```

6. Extract 3 RailCom packets:

```
Input 3  = 0xA3 0xAC
Input 7  = 0x8B 0x69 0x4E 0xA5
Input 11 = 0x99 0x9A
```

7. Decode RailCom as specified in NMRA RP-9.3.2:

Input 3 = 0xA3 0xAC
= 0x04 0x00
= 00 0100 00 0000
= 0001 0000 0000
= 0x100
= Address Part 1: Short Address (No Consist)
Input 7 = 0x8B 0x69 0x4E 0xA5
= 0x0E 0x17 0x21 0x03
= 00 1110 01 0111 10 0001 00 0011
= 0011 1001 0111 1000 0100 0011
= 0x397 0x843
= Decoder 1 Actual Speed: 23
Decoder 1 Temperature: 67
Input 11 = 0x99 0x9A
= 0x08 0x07
= 00 1000 00 0111
= 0010 0000 0111
= 0x207
= Address Part 2: 7

8. Examine next byte: 0x01. Matches

1	1	6
0	0	DCC_LEN

∴ is type 1 packet.

9. Decode packet:

1	1	6	?	?
0	0	DCC_LEN	DCC_DATA	RAILCOM_DATA...

DCC_LEN = 1
DCC_DATA = 0x01 0x91

10. States unchanged from previous packet: inputs 2 and 6 = Occupied, inputs 3 and 7 = RailCom Data With Orientation A and input 11 = RailCom Data With Orientation B.

11. Decode Duplicate information for 3 inputs (3, 7 and 11):

0x04
= 00 01 00
Input 3 = 00 (No Duplicate)
Input 7 = 01 (Duplicate Of Previous Packet)
Input 11 = 00 (No Duplicate)

12. Decode Length information for 2 inputs (3 and 11):

0x22
Input 3 = 2
Input 11 = 2

13. Extract 2 RailCom packets:

Input 3 = 0x99 0xA5
Input 11 = 0xA3 0xAC

14. Decode RailCom as specified in NMRA RP-9.3.2:

Input 3 = 0x99 0xA5
= 0x08 0x03
= 00 1000 00 0011
= 0010 0000 0011
= 0x203
= Address Part 2: 3
Input 7 = Decoder 1 Actual Speed: 23
Decoder 1 Temperature: 67
Input 11 = 0xA3 0xAC
= 0x04 0x00
= 00 0100 00 0000
= 0001 0000 0000
= 0x100
= Address Part 1: Short Address (No Consist)

15. Examine next byte: 0x81. Matches

1 (1)	CMD (7)

∴ is type 3 packet.

16. Decode packet:

1 (1)	CMD (7)	DATA... (?)

0x81
= Overflow

17. Examine next byte: 0x42. Matches

0 (1)	1 (1)	DCC_LEN (6)

∴ is type 2 packet.

18. Decode packet:

0 (1)	1 (1)	DCC_LEN (6)	DCC_DATA (?)	STATES (?)	RAILCOM_DATA... (?)

DCC_LEN = 2
DCC_DATA = 0x03 0x3F 0xBC
STATES = 0x80 0x90 0xC0 0x00

19. Decode States:

0x80
= 10 00 00 00
Input 0 = 00 (Unoccupied)
Input 1 = 00 (Unoccupied)
Input 2 = 00 (Unoccupied)
Input 3 = 10 (RailCom Data With Orientation A)
0x90
= 10 01 00 00
Input 4 = 00 (Unoccupied)
Input 5 = 00 (Unoccupied)
Input 6 = 01 (Occupied With No RailCom Data)
Input 7 = 10 (RailCom Data With Orientation A)

0xC0
= 11 00 00 00
Input 8 = 00 (Unoccupied)
Input 9 = 00 (Unoccupied)
Input 10 = 00 (Unoccupied)
Input 11 = 11 (RailCom Data With Orientation B)
0x00
= 00 00 00 00
Input 12 = 00 (Unoccupied)
Input 13 = 00 (Unoccupied)
Input 14 = 00 (Unoccupied)
Input 15 = 00 (Unoccupied)

20. Decode Duplicate information for 3 inputs (3, 7 and 11):

0x20
= 10 00 00
Input 3 = 00 (No Duplicate)
Input 7 = 00 (No Duplicate)
Input 11 = 10 (Duplicate Of Second Previous Packet)

21. Decode Length information for 2 inputs (3 and 7):

    0x24
    Input 3 = 4
    Input 7 = 2

22. Extract 2 RailCom packets:

    Input 3 = 0x8B 0xD4 0x4E 0x59
    Input 7 = 0xA3 0xAC

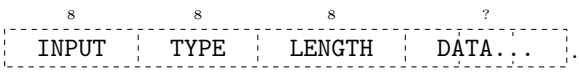
23. Decode RailCom as specified in NMRA RP-9.3.2:

    Input 3 = 0x8B 0xD4 0x4E 0x59
    = 0x0E 0x2D 0x21 0x1D
    = 00 1110 10 1101 10 0001 01 1101
    = 0011 1010 1101 1000 0101 1101
    = 0x3AD 0x85D
    = Decoder 3 Actual Speed: 45
    Decoder 3 Temperature: 93
    Input 7 = 0xA3 0xAC
    = 0x04 0x00
    = 00 0100 00 0000
    = 0001 0000 0000
    = 0x100
    = Address Part 1: Short Address (No Consist)
    Input 11 = Address Part 2: 7

In summary: Section 3 contains decoder 3 which has a speed of 60, is driving forwards, has orientation A, actual speed 45 and temperature 93. Section 7 contains decoder 1 which has a speed of 40, is driving forwards, has orientation A, actual speed 23 and temperature 67. Section 11 likely contains decoder 7 however has yet to be reliably identified. Additionally section 6 contains something which is drawing current: either a lit coach or a non-RailCom-enabled decoder. Section 2 used to contain something but is now unoccupied. All other sections are unoccupied.

9.4 Cooked RailCom Encoding

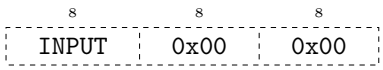
RailCom Data Packets have the general form



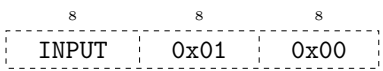
.

The following packet types are supported:

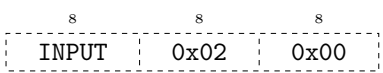
0x00: Input is Unoccupied



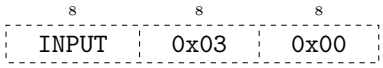
0x01: Input is Occupied



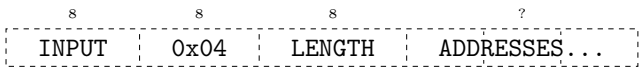
0x02: Input is Occupied with RailCom Orientation A



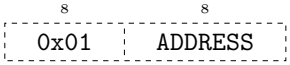
0x03: Input is Occupied with RailCom Orientation B

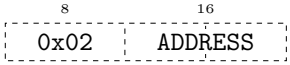


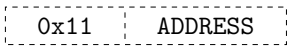
0x04: Address Changed

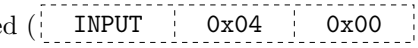


`ADDRESSES` is a list of address sub-packets listing all of the decoder addresses currently observed by this input. Address sub-packets are formatted as 

. Where `TYPE` indicates the format of `ADDRESS`. The low four bits of `TYPE` encode the length of `ADDRESS`. A short address is encoded as 

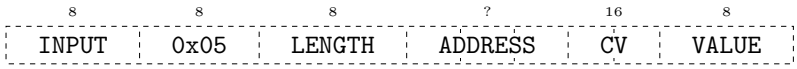
, a long address as 

and a consist address as 

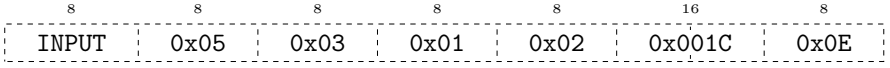
. If no addresses are observed by this input then an empty packet should be transmitted (

).

0x05: CV Value



`ADDRESS` is a single address sub-packet as described above. `CV` is the 10 bit CV number as specified in RP-9.2.1 (e.g. CV#1 has `CV` as 0) and `VALUE` is the new value of the CV.

Example: 

= “Locomotive 2 CV#29 = 14”

A CRC Calculation

CRC calculation is performed using the polynomial:

$$x^{16} + x^{14} + x^{13} + x^{12} + x^{10} + x^8 + x^6 + x^4 + x^3 + x^1 + x^0$$

Bytes are processed in the order transmitted from most significant bit to least significant bit. The CRC is initialised to `0xFFFF`, which requires initialising to `0x0341` in table driven calculators like the following (which may be used for convenience):

```
#define CRC_INIT 0x0341

static const unsigned short crc_table[256] = {
        0x0000,0x755b,0xeab6,0x9fed,0xa037,0xd56c,0x4a81,0x3fda,
        0x3535,0x406e,0xdf83,0xaad8,0x9502,0xe059,0x7fb4,0x0aef,
        0x6a6a,0x1f31,0x80dc,0xf587,0xca5d,0xbf06,0x20eb,0x55b0,
        0x5f5f,0x2a04,0xb5e9,0xc0b2,0xff68,0x8a33,0x15de,0x6085,
        0xd4d4,0xa18f,0x3e62,0x4b39,0x74e3,0x01b8,0x9e55,0xeb0e,
        0xe1e1,0x94ba,0x0b57,0x7e0c,0x41d6,0x348d,0xab60,0xde3b,
        0xbebe,0xcbe5,0x5408,0x2153,0x1e89,0x6bd2,0xf43f,0x8164,
        0x8b8b,0xfed0,0x613d,0x1466,0x2bbc,0x5ee7,0xc10a,0xb451,
        0xdcf3,0xa9a8,0x3645,0x431e,0x7cc4,0x099f,0x9672,0xe329,
        0xe9c6,0x9c9d,0x0370,0x762b,0x49f1,0x3caa,0xa347,0xd61c,
        0xb699,0xc3c2,0x5c2f,0x2974,0x16ae,0x63f5,0xfc18,0x8943,
        0x83ac,0xf6f7,0x691a,0x1c41,0x239b,0x56c0,0xc92d,0xbc76,
        0x0827,0x7d7c,0xe291,0x97ca,0xa810,0xdd4b,0x42a6,0x37fd,
        0x3d12,0x4849,0xd7a4,0xa2ff,0x9d25,0xe87e,0x7793,0x02c8,
        0x624d,0x1716,0x88fb,0xfda0,0xc27a,0xb721,0x28cc,0x5d97,
        0x5778,0x2223,0xbdce,0xc895,0xf74f,0x8214,0x1df9,0x68a2,
        0xccbd,0xb9e6,0x260b,0x5350,0x6c8a,0x19d1,0x863c,0xf367,
        0xf988,0x8cd3,0x133e,0x6665,0x59bf,0x2ce4,0xb309,0xc652,
        0xa6d7,0xd38c,0x4c61,0x393a,0x06e0,0x73bb,0xec56,0x990d,
        0x93e2,0xe6b9,0x7954,0x0c0f,0x33d5,0x468e,0xd963,0xac38,
        0x1869,0x6d32,0xf2df,0x8784,0xb85e,0xcd05,0x52e8,0x27b3,
        0x2d5c,0x5807,0xc7ea,0xb2b1,0x8d6b,0xf830,0x67dd,0x1286,
        0x7203,0x0758,0x98b5,0xedee,0xd234,0xa76f,0x3882,0x4dd9,
        0x4736,0x326d,0xad80,0xd8db,0xe701,0x925a,0x0db7,0x78ec,
        0x104e,0x6515,0xfaf8,0x8fa3,0xb079,0xc522,0x5acf,0x2f94,
        0x257b,0x5020,0xcfcd,0xba96,0x854c,0xf017,0x6ffa,0x1aa1,
        0x7a24,0x0f7f,0x9092,0xe5c9,0xda13,0xaf48,0x30a5,0x45fe,
        0x4f11,0x3a4a,0xa5a7,0xd0fc,0xef26,0x9a7d,0x0590,0x70cb,
        0xc49a,0xb1c1,0x2e2c,0x5b77,0x64ad,0x11f6,0x8e1b,0xfb40,
        0xf1af,0x84f4,0x1b19,0x6e42,0x5198,0x24c3,0xbb2e,0xce75,
        0xaef0,0xdbab,0x4446,0x311d,0x0ec7,0x7b9c,0xe471,0x912a,
        0x9bc5,0xee9e,0x7173,0x0428,0x3bf2,0x4ea9,0xd144,0xa41f,
};

unsigned short calc_crc(unsigned short in,unsigned char byte)
{
        return (in << 8) ^ crc_table[(in >> 8) ^ byte];
}
```

B USB Interface

Any Omnibus device can optionally support a direct USB connection. In this mode, the device uses the USB CDC class driver (ACM subclass) to communicate with the PC. Consequently they appear as serial

devices (`COM`x under Windows and `/dev/ttyACM`x under Linux). They may be differentiated from other serial devices by checking the USB Vendor and Product IDs. A DCC_4PC USB device will have Vendor ID `0x16D0` and Product ID `0x064D`.

To communicate with such devices, use Omnibus commands as described in the preceeding sections. As only a single device is attached, the `ADDRESS` byte at the start of each command should be omitted. Additionally, the mandatory commands Set Address and Set Speed are illegal (as they make no sense in this context). For example a DCC_4PC Computer Interface Device might respond to a command

8
`0x00`

by sending

24	8	16	8	8	8
`'BUS'`	`0x00`	`0x0000`	`0x01`	`0x00`	`0x0F`

. To talk to other devices attached to a Computer Interface Device, use the commands in section 7 to send and receive Omnibus commands.

When opening the device, set flow control to none and ignore the baud rate. Commands are started by raising the `DTR` line and ended by lowering the `DTR` line. The end of response is indicated by `DSR` lowering and an error is signalled by `RING` being raised. Care must be taken to ensure that the serial driver's buffering doesn't change the order of events as observed by the device. In summary, the procedure to send a command is as follows:

1. Set `DTR` High
2. Write command
3. Ensure all bytes have been transmitted (e.g. sleep for 16 ms)
4. Set `DTR` Low
5. Loop Forever
   (a) Read data
   (b) Append data to response
   (c) If `DSR` Low
      i. If `RING` High
         A. Error
      ii. Done