Contents

.....
.....

1. Purpose:

This Guide is intended for those with technical knowledge wishing to develop additional hardware, software and firmware for use with CBUS. It also provides all the technical background and information to enable a better understanding of how CBUS works, along with the rationale for our choices and methods.

It is not a ‘User Guide’.

This is a draft version with work still to do in cross references and other aspects.

Content that is new or changed in this revision is marked with a bar to the left of the paragraph(s)

2. Introduction.

2.1. Introduction to CBUS

CBUS is a general purpose layout control system (layout control bus or LCB) for model railways (MR) of any gauge or scale. It also covers control of locomotives and integrated schemes for full layout automation including computer interfacing.

The philosophy of CBUS was simplicity, both of use and implementation, and flexibility. Its main characteristic is the separation of the messages from the transport. It is inherently ‘transport independent’. However, to allow the interchange of different manufacturer’s hardware, a common transport (or hardware level) was needed. Having considered many options we decided on the industry standard CAN bus. Not only has this been in use for MR LCBs for many years by Zimo and so is proven in its suitability for the MR environment, it has a massive hardware and software base in the industrial and automotive area so CAN hardware is readily available from multiple vendor sources and no design effort is needed to implement it for a MR LCB.

There are many references to CAN and no further description is needed here except to point out that it is a multiple access, bi-directional bus which uses its (unique) non-destructive bitwise arbitration scheme which always ensures the immediate transmission of the highest priority message.

The CAN protocol sets the data length per message at 8 bytes. This is a compromise between bytes per message and access time to the bus for other ‘nodes’. CAN is designed for systems requiring short messages but with guaranteed ‘latency’ or access time as it is used in safety critical situations. The ability to prioritise the access also is a great advantage.
Overall, we considered that CAN represented the best available solution for a MR LCB without inventing yet another bus with no hardware support or user base.

1. It is a tried and tested system with readily available hardware at low cost.
2. Minimal effort is required to incorporate it into MR products.
3. It is an open standard with no IP restrictions.
4. The data rates and transmission distances are suitable for a MR LCB. (125 Kbits/sec)

The arbitration scheme of CAN requires that each node has a unique ‘header’ section, in the CBUS case of 11 bits. We have developed a mechanism whereby a node can self-allocate its own unique ID without needing user intervention or the inclusion of a manufacturer programmed unique number.

The scheme allows for peer-to-peer, master-slave or any combination of transaction.

Note 1: Although CAN is the currently implemented transport channel, the protocol allows for any other transport scheme that may be desired including various wireless systems and Ethernet.

Note 2:The CBUS protocol is transport independent. The parts of this document that relate to the transmission of CBUS over CAN do form part of the CBUS specification but are only applicable when CAN is being used for the transport. The specification will be developed to include any alternative requirements for CBUS over other transport layers as and when they are implemented.

Note 3: CBUS was simply a name that originated within the development team. It is not trademarked. Any commercial development may need to use a different name.

Note 4: All CBUS documentation has avoided using abbreviations or acronyms which are used by the NMRA DCC Standards or RPs. We have deliberately avoided the use of 'CVs' to prevent any confusion. Also we do not refer to 'packets' or 'decoders' as this is again DCC parlance.

2.2. Nomenclature:

For the purposes of this document. the following nomenclature will be used.

Module: A hardware implementation containing the processing and input / output components for attachment to layout devices.

Node: A discrete attachment to the bus using a suitable transceiver. Normally, each module would have one node but multiple nodes per module are possible.

Device: A physical layout device such as a turnout mechanism, light, relay or switch etc. Can also be a 'virtual' device on a PC.

Message: A self contained set of bytes conveying information to or from a node. CBUS messages are usually limited to 8 bytes but may be extended to multiples of 8 bytes.

Frame: A complete set of data as used in a CAN transmission (or other transport). For CAN, a frame includes the CAN header bits and the CBUS message.

Event: A generic term derived from the 'Producer / Consumer' model for a message conveying layout control or state information. An event is sent in response to a change in state of a layout device and will result in an action on the layout of other devices.

3. The CBUS message structure (summary)

Each message has between one and eight bytes. The first byte is the ‘command’ byte which also includes information on the number of bytes in the message. This command byte is also referred to as the OpCode or OPC. All messages have one command byte or OpCode.

The remaining part of the message will depend on the purpose of the message. There are several categories of message.

3.1. Module configuration.

CBUS is highly flexible and allows for many different types of module, each with different configuration requirements. A set of OpCodes and associated messages has been defined for this module configuration. Included here is the ability to read back information relating to module status and configuration.

3.2. Layout control.

Layout control has a smaller set of OpCodes defining ‘events’. These are ON events, OFF events and ‘request / response’ events. The nature of these events will depend on the ‘model’ being used, the ‘long’ event or the ‘short’ event. This will be described in section 4.1. A normal event is one OpCode and 4 additional bytes but there is the capability of adding up to three extra data bytes where further information needs to be conveyed, such as ‘analogue’ values.

3.3. Locomotive control.

There is a specified set of OpCodes and messages for locomotive control. Although intended for use with the NMRA DCC standards, the structure would also apply to analogue (DC) controllers if required. Included in this category are commands for track power control.

3.4. Bus control.

A small set of OpCodes / messages for control of the bus.

4. The message format.

The CBUS message format was only arrived at after a number of years of discussion and debate, including the testing of several alternative schemes. Crucial to the requirements were the transport independence, the messages had to be self-contained, and were not to rely on any specific hardware capabilities such as the ‘filtering’ available in CAN. Having tried CAN filtering, we realised that the number of required options far exceeded the capability of the CAN filter or mask mechanism. Hence the ability to distinguish and ‘parse’ messages was best handled entirely in the module firmware. As CAN filtering only takes place (with some options in some processors) in the CAN header section, we opted to avoid any protocol that has any message information in any transport header. Other transports may not have the ability to process header information anyway. A consequence of this, when using CAN, was to allow the use of the ‘standard’ CAN frame with its 11 bit header. (See section 6 for CAN header requirements). The OpCode.

All CBUS messages start with a single byte OpCode. For subsequent parsing, we chose to include in this OpCode, information regarding the number of bytes in a message. The top three bits of the OpCode byte give the number of message bytes, i.e. 000 indicates no message bytes, just the OpCode, and 111 indicates 7 added bytes which is the maximum. The remaining 5 bits of the OpCode define its purpose. Hence there can be 32 messages per message length. However, each block of 32 has an OpCode allowing an extention OpCode byte adding 256 more OPCs per length. This has not proven necessary so far and these extension OPCs have not been defined. The presently defined OpCodes are given in the Specifications section (Section 12) of this document.

There is no ‘error’ byte in a CBUS message. The integrity of the message must be ensured by the transport. With CAN, error detection is inherent in its working.

4.1. Layout control.

The layout control scheme will be described first as it requires some explanation, including its history.

CBUS uses two alternative methods, or ‘models’ for layout control, both of which are fundamentally different in concept. These are the ‘Producer / Consumer’ (P/C) model and the ‘Device Addressed’ model. CBUS initially used just the P / C model but experience on real layouts led us to the alternative Device Addressed scheme.

4.1.1. The Producer / Consumer (P / C) model.

In P / C terminology, a ‘producer’ creates an ‘event’ that is sent onto the bus. Strictly, this is just a unique number. Any ‘consumer’ needing to act on this event is ‘taught’ it so it recognises that event in future.

Different consumers can act on the same event but in different ways if wanted. This is a flexible arrangement where it is easy to add consumers. With true P / C, there would be a separate event for an ON action and an OFF action – two different numbers. CBUS uses the same 4 byte number so that it can be associated with a producer device, like a switch on a control panel, and uses the OpCode to indicate ON or OFF.

The requirement of P / C is that the event should be numerically unique even if the number is arbitrary. To ensure this in CBUS, the producer nodes have a 16 bit Node Number (NN) which forms the first 2 bytes of the unique event number. As long as this NN is unique, it ensures that the whole 4 byte number is unique on the layout, and meets the requirement for the event.

As each producer node may have many input actions, the third and fourth bytes make up a second 16 bit number for actions within a node. These could be the same for all similar nodes and ‘built in’ to the hardware. Hence a CBUS P / C event is a 32 bit (4 byte) number, preceded by an OpCode. The format, in bytes, is

<OPC><NNhi><NNlo><ENhi><ENlo>

Where EN is the value associated with specific input actions. For example, the OPC is 0x90 for an ON event and 0x91 for an OFF event. It is a 4 byte message so the OPC has ‘100’ as the top three bits. The remaining bits are 10000 or 10001. The only other event type used is a ‘request’ which is used to elicit a ‘response’ from another producer giving the state of (say) a producer input without performing a defined action. Here the ‘request OPC is 0x92 and the ‘response’ is 0x93 if the input is ON and 0x94 if it is OFF.

If there is only one producer of a specific event, that producer can be identified by its NN. This is useful for diagnostics as well as ensuring event uniqueness. A ‘one to many’ scenario. It is possible to have many producers sending the same event but you then lose the source NN identification. This represents a ‘many to many’ scenario. The CBUS P/C model focuses on the ‘one to many’ approach.

4.1.2. The ‘Device Addressed’ model

With the P/C model, there is no easily specified relationship between the numerical value of an event and either its source or destination. This is particularly the case with multiple consumers acting on the same event. You cannot always tell what will happen when a given event is sent.

This is a great feature of the P/C model, giving great flexibility as to how it is set up. However, it can also be a problem in certain situations.

Some computer programs struggle with this pure P/C approach, whilst others cope without difficulty.

Some computer programs need to be able to send commands to specific, known layout ‘devices’ and receive state changes from known layout devices. This is, in effect, an ‘addressed’ scheme where each layout device has a known address or, as we have termed it, a ’device number’ or DN.

Thus a switch on a control panel can have its device number and a turnout a different (or even the same) device number. CBUS has the capability to use either the P / C or the device addressed scheme, including both at the same time. We have kept the same message format for both but used different OpCodes to distinguish. The format for an ‘addressed’ event is

<OPC><NNhi><NNlo><DNhi><DNlo>

The node number of the sending node is still included for traceability and diagnostics but is ignored by the receiving nodes or any attached PC software. In this case it is the DN that is unique to the layout device which either produces or consumes the event. Clearly this limits the number of definable devices on a layout to a16 bit range (0 to 65535.)

Once again, there are ON, OFF and request / response OPCs.

4.1.3. Long and short events.

Although it may seem a bit misleading, the P / C events with all 4 bytes making up the unique event number have been termed ‘long’ events and the addressed events where only the lower two bytes are the significant device number, have been termed ‘short’ events, even though they are actually the same length in byte terms.

The long events are best suited to ‘hardware only’ layout control and for use with computer programs that support the P/C model. The short events are best used with PC software that does not support the P/C model.

5. SLiM and FLiM – the two hardware operating modes.

SLiM The Simple Layout implementation Model

FLiM The Full Layout implementation Model

One of the starting criteria for CBUS was the ability to set up and use the system without any need for a PC or any similar configuration device using the CBUS itself. Configuration was to be entirely manual using switches on the modules. This use of CBUS modules is called SLiM..

5.1. Characteristics of SLiM.

SLiM uses the 'long' event scheme. (P / C)
Producers have their Node numbers set by switches. This NN range varies with the physical implementation of the module itself but has a maximum range of 1 to 99.
It is up to the user to ensure that NNs are unique.
The set NN is also the CAN_ID. (See section 6 for CAN header requirements)
Consumers do not have a NN. There is no protocol limit to the number of consumers.
Consumers are taught the event and how to respond to it by use of a 'learn' switch and then sending the event from the producer.
Learned events can be unlearned and all events cleared.
Full route setting and state polling is possible in SLiM.

It is possible to use a PC for layout control with SLiM. The computer must be able to 'learn' the producer events (act as a consumer) and then, after processing these events, act as a producer to send the commands to the layout modules.

5.2. Characteristics of FLiM

The FLiM scheme relies on the use of a PC or similar configuration tool for setting up and teaching the modules. It has more comprehensive capabilities than SLiM.
It can operate in either the P / C (long event) mode, the 'addressed' or short event mode or any combination of both.
Although initial configuration requires a PC tool, subsequent operation doesn't need a PC to be involved. (CBUS has no 'node manager' requirement)
There is a comprehensive set of OpCodes and instruction messages for configuring FLiM modules including the ability to read back all settings to a file for subsequent reinstatement if required. This will be covered in section 7.

As all nodes need to accessed by the configuration tool, both producers and consumers must have node numbers. These are a 16 bit range (1 to 65535) and would normally be allocated by the configuration software. Nodes (modules) can be given names linked to their NNs.

NN of 00 00 is a special case reserved for SLIiM consumer nodes and configuration software.

6. Requirements for use with CAN.

Although CBUS is transport independent in its protocol, the present implementation uses CAN. This section describes the requirements for the CAN header, including the allocation of unique CAN_ID values and message priority.

6.1. The Standard CAN frame.

CBUS uses the standard CAN frame for all its LCB functions. (The exception is the bootloader which is described in section 11). The standard CAN header is 11 bits long. CBUS uses the top 4 bits for priority and the remaining 7 bits for the CAN_ID number. This decision was made simply because readily available CAN transceiver ICs only support up to 110 nodes on a CAN 'segment'. Hence we opted for a 7 bit range for the CAN_IDs. CAN uses a bitwise arbitration scheme whereby the header with the lowest value has priority if multiple nodes attempt bus access at the same time. Now the message with the lowest number (highest priority) will always gain immediate access to the bus and the remaining messages will automatically retry and be sent in priority sequence. This priority scheme is utilised by CBUS as follows.

Assuming the header bits are 0 (LSbit) to 10 (MSbit):

Bits 10 and 9 are called the Major Priority (MjPri.) There are three possible values
00 is the highest
01 is next
10 is lowest.

The CAN protocol prohibits a sequence of 7 or more 1 bits at the start of the header, so a MjPri. of 11 is not used. The MERG CBUS modules have a dynamic priority scheme where messages start with the lowest value (10) and will increase the priority to maximum if a message is not sent within a given time or a preset number of 'retries'. However, this particular 'latency' scheme is only a specific implementation. It does ensure no message waits too long. However, when using CAN, implementors should adhere to the priority values as indicated in the Specification section, at least for starting priority, to avoid possible 'bus hogging'.

Bits 8 and 7 are the Minor Priority (MinPri.) bits and are allocated to messages based on their urgency requirements. This is somewhat arbitrary but loco control (speed and direction messages or emergency stop for example) have a higher priority than layout control like turnout changing. These values should be adhered to for compatibility.

6.2. The CAN_ID

The remaining 7 bits of the header comprise the CAN_ID number. To avoid bus conflicts, each node requires a unique CAN_ID. With modules in SLiM mode, the CAN_ID value is set by the on-board switches. A NN of 00 00 is reserved so the firmware of a SLiM node should ensure that the CAN_ID is in the range of 1 to 99. CAN_ID values in the range 100 to 127 have been reserved (so far) for modules with fixed CAN_IDs such as the PC interfaces and DCC command station which have no DIL switches.

See Appendix A for a list of fixed CAN_IDs.

6.3. The self enumeration scheme.

MERG CBUS modules in FLiM mode and the DCC CABs implement a self enumeration scheme whereby any new module can allocate itself a unique CAN_ID. The mechanism is as follows. (An understanding of CAN is assumed here).

A node which is put from SLiM to FLiM for the first time, or a CAB when a loco is requested will issue a Remote Transfer Request (RTR) CAN frame. This prompts all other nodes currently active on the bus to send their CAN_ID values. This value is in their CAN header so there is no data byte with this frame.

The new node monitors all the incoming zero data frames and notes their CAN_ID values. After a delay (presently set to 100mSec), the new node chooses the lowest unused value. If a fixed node, it keeps this value. If a CAB it releases it if unplugged and will re-enumerate when next plugged in.

Note 1. The sequence in which the CAN_ID responses from other nodes arrive is not important.

Note 2. The scheme only works for existing nodes that already have a NN and are powered up. If no nodes have a NN, the first one to make a request will give itself the lowest value so only one node should perform the self enumeration at one time.

This means that self enumeration should not be performed by nodes at power up, as all nodes would be trying to self-enumerate at the same time.

The CAN_ID, once set by self enumeration, should be stored and kept static until changed. If a node is moved from one layout to another, or a module introduced back into a layout where other modules have since been added, it is possible that a CAN_ID conflict could occur. Developers should therefore provide a mechanism whereby self enumeration can be invoked again. The existing CBUS modules do this when first set into FLiM mode from the default SLiM mode and when the pushbutton is pressed briefly again whilst in FliM.

An alternative mechanism is for the developer to implement automatic CAN_ID conflict resolution.

The module checks all incoming packets and if it receives a packet with its own CAN_ID, it initiates self enumeration as described above, to get itself a new CAN_ID. Thus all conflicts are resolved automatically, and the user need not be concerned with CAN_IDs and CAN headers at all.

At the time of writing, only universal CANMIO, CANCOMPUTE and CANPanel implement automatic CAN_ID conflict resolution.

When transferring a module from one layout to another, if it is not known that it supports automatic CAN_ID conflict resolution, the user should ensure that it performs self enumeraton to get a new CANID on the destination layout.

Wth most modules, this an be done by a short press on the pushbutton. If this feature is not available, the module should first be released from the existing layout by setting back to SLiM. When powered up on the new layout, setting to FLiM will introduce the new module to that layout and perform the required self-enumeration. See section 7 for module configuration.

For most users, the exact values of CANID should not matter and should be considered “below the hood”. However,, some users may prefer to direcdtly manage the CAN_ID themselves.

There are 2 opcodes to facilitate this,ENUM (0x5D) allows a software tool to force a self-enumeration sequence for any node at any time and OpCode CANID (0x75) allows a software tool to set a user specified CAN_ID. However, at the time of writing, the firmware for the majority of modules in use has not yet implemented these opcodes.

The use of CAN as transport for CBUS is not mandatory but for interchangeability of existing CAN based hardware, the scheme should be adhered to where CAN is used.

6.4. The CAN header format:

For subsequent documentation purposes, including the main Specification document, the following nomenclature for the CAN header will be used.

[<MjPri><MinPri><ID>]

When implementing the header in microcontrollers like the PIC 18Fxx8x series, the standard header is in two bytes. – SIDH and SIDL. The 11 actual bits are ‘left justified’ and the remaining 5 bits of SIDL must be set to zero. As extended frames are not used, the EIDH and EIDL bytes should also be set to zero. As an example, if the CAN_ID is 99, the MjPri = 10 and the MinPri = 11 then the two header bytes SIDH and SIDL are

1011 1100 0110 0000 or 0xBC, 0x60

Also, if using the MERG CAN_USB or CAN_RS PC interfaces, the CAN header (two bytes) must be provided by the software.

7. Configuring the modules.

The following section covers the basic principles of module configuration. This can be a relatively complex process and some specifics will be dependent on the properties of the actual module being configured. Each of the present MERG CBUS modules will be covered separately in the 'MERG CBUS Modules' document.

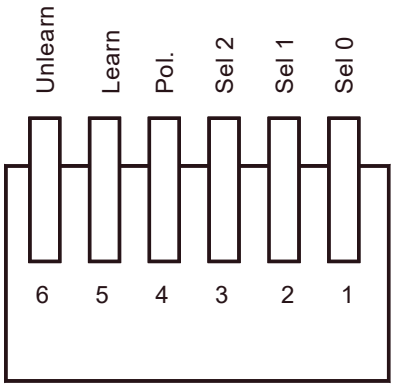
Although this document is primarily orientated towards the FLiM scheme, the SLiM configuration will be briefly covered here. It applies only to the "classic" MERG modules. Other designs may employ a different system for manual configuration provided it is suitably documented.

Note that CAN_RS and CAN_USB modules are neither SLiM nor FLiM, the green LED only indicates that the module is operational.

7.1. SLiM configuration.

SLiM mode is indicated by the green LED (only) being illuminated. It is the default state for all new modules.

Each module has a DIL switch and on some modules, additional jumpers. For producer only modules, it is only necessary to set their switches or jumpers so each producer module has a different value (bit combination). The value will be that set for the NN with one added. Hence a bit switch value of 000 will be translated into a NN of 1 and a bit switch value of 111 into a NN of 8. This is to avoid a NN value of 0. When an input on a producer module changes, a long event (ON or OFF) will be sent. The lower two bytes (the EN number) are fixed by the hardware. (The CANACE3 is an exception here as it can be taught what event to send for a given switch change but this is an unusual situation if, say, you require two or more control panels to send the same long event.) Consumer modules require teaching the events to respond to. They have a DIL switch (6 way) and, in some modules, additional jumpers. Typical of an 8 output module such as the CANACC8 would be as shown below.



DIL switch Top view

The three 'Sel' switches select which output the 'event' will apply to. There are 8 outputs numbered 1 to 8. The three switches allow a selection of one of the 8 outputs using a binary sequence. When the switch is 'down' (ON as written on the switch) this represents a logic 0. A switch in the up (OFF) position is a logic 1. With all three switches down, this gives a value of binary 000 and selects output 1

To train the CANACC8 module, you need a CBUS 'producer' module which creates events.
Select the output to which the event will apply. Put the 'learn' switch down. Send the event. Put the learn switch up. The event is now remembered. You can add more outputs to an event by repeating the above sequence with different settings of the Sel switches. Thus, an 8 output module can create any binary number output with a single event. This is useful for signal aspects or driving a 7 segment display. The MERG modules also have a polarity (Pol) switch which, if set during learning, reverses the action of the output relative to the command (ON or OFF). With the present modules, you can add outputs to an event and change the polarity of an existing output but you cannot remove an output once set.

You can remove the complete event by setting both learn and unlearn switches down and then sending the event. (Put them back afterwards).

To completely clear all events from a SLiM consumer, power it down, put the unlearn switch down and apply power. Return the unlearn switch to its OFF (up) position.

Most existing MERG modules will store 32 or 128 events depending on the firmware revision. (the CANLED will store 255 events) If the limit is reached the yellow LED will flash.

For developers designing new hardware, any suitable teaching method can be employed. The only requirements are the ability to set a NN for producers in the range 1 to 99 and for consumers to learn 'long' events, including what to do with them. The MERG scheme is only illustrative.

7.2. FLiM configuration.

This is the major part of this Developer's Guide. CBUS is a very flexible protocol and allows modules to work in many different ways with a significant number of OpCodes and corresponding messages being devoted to module configuration. Frequent reference will be made to these OpCodes and messages which are more formally described in the accompanying Specification section (12.3) to which reference should be made when required. Also each of the existing modules will have different configuration requirements and again, these are spelt out in the associated detailed setting up for specific modules.

The following section is a generic description of the FLiM setup mechanism along with the various facilities for checking / verification and configuration retrieval. It assumes that the module(s) will be connected to a PC or similar tool via a PC interface such as the MERG CAN_USB or CAN_RS modules and be powered up.

7.2.1. Putting into FLiM mode.

Most existing modules have a default mode of SLiM. When first powered up, the green LED will be on. To put a module into FLiM, hold the small pushbutton on the module in until the green LED extinguishes. This is about 8 seconds – to avoid accidental presses. Release the pushbutton and the yellow LED will flash. The module is now in 'setup' mode and will automatically perform the 'self enumeration' process as well as prompting for a node number. The enumeration will occur even if it already had a CAN_ID previously. It will probably allocate a different CAN_ID as a result. Note: this arrangement is that used by the MERG modules. Other implementations that achieve the same objective are possible.

7.2.2. Allocating a node number.

The NN prompt is RQNN (request node number). In a 'virgin' node the accompanying NN will be 00 00 but if the node already has a NN, this will be included instead. The message is

<0x50><NNhi><NNlo>

A software tool should respond to this message. You can read the node parameters at this stage as a block and then allocate a node number or simply allocate a node number.

Note: Certain node numbers are presently allocated to specific devices and should not be taught to any modules These are listed in Appendix A:

A software tool should not restrict the range of node numbers available within 1 to 65535, and should simply warn if users try to allocate a node number in the fixed node number range. This is because some modules, such as CANCMD, permit the allocation of a new node number if the user wishes. It is important that the software tool does not stop the user from putting it back to the fixed node number if required.

7.2.3. Node parameters.

It is useful for the software tool to know what the new module is so it can allocate the appropriate node number and identify any files or information needed for the remaining configuration process. Modules should contain a 'parameter set' giving the information as listed below. Parameters 1 to 7 can be read in setup mode using RQNP. This is a single OPC message as the node does not, at this stage have a node number. It is important that there is only one node on the system in 'setup' mode at any one time. While the yellow LED is flashing, send

<0x10> (RQNP)

The module will respond with PARAMS. This is the OPC followed by parameter bytes 1 to 7..

<0xEF><PARA 1><PARA 2><PARA 3> <PARA 4><PARA 5><PARA 6><PARA 7>

The full set of parameters are defined as:

Para 0	Returns the number of parameters defined (not including this one)
Para 1	The manufacturer ID as a HEX numeric (If the manufacturer has a NMRA number this can be used)
Para 2	Minor code version as an alphabetic character (ASCII)
Para 3	Manufacturer's module identifier as a HEX numeric
Para 4	Number of supported events as a HEX numeric
Para 5	Number of Event Variables per event as a HEX numeric
Para 6	Number of supported Node Variables as a HEX numeric
Para 7	Major version as a HEX numeric. (can be 0 if no major version allocated)
Para 8	Node Flags
Para 9	CPU type the firmware is built for – see the CBUS header files for a full list of codes
Para 10	Transport type eg: CAN, Ethernet, MiWi etc. See codes below.
Para 11	Code load address (4 bytes to allow for future use of 32 bit processors)
Para 15	CPU manufacturers code as read directly from the chip by the firmware (4 bytes)
Para 19	Code for CPU manufacturer
Para 20	Beta release build number – set to zero for release version

Parameter 8, the Node Flags byte contains bit flags defined as:

Bit 0: Consumer
Bit 1: Producer
Bit 2: FLiM Mode
Bit 3: The module supports bootloading
Bit 4: The module can consume its own produced events
Bit 5: Module is in learn mode

If a module is both a producer and a consumer then it is referred to as a "combi" node and both bits 0 and 1 will be set.

It has been proposed that a module should set bit 5 of parameter 8 when it is in learn mode. This enables a software tool to check that only one module is in learn mode. See section 7.2.8.4.

Parameter 10, the hardware transport type, contains a code as follows:

1 – CAN
2 – Ethernet
3 – MiWi (Microchip mesh network)

Parameters 11-14, Firmware load address:

This is a 4 byte address that can be used by a bootloader to load code into Flash.

A 4 byte field has been allocated to allow for the use of 32 bit processors.

Note that although parameters above 7 cannot be read using RQNP whilst in setup mode, any parameter may be read once a node number has been assigned using RQNPN, the module responds with PARAN.

CBUS also allows for a module to have a manufacturer defined ‘name’ which can be read at this stage if one has been allocated.

This is requested by RQMN.

<0x11>

The response is NAME

<0xE2><><char1><char2><char3><char4><char5><char6><char7>

The NAME string is limited to 7 chars, all 7 characters are used, the name string must be space filled. If the module uses CAN as its transmission method, the NAME will have ‘CAN’ added as the first part of a name so the actual module name could be 10 characters including the assumed CAN The actual name prefix is must be added by a configuration software tool and depends on the Interface Protocol parameter defined in section 7.2.3.

Reading the parameter set and a name (if any) is a useful but optional process. However, to exit the ‘setup’ mode, the node must be allocated a node number using SNN (set node number)

<0x42><NNhi><NNlo>

It is up to the software tool or user to allocate a node number which should be unique to that node. Only where duplicate nodes such as for ‘replacements’ should a NN be repeated. The software can attach a user defined descriptor to the NN so any subsequent reference can be to a recognisable name.

The module will reply with NNACK (node number acknowledge) as

<0x52><NNhi><NNlo>

The module will now be in FLiM mode and the yellow LED will go steady. It will retain its FLiM status and NN (including its CAN_ID) during any power down.

The module is now ready for teaching. If no NN is allocated, the yellow LED will remain flashing. It can be cancelled back to SLiM by holding in the button again for about 8 seconds.

For compatibility and uniformity, it is recommended that the above sequence and LED colours are adhered to in any module design.

7.2.4. Switching back from FLiM to SLiM .

Once in FLiM mode, if the pushbutton is held down again (about 8 seconds), the yellow LED goes out and the module reverts to SLiM mode with the green LED on. If it is a producer or combi node, then the NN should revert to that set by the DIP switches. If it is a consumer, then it reverts to having a zero NN. The node should issue a NNREL opcode for the NN being released.

Note that switching between FLiM and SLiM does not remove taught events.

7.2.5. Changing NN and/or forcing CAN ID self enumeration.

If the pushbutton of a FLiM module is pressed briefly, the module re-enters setup mode as described from section 7.2.1.

The RQNN sent from the module as it enters setup mode will include the current NN. The PC configuration tool may prompt the user as to whether they wish to keep this NN or allocate a new one. It can then send the same or a new NN using the SNN opcode.

If the pushbutton is pressed briefly again without a new NN being received, the module continues in FLiM with the previously allocated NN. This enables you to check the NN of any module.

A module should carry out a self enumeration to fetch a new CAN ID when going into setup mode.

With the present implementation, most modules will carry out a self enumeration in this way. Some earlier firmware may not do this, and such a module will retain its CAN_ID even if allocated a new NN. To force self-enumeration of the CAN_ID, the module must first be returned to SLiM mode.

7.2.6. Teaching Node Variables. (NVs)

Node variables are parameters (in bytes) which affect the operation of the whole node or module. They are taught by reference to the node number (NN) and the node variable index (NV#). The number of NVs depends on the type and function of the module. Each NV has a single teaching command. (NVSET). The node variable index starts at a value of 1. The command format is:

<0x96><NNhi><NNlo><NV#><NVval>

where NVval is the actual byte value of the node variable.

Repeat the command for each node variable required. Node variables are stored in non-volatile memory and the time required to write to memory may exceed the CAN frame repetition rate. As a result, successive writes may get missed so the CBUS module will confirm a write operation with a WRACK (write acknowledge).

<0x59><NNhi><NNlo>

If you try to write more NVs than the module can accept, without waiting for WRACK, instead of a WRACK you will get an error message. (CMDERR), or if the module is busy writing to its non-volatile memory, it may miss the subsequent NVSET message altogether.

<0x6F><NN hi><NN lo><Error number>

The error number for too many NVs is 10 (decimal) or ‘Invalid Node Variable Index’. The error messages are in section 12.4 f the specification section. Note that the number of allowed NVs is contained in the module parameter set (Parameter 6) so any software tool should be aware of the NV# limit.

7.2.7. Readback of node variables.

You can retrieve stored node variables form a module with NVRD

<0x71><NN hi><NN lo><NV#>

NV# is again the index of the NV required. The response is NVANS.

<0x97><NN hi><NN lo><NV# ><NV val>

Repeat for all NV index values.

A module does not need to be in learn mode to read and write Nvs..

7.2.8. Teaching events and event variables.

Unlike Node Variables, the teaching of events and their associated variables is performed in two stages. This is determined by the 8 byte limit of a CAN frame (and the defined size of a CBUS message). As previously described, the CBUS event comprises 4 bytes. However, in FLiM mode, the module also needs to be told what it is to do with an event. With SLiM setup, this is set by using the switches when teaching. For FLiM teaching, the information must accompany the events.

As with the node variables, each taught event can have a number of Event Variables (EVs) as determined by the functionality of the module. The Evs determine what action the module takes on receiving this event (for consumed events) or what happening in the module causes this event to be sent (for produced events).

Due to the message size limitation, the EVs are taught one at a time using an indexing scheme.

7.2.8.1 Event Teaching Process

The event teaching process is as follows:

a. Put the module into its ‘learn’ mode. (NNLRN)

<0x53><NNhi><NNlo>

b. Send the events to be taught (EVLRN)

<0xD2><NN hi><NN lo><EN hi><EN lo> <EV#><EV val>

Where the <NN hi><NN lo><EN hi><EN lo> are the node and event values of the event to be taught. For teaching device numbers using device addressing, NN hi and NN lo must be set to 00 00. EN hi and EN lo now become the DN hi and DN lo. e.g.

<0xD2><00 >< 00><DN hi><DN lo> <EV#><EV val>

As for the NVs, the EVs are indexed starting at 1. Different modules will have different numbers of EVs and their EV values will also be module specific.

A separate EVLRN message must be sent for each EV.

No assumption should be made as to how events are retained in non-volatile storage. Events may not be stored sequentially in the module if a hash table is used. The number of possible stored events and the number of EVs per event can be read from the parameter list. See sections 7.2.3 and 7.2.8.2. If an attempt is made to store too many events or the EV# is out of range, an error message (CMDERR) will be returned. See section 12.4.

As event storage will be to non-volatile memory, each EVLRN message is acknowledged with a WRACK – or CMDERR. Do not attempt ot write the next EVLRN until the WRACK has been received from the previous one.

c. Take module out of learn mode. (NNULN)

<0x54><NN hi><NN lo>

The above teaching process can be performed at any time while the bus is in use by other activities. You can change individual event variables at any time by repeating the learn sequence using the same first 4 bytes to define the event.

To remove an event while in learn mode, use EVULN.

<0x95><NN hi><NN lo><EN hi><EN lo>

7.2.8.2 EVs per event

Most modules have a fixed number of Evs per event. This is returned In parameter 5. If the software tool does not teach all the EVs for a given event, the remaining Evs should have a default value, defined by the module in question.

Some newer modules may require a lot of EVs for some events, and may implement this by only storing the number of EVs it needs for a given event, thus saving storage space in its non-volatile memory.

A module that implements a variable number of EVs per event will define the maximum number of EVs per event in parameter 5.

The software tool can then just teach the number of EVs required for a given event. For example, with CANPanel, only the EVs for the number of LEDs to be affected need be taught.

The number of EVs actually in use for a given event can be determined by reading EV0 for that event. Note that modules that do not implement a variable number of EVs per event probably will not implement EV0.

If an attempt is made to read an EV that is not actually in use for that event, the module will return a CMDERR message with an error code of 5 (NO_EV). This is distinct from error code 6 (INV_EV_IDX) which means that the EV is beyond the maximum number of EVs per event.

Where a module supports the teaching of both consumer and/or producer events, these are both taught by the same mechanism, and the contents of the EVs will define whether a particular event is consumed, produced or both.

7.2.8.3 Consuming own events

A message sent by CAN cannot be seen as a received message by the transmitting module. For this reason, many modules cannot be taught to consume their own produced events. For example, you would not be able to have a switch connected to a module produce an event that can be taught to operate an output on the same module, using the usual teaching mechanism.

However, some firmware gets around this by buffering all transmitted events as though they had also been received. This allows it to consume its events which can be taught in the usual way.

A module that can consume its own events should set the “COE” flag, bit 5 of parameter 8. A software tool should permit teaching a module its own events when this parameter flag is set.

7.2.8.4 Learn Mode

Important. Only one node must be in learn mode at any one time or more than one node will be taught the same events.

For resilience, if a node receives a NNLRN or NNULN for another node whilst it is in learn node, it MUST exit learn mode immediately, to avoid it learning spurious events.

If the configuration utility recieves a WRACK from the wrong node number whilst teaching events, it should issue NNULN to both the correct and eroneous nodes and restart the teaching process. Ideally, I should be assumed that the node that issued the eroneous WRACK now has an incorrect event taught, which should be checked.

Although these situations should not arise, experience has shown that it can happen due to a bug in the module firmware or PC conifguration utiltiy, or due to a module missing its NNULN command.

It has also been proposed that a module should set bit 5 of parameter byte 5 (flags) when in learn mode. This allows a software tool to check that only one module is in learn mode by reading this parameter or issuing a QNN.

The flags byte is also returned in the PNN response from each module, see section 7.2.9.

Other facilities for module configuration.

Apart from the basic teaching process above, there are other OPCs related to module configuration.

7.2.9. Query all nodes.

It is possible to rebuild a FLiM configuration from scratch. The first step is to query all the nodes to find out what is connected. This is done using QNN:

<0x0D>

Every connected node, that has a node number, should respond with a PNN message:

<0xB6><<NN Hi><NN Lo><Manuf Id><Module Id><Flags>

The manufacturer id, module id and flag byte are as defined for the module parameters, see section 7.2.3.

This will result in a stream of PNN (0xB6) packets, one from each node. The requesting node must be designed so that it can cope with this stream without packet loss.

Once a list of all nodes has been built, further information can be obtained from the nodes by reading Parameters, Node Variables, Events and Event variables as described in sections 7.2.6 and 7.2.10.

The rebuilt configuration will include SLiM producers but will not include any SLiM consumers, or events they have been taught, because without a node number a SLiM consumer cannot respond to the QNN message.

Of course, whilst this rebuilt configuration contains all the information about node settings, events taught and event settings, it cannot contain any information about how the various events are used or what physical devices (turnouts, signals etc) they refer to, as this will be specific to the layout to which the modules are connected. It is not possible to recover a module name if it had one.

Note also that the QNN query and PNN response was introduced to the specification at version 7h. Modules that contain older firmware written to an earlier version of the specification may not respond to the QNN message.

7.2.10. Readback of events and EVs.

You can read back all stored events in a module using the NERD or NENRD OPCs. For NERD use:

<0x57><NN hi><NN lo>

This will cause the module to send a sequence of messages giving all the stored events and their index number in the events table. (ENRSP)

<0xF2><NN hi><NN lo><EN3><EN2><EN1><EN0><EN#>

Here, the NN hi and NNlo are the node number of the module being polled. EN0 to EN3 are the four bytes of the stored event (EN2 and EN3 will be zero for short events). EN# is the index of the event in the module's event table. This can be used subsequently for reference to individual events.
Note that the value of EN# for any particular event should not be retained by a software tool. The value of EN# for an event can change if a new event is added or an existing event is deleted.

If you know this index and want to retrieve an event from a module individually, you can use NENRD.

<0x72><NN hi><NN lo><EN#>

This will produce the ENRSP message but only for the one event.

If you know the event index, it is possible to change the whole event rather than just the event variables as with EVLRN. This uses the EVLRNI OpCode (Event learn by index). Use the same sequence as for EVLRN but with the following format:

<0xF5><NN hi><NN lo><EN hi><EN lo><EN#><EV#><EV val>

Now the event location is pointed to by EN#. This could be useful for changing the device numbers which had been previously allocated without deleting the event first.

Other useful OpCodes for configuration purposes are:

NNCLR Clear all events from a node. Must be in learn mode first for safety reasons.

NNEVN Allows a read of available event space left. Answer is EVLNF

RQEVN Read number of stored events. Answer is NUMEV

8. Layout control and associated commands.

This section may be of assistance for those developing layout control software.

CBUS is essentially an ’event driven’ scheme. A producer such as the software creates a CBUS message or event which is recognised and acted upon by one or more consumers. This applies equally to either the long or short schemes. In the latter case, the consumer would be a numbered device. Equally, layout based devices create an event when a change occurs. These are flagged by their OpCodes as either ON or OFF events. All events are essentially ’broadcast’ by nature and it is up to the user software or module firmware to determine whether to act on that event or not.

As previously described, a CBUS event comprises an OpCode and 4 subsequent bytes. The OpCode determines the nature of the event. However, CBUS allows for extensions of this rule.

8.1. Events with added data.

The basic CBUS message has a limit of 7 bytes plus the OpCode. While still maintaining the original ON / OFF style, different OpCodes define events with one, two or three added bytes which can contain data. These bytes may carry digitised ’analog’ data such as track current, turntable position, speed settings etc. so a consumer may use the data to determine why the event has been created, e.g track current is too high. When sent by the producer, the data may be used to set speeds, positions, lamp colour / brightness or anything else in the consumers.

8.2. Polling of producers.

Although event driven, CBUS has OpCodes that allow a device such as a computer to determine the state of any layout device or input on request without a change having taken place. The format is the same as for an ON or OFF event but uses a different OpCode. The response to such a ’request’ also has the same format but different OpCodes so a response can be differentiated from an actual change event. This process applies to both short and long events and these again are distinguished by different OpCodes.

8.3. Data only events.

There may be situations where more data bytes are required than the allowed extra three (see events with added data). If only the node number or device number is utilised, (two bytes), then 5 data bytes can be included in a message. This particular situation arose with the use of RFID tags for locos and rolling stock. CBUS has four OpCodes to define these transactions.

a. Accessory node data event (ACDAT)

<0xF6><NN hi><NN lo><data1><data2><data3><data4><data5>

Produced by a node when the data has changed or at any preset interval.

b. Accessory node data Response (ARDAT)

<0xF7><NN hi><NN lo><data1><data2><data3><data4><data5>

Produced by a node in response to a request RQDAT.

c. Device data event (short mode) (DDES)

<0xFA><DN hi><DN lo><data1><data2><data3><data4><data5>

Produced by a device when the data has changed or at any preset interval.

d. Device data response (short mode) (DDRS)

<0xFB><DN hi><DN lo><data1><data2><data3><data4><data5>

Produced by a device in response to a request RQDDS.

The use of the short mode (device addressing) here allows for multiple RFID readers to be connected to a single node. Each reader has its own device number. A computer can read these data sets at any time using the request OpCode.

8.4. Layout control with ‘short’ events.

As described in section 4.1.2, there may be situations where it is more convenient to describe layout devices using a specified number (Device Number) for each layout ‘device’ which may be a single element such as a turnout or signal or a complete route.

Here, the consumers and producers can be allocated their own DNs. A module with multiple inputs or outputs will have a DN per input or output and hence can be addressed directly. If several consumers have outputs allocated the same DN, then all will respond to a command to that DN as would be required for route setting etc.

8.4.1. Teaching Device Numbers.

Device numbers are taught in the same way as for ‘long’ events (see section 7.2.8). However, the first two bytes of the ‘event’ must be set to 00, 00. The second two bytes are the DN.

As the allocation of a DN is a user choice, it is preferable to use a ‘configuration utility’ for teaching short events. Now the upper two bytes can be set to 00, 00 and the DN chosen appropriately. Given the DN range of 0 to 65535, it may be of value to segment the DN ranges either on the basis of device type or on a layout module basis.

Producer modules can also have their inputs or switches allocated specific DNs in the same way. Hence an event generated by one of these modules can be identified with a specific input or switch. This will be required if a PC is interposed between the producers and consumers (the conventional method for PC based layout control) so the PC will know where the event came from. If no PC is used, the switch, on a control panel say, will be given the same DN as the device it is actuating (e.g. a turnout) giving a ‘one to one’ control or if several consumer devices have that number, the switch can set a route.

9. The DCC system.

CBUS incorporates a set of OpCodes for messages relating to loco control. Although primarily intended for implementation of DCC systems compliant with the NMRA Standards and RPs, it does not preclude use with analog or DC loco control systems given a suitable ‘command station’. The controlling devices (CABs) or computer control along with a suitable command station, communicate via the CBUS wiring. CABs can also send layout control messages if required directly to the accessory modules as the bus is common.

9.1. CABs

The CABs plug into the CBUS and appear as a node on the bus. However, CABs are considered generic in that they are all identical in functionality and are not programmable in the way the layout nodes are. CABs have been given a fixed node number of 0xFFFF for the purposes of ‘bootloading’ and reading properties. (See section 11on bootloading for details). This does not preclude CABs being allocated different node numbers if implemented differently. Except when controlling layout modules directly, CABs require a matching Command Station or a computer emulating a Command Station also connected to the CBUS. Loco control interactions take place between the CABs and the Command Station using a dedicated set of OpCodes.

Cabs may also issue CBUS short or long events for layout control, using whatever scheme the cab designer wishes.

9.1.1. Operating sequence.

When first plugged in or powered up, a CAB is in its ‘reset’ state. It can receive CBUS messages but does not send DCC related messages. It may send layout control messages to devices (short events) if equipped to do so. Hence any CAB can activate the same layout ‘device’. See section 9.5 for more details.

9.1.2. Allocation of a session.

When operating a loco, the CAB is allocated a ‘session’ number by the command station. In practice, it is a pointer to a ‘slot’ in the command station refresh table or ‘stack’. This is a single byte so limiting the number of possible active CABs (or slots) to 256. If a CAB can control multiple locos, each loco is allocated a ‘session’. To activate a session, the user needs to enter the loco address and send the following message (RLOC) to the command station.

<0x40><AAAAAAAA><AAAAAAAA>

The two bytes are the address of the loco requested. The address format is that used by the NMRA DCC scheme. If it is a ‘short’ address, the upper byte is all zeros as is bit 7 of the lower byte. Thus short addresses of 0 to 127 are allowed. An address of 0 should only be used if the command station allows for decoderless locos to be run on DC. For long addresses, bits 6 and 7 of the upper byte should be set by the CAB. This format allows for long addresses in the range 0000 to 3FFF although the NMRA specification only allows a range of 27FF (10239).

The number of active sessions will depend on the command station design and is really limited by the refresh rate required for the DCC packets to the track. Provided a session or slot is available, then command station will reply with PLOC.

<0xE1><Session><AddrH><AddrL><Speed/Dir><Fn1><Fn2><Fn3>

This comprises the full information of the ‘slot’ but tells the CAB its allocated session number.

For a new allocation, the speed / dir byte will be 0. If a moving train is being re-acquired, then the speed/dir byte will indicate the current speed and direction of the train."

If the command station remembers the function settings of previously controlled locos, these will be passed back to the cab in this message, otherwise they will be set to zero. The direction of the stationary loco can also be included, as this may affect which lights are illuminated.

If there are no available slots, the command station will give an error message ERR. Loco stack full.

<0x63><AddrH><AddrL><1>

If the loco with that address is already allocated to another session, either on a different CAB or on the same CAB if multiple locos are allowed, it will send the error message ERR. Loco address taken.

<0x63><AddrH><AddrL><2>

To release a loco from a session use KLOC.

<0x21><Session>

This clears the slot and makes it available for other locos. If a loco is released whilst moving, this is referred to as "Dispatched". The train keeps moving and continues to be refreshed by the command station. It can then be re-acquired by a cab by issuing RLOC as described above.

A cab must monitor for ERR messages and cancel the current session if it receives an ERR message on its current session of either “No session” or “Session cancelled”.

If the cab receives a “loco taken” error message, it may then “Steal” or “Share” the session.

To do this, the cab issues a GLOC (Get Loco) message to the command station:

<0x61><AddrH><AddrL><Flags>

The flags byte is defined as:

Bit 0: Set for "Steal" mode - The cab wants to steal the session from the cab(s) that currently own it, so their sessions will be canceled

Bit 1: Set for "Share" mode - The cab wants to share the session with the cab(s) that currently own it.

Both bits set to 0 is exactly equivalent to an RLOC request
Both bits set to 1 is invalid, because the 2 modes are mutually exclusive

Possible returned packets from the command station are as follows:

PLOC - the request was successful, the PLOC packet contains the session number, speed/direction and function settings as described above in the response to RLOC.

ERR with error code 2 (loco taken) - Requested operation (Steal or share) is disabled or not supported

ERR with error code 3 (no session) - There is no existing session on that loco to steal or share

ERR with error code 7 (invalid request) – There is an invalid combination of flags, i.e: both steal and share are set.

When the command station accepts a steal request, it will first issue an ERR message on the session in question with the error code 8 "Session Cancelled".

This indicates that the current session is being stolen. Cab(s) with that session must cancel the session. A suitable message can be displayed to the user. The cab doing the stealing should ignore this error code whilst waiting for a PLOC.

GLOC was introduced with version 8a of the CBUS specification. Cab and command station designers should bear in mind that their design should also work with older Cabs/command stations that do not implement GLOC to ensure backwards compatibility.

Any cab implementing the new OpCode GLOC must have a timeout in case no response is received.

This is so that if such a cab is used with command station firmware (that does not yet support GLOC), then it won't all lock up if you try to use Steal or Share. If a GLOC request times out, a suitable message can be displayed to the user.

Any command station implementing GLOC must continue to support RLOC as well.

Sharing sessions implies a need for each cab to monitor activity from the other so that it is aware of current speed/direction and function settings for the shared session. A mechanism for speed matching between the cabs that are sharing a session may also be required. All of the information required for this is available in the CBUS messages, but it is left to the cab designer to implement suitable mechanisms on the cab user interface.

9.1.3. Allocate loco to activity

A locomotive may be allocated to a specific activity using the ALOC OpCode.

<0x43><Session><AllocCode>

The meaning of the allocation code is dependent on the application. For example, in the CANCMD, it is used to allocate a loco to an automatic shuttle, and the allocation code is the shuttle number.

If the application requires that the loco is released or dispatched after the allocation, then a separate KLOC message must be sent.

9.1.4. Setting speed step range.

Provided the session is granted, (PLOC) then the CAB should send a message giving the speed step range required for that loco. This is done with a STMOD.

<0x44><Session><MMMMMMMM>

Only the last two bits of the data byte are used. See specification section 12.3for values. It is recommended that the command station defaults to 128 speed steps. Unless the loco requires a different range, the STMOD message may be omitted.

9.1.5. Keepalive.

Once a session has been granted, the CAB must send regular ‘keepalive’ messages DKEEP. This enables a command station to recognize if a CAB has been disconnected.

The minimum keep alive period is defined by the command station, and could be configurable The current cab design sends a keepalive at least once every 4 seconds.

<0x23><Session>

9.1.6. Speed and direction.

Once a CAB has its session, it can send speed / direction and function messages to the command station. The speed / dir is a byte where bits 0 – 6 are the speed and bit 7 is the direction. 1 is forward and 0 is reverse. The speed / direction is DSPD.

<0x47><Session><Speed/Dir>

Note that any valid speed / direction message to that session should also act as a keepalive and reset any command station timer. Also, the speed format follows that of the NMRA DCC where speed 1 is considered as emergency stop. For normal running, a CAB should send speeds of 0, 2, 3 .. . etc. It is recommended that when speed is being changed, the rate of sending DSPD messages is limited to prevent congestion on the bus. This rate is a compromise between too many bus messages and too sluggish a loco response. Experience with the MERG system which has 32 possible locos active (32 slots in the stack) shows that a 32 millisecond interval between speed change messages from any one CAB is satisfactory.

9.1.7. Function control.

There are presently two methods of setting functions. The first, using DFUN, is the one currently implemented by the MERG command station and follows the NMRA DCC function byte protocol. The second uses DFNON and DFNOF.

9.1.7.1 Function control using DFUN

With DFUN, the CAB sends the function byte in the DCC format. This makes life easier for the command station.

<0x60><Session><FR><Fn byte>

Here FR is the function range as shown in the specification section 12.3and Fn byte is the DCC formatted function command byte for that range. This CBUS message will be sent for any requested change in a function setting. In accordance with the NMRA recommendations, functions F0 to F12 should be regularly refreshed while functions F13 to F28 may be sent once only. The information returned in a PLOC or QLOC message will contain the three function control bytes covering ranges F0 to F4, F5 to F8 and F9 to F12 as these will normally be contained in the stack.

9.1.7.2 Function control using DFNON & DFNOF

The use of DFNON and DFNOF simplifies the setting of functions by the CAB or computer but needs additional processing by the command station. The format is very simple.

DFNON (Function ON)

<0x49><Session><Fnum>

DFNOF (Function OFF)

<0x4A><Session><Fnum>

As Fnum is a byte it allows for 256 possible functions but only F0 to F28 are presently defined by the NMRA.

9.1.8. Emergency stop.

The CBUS protocol has a request emergency stop all OpCode, RESTP. . This can be used to stop all locos via the command station. To initiate an emergency stop all, a cab (or PC software) should issue the single byte RESTP message:

<0x0A>

The command station should respond by issuing the broadcast stop DCC packet to the track, and sending the track stopped ESTOP single byte message on CBUS:

<0x06>

All other cabs should then respond by displaying a suitable message to the user and issuing no more non-zero speed packets until stop all is cleared for that train. Keep alive messages should continue to be sent. On the MERG cabs this requires the user to reset the knob to zero which then allows the train to be restarted from rest. A cab with an encoder or a software throttle such as JMRI can simply set the displayed speed to zero and then similarly allow the user to restart from rest.

However, we have found that the ability to stop just ‘your’ loco is useful. This is simply achieved by sending a speed / dir message (DSPD) with the speed set to 1.

In addition to the command station receiving the ESTOP, all other active CABs should recognize this and act appropriately, e.g. cease sending DSPD messages and indicate a ‘stop all’ has been issued.

9.2. Consisting.

The CBUS protocol has OpCodes for ‘advanced consisting’ where the loco is put into consist mode by writing the consist address to CV19. OPC is PCON.

<0x45><Session><Consist#>

The session is that of the loco to be put into the consist. Consist# is the 7 bit consist address with bit 7 indicating the direction in the consist. Bit 7 set reverses the direction.

To run the consist, you establish a session for the consist address as if it were a loco with a short address. The NMRA advanced consisting principle does not allow you to control a single loco and the consist separately if both have the same short address.
The CBUS specification also has the OpCode KCON for removing a loco from a consist but this functions identically to PCON but with the Consist# set to 0. You need to establish a session for the loco by its own address, not that of the consist.

<0x46><Session><0>

By using PCON, you can change a loco from one consist to another without using KCON.

Note: For anyone wishing to use the so called ’universal consisting’ where locos are sent individual speed and direction packets in sequence, the command station needs just to know which locos in its stack are also in a consist and send the appropriate speed and direction packet to each. In this case, PCON can be interpreted by the command station in a different way. It is not practicable to run advanced consisting and universal consisting at the same time.

The opcode QCON, Query Consist, is provided to allow reading back which locos are in a consist. If a command station implements advanced consisting, and has no way of reading back the CV containing the consist address in the loco, then it need not support this opcode.

9.3. CV programming.

The CBUS specification has OpCodes to allow all the NMRA specified programming modes. This includes On The Main (OTM) programming and full 'service mode' programming and CV readback. Currently, readback OTM using systems like RailCom[tm] has not been implemented but CBUS would be very suitable as a 'feedback' bus from any track detector scheme.

9.3.1. Ops Mode (On The Main)

OTM programming uses WCVO for byte write and WCVB for bit mode write. For byte write

<0x82><Session><High CV#><Low CV#><Val>

The session is that of the loco to be programmed. The two CV bytes allow for 65535 possible CVs but only 1023 are allowed by the NMRA. Val is a byte to be entered into the CV. For bit mode write, use WCVB

<0x83><Session><High CV#><Low CV#><Val>

Here Val is a byte as defined in the NMRA RP 9.2.1 for OTM bit manipulation in a DCC packet.

9.3.2. Service mode (Programming track)

Service mode read and write of CVs are allowed in direct, page, register and address only modes.

To write a CV in service mode use WCVS

<0xA2><Session><High CV#><LowCV#><Mode><CVval>

As a loco in service mode will be on the programming track, it need not have a specific address. Thus the 'session' is only used to link a CAB or programmer software to the command station / programming device. Any 'session' will do. You can continue running a loco on the main and use its 'session' to program a different loco on the programming track.
The 'mode' byte defines the service mode to be used and CVVal is the value to be written, or, if in 'bit manipulation' mode, the appropriate bit pattern as defined in RP 9.2.2. The modes are shown in the specification section but are repeated here for convenience:

0 Direct Byte
1 Direct Bit
2 Page Mode
3 Register Mode
4 Address Only Mode

A command station / programmer may not support all modes.

Following the WCVS message, the command station / programmer will respond with SSTAT.

<0x4C><Session><Status>

The status indicates the result of the write process. Again this is defined in the specification section but repeated here:

0 Reserved
1 No Acknowledge
2 Overload on service mode programming track
3 Write Acknowledge
4 Busy
5 CV out of range

Reading a CV follows the same process. The OpCode is QCVS.

<0x84><Session><High CV#><Low CV#><Mode>

A successful read results in a PCVS.

<0x85><Session><High CV#><Low CV#><Val>

Where Val is the value of the CV as a HEX byte. If the read process fails, a SSTAT message will be sent by the programmer with the reason for failure in the ‘status’ byte.

9.4. Direct transmission of DCC packets.

CBUS also has a set of OpCodes which allow a ‘device’, usually a PC, to act as a command station and send messages in direct NMRA DCC format. The track interface will now simply take these messages and convert them directly into a DCC packet with the correct bit format and timing. These OpCodes are RDCC3, RDCC4, RDCC5 and RDCC6. The minimum DCC packet has three bytes and the maximum had six bytes, including the error byte. The OpCode also includes a byte specifying the number of times the DCC packet is to be sent.

9.5. Layout control from CABs.

It may be useful to control accessories on the layout directly from CABs. This simply requires CABs to send the appropriate CBUS event. However, CABs are considered ‘generic’ in that they are all essentially the same and so have no unique Node Number. By default, all cabs have the same constant node number of 0xFFFF. This means that any ‘long’ events sent from a cab will always have the cab node number 0xFFFF. Therefore long events are not really suitable for general purpose layout control from a cab, instead short events are more suitable as the device number used can refer to the device being controlled. Long events are, however, suited to events issued from a cab that are directly related to a cab operation.

There is no restriction on the events a CAB can send for accessory control. Sequences of events are possible, depending in the implementation.

A couple of examples: If F3 always works a horn, the effect can be enhanced by teaching the F3 cab CBUS event to a consumer that sounds a horn via a larger speaker under the layout. Thus non-sound fitted locos would still get a horn noise. A function that can operate a layout uncoupler could give a coupling noise at the same time.

A user can also teach events from function buttons that are not used for their locos as quick access layout controls, such as often used routes. For example, a user who is not using sound is unlikely to use any loco functions above F4 or so, leaving F5 upwards available for quick access layout control.

The use of these events is entirely up to the user, and can be ignored if the user does not desire to use this feature.

9.6. Cab signalling

The cab data opcode allows a layout control system to send signal aspects that can be displayed on a cab handset to provide cab signalling. The layout control system is responsible for tracking and knowing where a given train is, and the aspect of the signal ahead of it. It is likely that this layout control system would be a computer program, such as JMRI.

The layout control system should send the following CABDAT CBUS message whenever the signal aspect ahead of the train changes, either because the train has moved past one signal so the next signal is now a different one, or because the signal ahead has changed aspect:

<0xC2><addrH><addrL><datcode><aspect1><aspect2><speed>

<addrH > and <addrL> are the 2 byte loco address in the same format as other opcodes described above that specify a loco address, such as RLOC and GLOC.

<datcode> is set to 01 to indicate that the CABDAT opcode is being used for cab signalling (other values of <datcode> may be defined to send other types of data to cabs).

The aspect is specified in 2 bytes, to future proof it by allowing for all other aspect combinations that might be possible in signalling systems from different countries.

The first <aspect1> byte is defined as follows:

Bits 0-1 - Two bit code 00=danger (red or home semaphore horizontal in UK),
01=caution (yellow or distant semaphore horizontal in UK)
10=preliminary caution (double yellow in UK)
11=proceed (green or semaphore arm angled up or down in UK)

Bit 2 - Set 1 for calling on aspect (bits 0-1 would be set to 00 for danger when calling on)
Bit 3 - Set 0 means upper nibble is feather location, set 1 means upper nibble is route indicator
Bits 5-8 - Set 0 = no route indicated, 1 to 6 = feather position or 1 to 16 for theatre indication **

For most signalling systems, the basic bit definitions for danger, caution, preliminary caution and proceed should be the same. Other bit definitions will be dependant on the signalling system in use, the layout control system and the cab firmware.

** For UK semaphore signalling, there are multiple arms for different routes on a mast, these bits could indicate which of the arms has been pulled off, which is equivalent to a feather on colour light signalling. However, this is not mandated, as it may be difficult to implement for some layout control systems.

The second byte can be used for additional requirements for various signalling systems, such as lit/unlit, lunar etc.

For UK signalling, <aspect2> bit 0 is defined as a flashing aspect, applicable to caution, preliminary caution and proceed aspects.

<**speed**> is a speed limit indication that a cab may optionally display to the driver, if both layout control system and cab support it. If <speed> is not implemented by a layout control system, or whenever speed limit information is not available, this byte should be set to 0xFF (255).

A simple cab signalling display will probably just show red if the aspect is danger and green for everything else. Likewise UK semaphore signalling would only use a subset.

10. The computer interface protocol.

Unless CBUS is used only in the SLiM mode, connection to a computer is essential, if only for configuration. It can also be useful for DCC decoder programming, e.g with DecoderPro. Virtually all PCs have a USB socket and some (still) have a serial RS232 socket. Simple interface modules have been designed for both USB and RS232 but the use of the former is recommended.

Both USB and RS232 are serial interfaces and require a means of delimiting each message. As CBUS messages themselves can contain any bytes, it was decided to use an ASCII string protocol to communicate between the PC and the CBUS interface module. There are several commercial interface protocols for USB to CAN communication. We have used the serial protocol published by the 'Gridconnect' company, albeit with a slight modification.

The information on the serial side uses ASCII characters. This simplifies message parsing by the PC and is compatible with most software. However, the structure of the ASCII string follows that of a CAN frame so there is direct correspondence between the CAN frame and the serial string.

There are also several modules to provide wired or wireless network connection between CBUS and an Ethernet network. These modules use the same Gridconnect protocol, encapsulated in TCP/IP packets.

10.1. The header.

Following the 'Gridconnect' scheme, the ASCII string starts with a ":" followed by an "S" to indicate a Standard CAN frame or an "X" for an extended frame. CBUS only uses Standard frames but all of the computer interface modules allow for both types of frame so can be used for bootloading (see below).

Some of the networking modules use other codes other than S or X after the ":" for housekeeping messages, so any software tool should ignore any Gridconnect packets that do not start with :S or :X.

The next 4 chars are the ASCII version of the two header bytes in HEX for a standard frame or 8 chars for the four bytes of an extended header. This is departure from the Gridconnect format as CBUS uses a 7 bit node ID and 4 priority bits rather than just an 11 bit number. These two bytes map directly into the bytes sent and received by the CAN processor as SIDH and SIDL. (Standard IDentifier High byte and Standard IDentifier Low byte) For an extended header, the four bytes map directly to the SIDH, SIDL, EIDH and EIDL.

An example would be where the CBUS priority bits are 1011 and the CAN ID number is 0000001. These bits become the two bytes of the CAN header as follows

10110000 00100000 or in HEX form, B020. SIDH is B0 and SIDL is 20. This gives the string so far as :SB020 or in ASCII,

3A 53 42 30 32 30

The CANUSB and CANRS transfer the complete Gridconnect message between the computer and CBUS witjhout changing the CAN header. This means that the CAN ID of any message sent onto the CBUS will be set by the computer software.

The CANEther module, for networking connections, is different in this respect. It subsitutes the CAN ID received from the computer with its own CAN ID before transmitting on to CBUS. This is necessary to support bridge mode, where the CBUS message may be destined for another CAN segment, to avoid the possibility of CAN ID conflicts.

10.2. The frame type

The next character is either “N” or “R” signifying a Normal or a RTR frame (RTR is Remote Transfer Request). Except during the self enumeration process, CBUS only uses Normal frames.

10.3. The data segment

A CBUS frame has up to 8 data bytes and the remainder of the string is the data bytes in ASCII (HEX) form. The string is concluded by a “;” Note, there is no value indicating the number of data bytes. This is worked out by the firmware in the interface module. If a frame has all 8 data bytes then the format for a normal frame is as follows.

:ShhhhNd0d1d2d3d4d5d6d7;

Where hhhh is the two byte header and d0 to d7 are the 8 data bytes. If the header is B020 as above and the data is 1,2,3,4,5,6,7,8 then the ASCII string becomes

3A 53 42 30 32 30 4E 30 31 30 32 30 33 30 34 30 35 30 36 30 37 30 38 3B

11. The Bootloader.

CBUS has an OpCode to put a numbered node into its ‘boot’ mode. This allows an update or complete change of the code in the node processor over the CBUS itself. (BOOTM)

<0x5C><NN hi><NN lo>

Clearly, the actual bootloading process is processor dependent and the processor itself must contain code that allows for bootloading. Also the bootloader program is responsible for taking the node out of boot mode and restoring it to normal functioning. With the existing bootloader code, the last byte of EEPROM is set to 00 for normal running and any non-zero value for bootloading.

(Note: The MERG modules all use the Microchip 18F2x80 or 18F2xK80 processors and the bootloader code is a slightly modified version of one supplied by Microchip.)

As far as CBUS goes, there is one big difference when using the bootloader and that is the use of extended CAN frames. The above processors, and others as well, suit loading in 8 byte blocks. By using the extended CAN header, all 8 data bytes may be used for the code. The ‘commands’ used by the Microchip code are included in the header. However, only one CBUS module should be in the bootloader mode at any one time or the new code will be loaded into more than one module.

The other advantage of using extended frames for bootloading is the processors can be configured to accept only standard frames or only extended frames. This allows bootloading of one (or more) modules to take place simultaneously with normal CBUS operation without interference. In effect, it creates a separate independent ‘channel’.

Note. It might be possible to use this alternative channel for streaming large amounts of data to a module (sound files?) or between PCs connected to the bus.

12. Reference section.

This comprises the formal specification document for CBUS 4 Rev 8j. This is current as of 27th October 2018. However, it will be subject to updates and revisions, including the addition of OpCodes as required. Any such changes will be 'backward compatible' so systems using a new revision will not become inoperable. The specification document is also available as a separate file.

There is a separate document (MERG CBUS Modules.doc) for reference and details of all the existing MERG CBUS modules. This is of particular relevance to those writing software or configuration tools for these modules. It will also act as guidance for developers of new hardware modules. However, the actual implementation is not part of the CBUS Specification.

12.1. CBUS 4.0 Specification Rev. 8j

Note. To preserve compatibility, no changes to the protocol or additional OpCodes should be made without the approval of the primary author. Such changes, if agreed, will require the issue of an updated specification document. (mike@threerivers.wanadoo.co.uk)

Update history:

Draft 7c	by Andrew Crosland, Mike Bolton and Roger Healey 22/11/09
Draft 7d	Updates to rev 7c by Mike Bolton 12/01/11
Draft 7e	Update (OPC 0x59 only) by Mike Bolton 05/04/11
Draft 7f	Updates Added OPCs (Pete Brownlow) and one correction 13/04/11
Draft 7g	Added OPCs for short data events and requests. MB. 02/07/11 Added OPCs for RQMN,NAME,DFNON,DFNOF,QNN,renamed FliM setup opcodes to match implementation, reinstated DCC session keep alive. PNB 04/07/11 Updated definition of BOOTM (0x5C) Added Appendix 1, Node Parameter Definitions RKH 04/07/11 (Now section 7.2.2.1)
Draft 7h	Major changes including added OPCs, changed mnemonics and some OPCs moved. (new values). MPB 03/08/11
Draft 8a	Added opcodes PNN, GLOC, FCLK, some new error messages and description updates by Pete Brownlow 18/2/12
Draft 8b	Minor changes to format. Mike Bolton, 13/07/12
Version 8c	Added OpCodes ENUM and CANID 02/08/12. Dropped use of 'Draft'.
Version 8d	Added OpCode ALOC (0x43). Reinstated OpCode QCON (0x41) Minor typing corrections. Mike Bolton (30/06/15) Added outline numbering, auto-gen contents table & cross refs. Pete Brownlow 6/7/15

12.2. Communication Protocol

General CAN message format:

[<MjPri><MinPri><ID>] <Opcode><Dat0> ..<DatN>

where:

- **<MjPri>** bits 9 – 10 of the CAN header. Dynamic Priority, elevated by the node to gain access based on a transmit fail count. Values:
  0 – Emergency priority
  1 – High priority
  2 – Normal priority

- <**MinPri>** bits 7 - 8 of the CAN header. Static priority based on message and node type. Values:
  0 – High access
  1 – Above Normal access
  2 – Normal access
  3 – Low access
- **<CANID>** bits 0 – 6 of the CAN header, is a CAN segment-unique ID, assigned via enumeration.
- **<Opcode>** the first data byte is the opcode which includes the length of the message in the upper 3 bits.

In some associated documents, the Opcode is also referred to as the 'command' byte. The abbreviation OPC may also be used. In this document the Opcodes are in hexadecimal.

12.3. Packet Definitions (by OPC field)

The first column is a decimal OPC reference number. The second column is the actual OPC in hexadecimal.

00-1F – 0 Data bytes packets

[<MjPri><MinPri><CAN ID>]<Opcode>

0. 00 General Acknowledgement (*ACK*)
   Format:
   [<MjPri><MinPri=2><CANID>]<00>
   Positive response to query/ request performed or report of availability on-line.

1. 01 General No Ack (*NAK*)
   Format:
   [<MjPri><MinPri=2><CANID>]<01>
   Negative response to query/ request denied.

2. 02 Bus Halt (*HLT*)
   Format:
   [<MjPri><MinPri=0><CANID>]<02>
   Commonly broadcasted to all nodes to indicate CBUS is not available and no further packets should be sent until a BON or ARST is received.

3. 03 Bus ON (*BON*)
   Format:
   [<MjPri><MinPri=1><CANID>]<03>
   Commonly broadcasted to all nodes to indicate CBUS is available following a HLT.

4. 04 Track OFF (*TOF*)
   Format:
   [<MjPri><MinPri=1><CANID>]<04>
   Commonly broadcasted to all nodes by a command station to indicate track power is off and no further command packets should be sent, except inquiries.

5. 05 Track ON (*TON*)
   Format:
   [<MjPri><MinPri=1><CANID>]<05>
   Commonly broadcasted to all nodes by a command station to indicate track power is on.

6. 06 Emergency Stop (*ESTOP*)
   Format:
   [<MjPri><MinPri=1><CANID>]<06>
   Commonly broadcast to all nodes by a command station to indicate all engines have been emergency stopped.

7. 07 System Reset (*ARST*)
   Format:
   [<MjPri><MinPri=0><CANID>]<07>
   Commonly broadcasted to all nodes to indicate a full system reset.

8. 08 Request Track OFF (*RTOF*)
   Format:
   [<MjPri><MinPri=1><CANID>]<08>
   Sent to request change of track power state to “off”.

9. 09 Request Track ON (*RTON*)
   Format:
   [<MjPri><MinPri=1><CANID>]<09>
   Sent to request change of track power state to “on”.

10. 0A Request Emergency Stop ALL (*RESTP*)
    Format:
    [<MjPri><MinPri=0><CANID>]<0A>
    Sent to request an emergency stop to all trains . Does not affect accessory control. See section 9.1.8

11. 0B Reserved

12. 0C Request Command Station Status (*RSTAT*)
    Format:
    [<MjPri><MinPri=2><CANID>]<0C>
    Sent to query the status of the command station. See description of (STAT) for the response from the command station.

13. 0D Query node number (QNN)
    Format:
    [<MjPri><MinPri=3><CANID>]<0D>
    Sent by a node to elicit a PNN reply from each node on the bus that has a node number. See OpCode 0xB6

14. 0E Reserved

15. 0F Reserved

16. 10 Request node parameters(RQNP)
    Format:
    [<MjPri><MinPri=3><CANID>]<10>
    Sent to a node while in 'setup'mode to read its parameter set. Used when initially configuring a node. See section 7.2.3.

17. 11 Request module name (RQMN)
    Format:
    [<MjPri><MinPri=2><CANID>]<11>
    Sent by a node to request the name of the type of module that is in setup mode. The module in setup mode will reply with opcode NAME. See OpCode 0xE2

18. 12 Reserved

19. 13 Reserved

20. 14 Reserved

21. 15 Reserved

22. 16 Reserved

23. 17 Reserved

24 18 Reserved

25 19 Reserved

26 1A Reserved

27 1B Reserved

28 1C Reserved

29 1D Reserved

30 1E Reserved

31 1F Reserved

20–3F - 1 Data byte packets

<MjPri><MinPri><CAN ID>]<Opc><Dat1>

32 20 Reserved

33. 21 Release Engine (*KLOC*)
Format:
[<MjPri><MinPri=2><CANID>]<21><Session>
<**Dat1**> is the engine session number as HEX byte.
Sent by a CAB to the Command Station. The engine with that Session number is removed from the active engine list.

34. 22 Query engine (*QLOC*)
Format:
[<MjPri><MinPri=2><CANID>]<22><Session>
<**Dat1**> is the engine session number as HEX byte.
The command station responds with PLOC if the session is assigned. Otherwise responds with ERR: engine not found. See section 12.5.

35. 23 Session keep alive (DKEEP)
Format:
[<MjPri><MinPri=2><CANID>]<23><Session>
<**Dat1**> is the engine session number as HEX byte.

The cab sends a keep alive at regular intervals for the active session. The interval between keep alive messages must be less than the session timeout implemented by the command station.

36. 24 Reserved

37. 25 Reserved

38. 26 Reserved

39. 27 Reserved

40. 28 Reserved

41. 29 Reserved

42. 2A Reserved

43. 2B Reserved

44. 2C Reserved

45. 2D Reserved

46. 2E Reserved

47. 2F Reserved

48. 30 Debug with one data byte (*DBG1*)
    Format:
    [<MjPri><MinPri=2><CANID>]<30><Status>
    <**Dat1**> is a freeform status byte for debugging during CBUS module development. Not used during normal operation

49. 31 Reserved

50. 32 Reserved

51. 33 Reserved

52. 34 Reserved

53. 35 Reserved

54. 36 Reserved

55. 37 Reserved

56. 38 Reserved

57. 39 Reserved

58. 3A Reserved

59. 3B Reserved

60. 3C Reserved

61. 3D Reserved

62. 3E Reserved

63. 3F Extended op-code with no additional bytes (EXTC)
    Format:
    [<MjPri><MinPri=3><CANID>]<3F><Ext_OPC>
    Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

40–5F - 2 data byte packets

[<MjPri><MinPri><CAN ID>]<Opc><Dat1><Dat2>

64. 40 Request engine session (*RLOC*)

Format:
[<MjPri><MinPri=2><CANID>]<40><Dat1><Dat2 >
<**Dat1**> and <**Dat2**> are [AddrH] and [AddrL] of the decoder, respectively.
7 bit addresses have (AddrH=0).
14 bit addresses have bits 6,7 of AddrH set to 1.
The command station responds with (PLOC) if engine is free and is being assigned. Otherwise responds with (ERR): engine in use or (ERR:) stack full. This command is typically sent by a cab to the command station following a change of the controlled decoder address. RLOC is exactly equivalent to GLOC with all flag bits set to zero, but command stations must continue to support RLOC for backwards compatibility.

65. 41 Query Consist (*QCON*)

Format:
[<MjPri><MinPri=2><CANID>]<41><ConID><Index>
<**Dat1**> is consist address.
<**Dat2**> is engine index in the consist.
Allows enumeration of a consist. Command station responds with PLOC if an engine exists at the specified index, otherwise responds with
ERR: no more engines
Note that a command station need not support this opcode if it uses advanced consisting and has no way of reading back the CV currently containing the consist address in a loco.

66. 42 Set Node Number (*SNN*)

Format:
[<MjPri><MinPri=3><CANID>]<42><NNHigh><NNLow>
<**Dat1**> is high byte of the node number.
<**Dat2**> is low byte of the node number.
Sent by a configuration tool to assign a node number to a requesting node in response to a *RQNN* message.
The target node must be in 'setup' mode.

67. 43 Allocate loco to activity. (ALOC)

Format:
[<MjPri><MinPri=2><CANID>]<43><Session ID><Allocation code >
<**Dat1**> is Session ID.
<**Dat2**> is application specific allocation code. (one byte)

68. 44 Set CAB session mode (STMOD)

Format:
[<MjPri><MinPri=2><CANID>]<44><Session><MMMMMMMM>
<**Dat1**> Session number
<**Dat2>** contains mode bits:
0 – 1: speed mode
00 – 128 speed steps
01 – 14 speed steps
10 – 28 speed steps with interleave steps
11 – 28 speed steps
2: service mode
3: sound control mode

69. 45 Consist Engine (*PCON*)

Format:
[<MjPri><MinPri=2><CANID>]<45><Session><Consist#>
<**Dat1**> Session number
<**Dat2**> is consist address (8 bits).
Adds a decoder to a consist.
Dat2 has bit 7 set if consist direction is reversed.

70. 46 Remove Engine from consist (*KCON*)

Format:
[<MjPri><MinPri=2><CANID>]<46><Session><Consist#>
<**Dat1**> loco session number
<**Dat2**> is consist address.
Removes a loco from a consist.

71. 47 Set Engine Speed/Dir (*DSPD*)

Format:
[<MjPri><MinPri=2><CANID>]<47><Session><Speed/Dir>
<**Dat1**> session number
<Dat2> is speed/dir value, where the most significant bit is direction and the 7ls bits are the unsigned speed value. Sent by a CAB or equivalent to request an engine speed/dir change.

72. 48 Set Engine Flags (*DFLG*)

Format:
[<MjPri><MinPri=2><CANID>]<48><Session><DDDDDDDD>
<**Dat1**> Session number
<**Dat2>** is the flags:
Bits 0-1: Speed Mode
00 – 128 speed steps
01 – 14 speed steps
10 – 28 speed steps with interleave steps
11 – 28 speed steps
Bit 2: Lights On/OFF
Bit 3: Engine relative direction
Bits 4-5: Engine state (active =0 , consisted =1, consist master=2, inactive=3)
Bits 6-7: Reserved.
Sent by a cab to notify the command station of a change in engine flags.

73. 49 Set Engine function on (*DFNON*)

Format:
[<MjPri><MinPri=2><CANID>]<49><Session><Fnum>
<Dat1> is the engine session number.
<**Dat2**> is the function number – 0 to 27.

Sent by a cab to turn on a specific loco function. This provides an alternative method to DFUN for controlling loco functions. A command station must implement both methods.

74. 4A Set Engine function off (*DFNOF*)

Format:
[<MjPri><MinPri=2><CANID>]<4A><Session><Fnum>
<Dat1> is the engine session number.
<**Dat2**> is the function number – 0 to 27.

Sent by a cab to turn off a specific loco function. This provides an alternative method to DFUN for controlling loco functions. A command station must implement both methods.

75. 4B Reserved

76. 4C Service mode status. (SSTAT)
Format:
[<MjPri><MinPri=3><CANID>]<4C><Session><Status>
Status returned by command station/programmer at end of programming operation that does not return data.

77. 4D Reserved

78. 4E Reserved

79. 4F Reset to manufacturers defaults (NNRSM)
Format:
[<MjPri><MinPri=3><CANID>]<4F><NN hi><NN lo>

Causes the module to reset settings to manufacturers defaults.
The module should retain any node number and remain in FLiM mode.
What the manufacturers defaults are will be defined for each module, but should be equivalent to putting a new module into FLiM, with no events taught, only default events defined (if any) and all Nvs returned to their default values.

80. 50 Request node number (RQNN)
Format:
[<MjPri><MinPri=3><CANID>]<50><NN hi><NN lo>

Sent by a node that is in setup/configuration mode and requests assignment of a node number (NN). The node allocating node numbers responds with (SNN) which contains the newly assigned node number. <NN hi> and <NN lo> are the existing node number, if the node has one. If it does not yet have a node number, these bytes should be set to zero.

81. 51 Node number release (NNREL)
Format:
[<MjPri><MinPri=3><CANID>]<51><NN hi><NN lo>
Sent by node when taken out of service. e.g. when reverting to SLiM mode.

82. 52 Node number acknowledge. (NNACK)
Format:
[<MjPri><MinPri=3><CANID>]<52><NN hi><NN lo>
Sent by a node to verify its presence and confirm its node id. This message is sent to acknowledge an SNN.

83. 53 Set node into learn mode (NNLRN)
Format:
[<MjPri><MinPri=3><CANID>]<53><NN hi><NN lo>
Sent by a configuration tool to put a specific node into learn mode.

84. 54 Release node from learn mode (NNULN)
Format:
[<MjPri><MinPri=3><CANID>]<54><NN hi><NN lo>
Sent by a configuration tool to take node out of learn mode and revert to normal operation.

85. 55 Clear all events from a node (NNCLR)

Format:
[<MjPri><MinPri=3><CANID>]<55><NN hi><NN lo>
Sent by a configuration tool to clear all events from a specific node. Must be in learn mode first to safeguard against accidental erasure of all events.

86. 56 Read number of events available in a node (NNEVN)

Format: [<MjPri><MinPri=3><CANID>]<56><NN hi><NN lo>
Sent by a configuration tool to read the number of available event slots in a node. Response is EVLNF (0x70)

87. 57 Read back all stored events in a node (NERD)

Format:
[<MjPri><MinPri=3><CANID>]<57><NN hi><NN lo>
Sent by a configuration tool to read all the stored events in a node. Response is 0xF2.

88. 58 Request to read number of stored events (RQEVN)

Format:
[<MjPri><MinPri=3><CANID>]<58><NN hi><NN lo>
Sent by a configuration tool to read the number of stored events in a node. Response is 0x74(NUMEV).

89. 59 Write acknowledge (WRACK)

Format:
[<MjPri><MinPri=3><CANID>]<59><NN hi><NN lo>
Sent by a node to indicate the completion of a write to memory operation. All nodes must issue WRACK when a write operation to node variables, events or event variables has completed. This allows for teaching nodes where the processing time may be slow.

90. 5A Request node data event (RQDAT)

Format:
[<MjPri><MinPri=3><CANID>]<5A><NN hi><NN lo>
Sent by one node to read the data event from another node.(eg: RFID data). Response is 0xF7 (ARDAT).

91. 5B Request device data – short mode (RQDDS)

Format:
[<MjPri><MinPri=3><CANID>]<5B><DN hi><DN lo>
To request a ‘data set’ from a device using the short event method. where DN is the device number. Response is 0xFB (DDRS)

92. 5C Put node into bootload mode (BOOTM)

Format:
[<MjPri><MinPri=3><CANID>]<5C><NN hi><NN lo>

For SliM nodes with no NN then the NN of the command is must be zero. For SLiM nodes with an NN, and all FLiM nodes the command must contain the NN of the target node. Sent by a configuration tool to prepare for loading a new program.

93. 5D Force a self enumeration cycle for use with CAN (ENUM)
Format:
[<MjPri><MinPri=3><CANID>]<5D><NN hi><NN lo>

For nodes in FLiM using CAN as transport.. This OPC will force a self-enumeration cycle for the specified node. A new CAN_ID will be allocated if needed. Following the ENUM sequence, the node should issue a NNACK to confirm completion and verify the new CAN_ID. If no CAN_ID values are available, an error message 7 will be issued instead.

94. 5E Restart node(*NNRST*)
Format:
[<MjPri><MinPri=3><CANID>]<5E><NN hi><NN lo>

Causes module to carry out a software reset to restart the firmware.
No settings are affected.

95. 5F Extended op-code with 1 additional byte (EXTC1)
Format:
[<MjPri><MinPri=3><CANID>]<5F><Ext_OPC><byte>

Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

60-7F - 3 data byte packets

[<MjPri><MinPri><CAN ID>]<OPC><Dat1><Dat2><Dat3>

96. 60 Set Engine functions (*DFUN*)

Format:
[<MjPri><MinPri=2><CANID>]<60><Session><Fn1><Fn2>
<Dat1> is the engine session number.
<**Dat2**> is the function range.
1 is F0(FL) to F4
2 is F5 to F8
3 is F9 to F12
4 is F13 to F20
5 is F21 to F28
<**Dat3>** is the NMRA DCC format function byte for that range in corresponding bits. Sent by a CAB or equivalent to request an engine Fn state change.

97. 61 Get engine session (*GLOC*)

Format:
[<MjPri><MinPri=2><CANID>]<61><Dat1><Dat2><Flags>
<**Dat1**> and <**Dat2**> are [AddrH] and [AddrL] of the decoder, respectively.
7 bit addresses have (AddrH=0).
14 bit addresses have bits 6,7 of AddrH set to 1.
<Flags> contains flag bits as follows:
Bit 0: Set for "Steal" mode
Bit 1: Set for "Share" mode

Both bits set to 0 is exactly equivalent to an RLOC request
Both bits set to 1 is invalid, because the 2 modes are mutually exclusive
The command station responds with (PLOC) if the request is successful.
Otherwise responds with (ERR): engine in use. (ERR:) stack full or (ERR) no session.
The latter indicates that there is no current session to steal/share depending on the flag bits set in the request.
GLOC with all flag bits set to zero is exactly equivalent to RLOC, but command stations must continue to support RLOC for backwards compatibility.
See section 9.1.2. for a detailed description of the use of DCC loco sessions.

98. 62 Reserved

99. 63 Command Station Error report (*ERR*)

Format:
[<MjPri><MinPri=2><CANID>]<63><Dat 1><Dat 2><Dat 3>

Sent in response to an error situation by a command station.

For use of the data bytes and error codes see Section 12.5

100. 64 Reserved

101. 65 Reserved

102. 66 Reserved

103. 67 Reserved

104. 68 Reserved

105. 69 Reserved

106. 6A Reserved

107. 6B Reserved

108. 6C Reserved

109. 6D Reserved

110. 6E Reserved

111. 6F Error messages from nodes during configuration (CMDERR)
Format:
[<MjPri><MinPri=3><CANID>]<6F><NN hi><NN lo><Error number>
Sent by node if there is an error when a configuration command is sent.
See Section 12.4for details of the error codes.

112. 70 Event space left reply from node (EVNLF)
Format:
[<MjPri><MinPri=3><CANID>]<70><NN hi><NN lo><EVSPC>
EVSPC is a one byte value giving the number of available events left in that node.

113. 71 Request read of a node variable (NVRD)
Format:
[<MjPri><MinPri=3><CANID>]<71><NN hi><NN lo><NV#>
NV# is the index for the node variable value requested. Response is NVANS.

114. 72 Request read of stored events by event index (NENRD)
Format:
[<MjPri><MinPri=3><CANID>]<72><NN hi><NN lo><EN#>
EN# is the index for the stored event requested.
Response is 0xF2 (ENRSP)

115. 73 Request read of a node parameter by index (RQNPN)
Format:
[<MjPri><MinPri=3><CANID>]<73><NN hi><NN lo><Para#>
Para# is the index for the parameter requested. Index 0 returns the number of available parameters
Response is 0x9B (PARAN) See section 7.2.3 for details of the node parameters.

116. 74 Number of events stored in node (NUMEV)
Format:
[<MjPri><MinPri=3><CANID>]<74><NN hi><NN lo><No.of events>
Response to request 0x58 (RQEVN)

117. 75 Set a CAN_ID in existing FLiM node (CANID)
Format:
[<MjPri><MinPri=3><CANID>]<75><NN hi><NN lo><CAN_ID >
Used to force a specified CAN_ID into a node. Value range is from 1 to 0x63 (99 decimal) This OPC must be used with care as duplicate CAN_IDs are not allowed.. Values outside the permitted range will produce an error 7 message.and the CAN_ID will not change.

118. 76 Reserved

119. 77 Reserved

120. 78 Reserved

121. 79 Reserved

122. 7A Reserved

123. 7B Reserved

124. 7C Reserved

125. 7D Reserved

126. 7E Reserved

127. 7F Extended op-code with 2 additional bytes (EXTC2)
Format:
[<MjPri><MinPri=3><CANID>]<7F><Ext_OPC><byte1><byte2>
Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

80-9F - 4 data byte packets

[<MjPri><MinPri><CAN ID>]<Opc><Dat1><Dat2><Dat3><Dat4>

128. 80 Request 3-byte DCC Packet (*RDCC3*)
Format:
[<MjPri><MinPri=2><CANID>]<80><REP><Byte0>..<Byte2>
<**Dat1(REP)**> is number of repetitions in sending the packet.
<**Dat2>..<Dat4>** 3 bytes of the DCC packet.
Allows a CAB or equivalent to request a 3 byte DCC packet to be sent to the track. The packet is sent <REP> times and is not refreshed on a regular basis.
Note: a 3 byte DCC packet is the minimum allowed.

129. 81 Reserved

130. 82 Write CV (byte) in OPS mode *(WCVO)*
Format:
[<MjPri><MinPri=2><CANID>]<82><Session><High CV#><Low CV#><Val>
<**Dat1**> is the session number of the loco to be written to
<**Dat2>** is the MSB # of the CV to be written (supports CVs 1 - 65536)
<**Dat3>** is the LSB # of the CV to be written
<**Dat4>** is the byte value to be written
Sent to the command station to write a DCC CV byte in OPS mode to specific loco.(on the main)

131. 83 Write CV (bit) in OPS mode (WCVB)
Format:
[<MjPri><MinPri=2><CANID>]<83><Session><High CV#><Low CV#><Val>
<**Dat1**> is the session number of the loco to be written to
<**Dat2>** is the MSB # of the CV to be written (supports CVs 1 - 65536)
<**Dat3>** is the LSB # of the CV to be written
<**Dat4>** is the value to be written

The format for Dat4 is that specified in RP 9.2.1 for OTM bit manipulation in a DCC packet.
This is '111CDBBB' where C is here is always 1 as only 'writes' are possible OTM. (unless some loco ACK scheme like RailCom is used). D is the bit value, either 0 or 1 and BBB is the bit position in the CV byte. 000 to 111 for bits 0 to 7.
Sent to the command station to write a DCC CV in OPS mode to specific loco.(on the main)

132. 84 Read CV (*QCVS*)
Format:
[<MjPri><MinPri=2><CANID>]<84><Session><High CV#><Low CV#><Mode>
<**Dat1>** is the session number of the cab
<Dat2> is the MSB # of the CV read (supports CVs 1 - 65536)
<**Dat3>** is the LSB # of the CV read
<**Dat4>** is the programming mode to be used

This command is used exclusively with service mode.
Sent by the cab to the command station in order to read a CV value. The command station shall respond with a PCVS message containing the value read, or SSTAT if the CV cannot be read.

133. 85 Report CV (*PCVS*)

Format:
[<MjPri><MinPri=2><CANID>]<85><Session><High CV#><Low CV#><Val>
<**Dat1>** is the session number of the cab
<Dat2> is the MSB # of the CV read (supports CVs 1 - 65536)
<**Dat3>** is the LSB # of the CV read
<**Dat4>** is the read value
This command is used exclusively with service mode.
Sent by the command station to report a read CV.

134. 86 Reserved

135 87 Reserved

136. 88 Reserved

137. 89 Reserved

138 8A Reserved

139. 8B Reserved

140. 8C Reserved

141. 8D Reserved

142. 8E Reserved

143. 8F Reserved

144. 90 Accessory ON (ACON)

Format:
[<MjPri><MinPri=3><CANID>]<90><NN hi><NN lo><EN hi><EN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number

Indicates an 'ON' event using the full event number of 4 bytes. (long event)

145. 91 Accessory OFF (ACOF)

Format:
[<MjPri><MinPri=3><CANID>]<91><NN hi><NN lo><EN hi><EN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number

Indicates an 'OFF' event using the full event number of 4 bytes. (long event)

146. 92 Accessory Request Event (AREQ)

Format:
[<MjPri><MinPri=3><CANID>]<92><NN hi><NN lo><EN hi><EN lo>
<**Dat1>** is the high byte of the node number (MS WORD of the full event #)
<**Dat2>** is the low byte of the node number (MS WORD of the full event #)
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number

Indicates a 'request' event using the full event number of 4 bytes. (long event)
A request event is used to elicit a status response from a producer when it is required to know the 'state' of the producer without producing an ON or OFF event and to trigger an event from a 'combi' node.

147. 93 Accessory Response Event (ARON)

Format:
[<MjPri><MinPri=3><CANID>]<93><NN hi><NN lo><EN hi><EN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number

Indicates an 'ON' response event. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

148. 94 Accessory Response Event (AROF)

Format:
[<MjPri><MinPri=3><CANID>]<94><NN hi><NN lo><EN hi><EN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number

Indicates an 'OFF' response event. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

149. 95 Unlearn an event in learn mode (EVULN)

Format:
[<MjPri><MinPri=3><CANID>]<95><NN hi><NN lo><EN hi><EN lo>
Sent by a configuration tool to remove an event from a node.

150. 96 Set a node variable (NVSET)

Format:
[<MjPri><MinPri=3><CANID>]<96><NN hi><NN lo><NV# ><NV val>
Sent by a configuration tool to set a node variable. NV# is the NV index number.

151. 97 Response to a request for a node variable value (NVANS)

Format:
[<MjPri><MinPri=3><CANID>]<97><NN hi><NN lo><NV# ><NV val>
Sent by node in response to request. (NVRD)

Short events. (Device addressing)

Although the producer will send the complete 4 byte event number, the consumer will ignore the producer's node number bytes. This allows a "many to many" situation where producers like DCC handsets can activate the same accessories even though they will have unique node numbers. Clearly this limits the number of 'short' events to 64K-1. For short events, the lower two bytes define the 'Device Number' or DN. The DN can also be considered as a 'device address'.

For these short events, the full 4 byte event is still sent, both to keep the format the same and to allow identification of the producer when required.

152. 98 Accessory Short ON (ASON)

Format:
[<MjPri><MinPri=3><CANID>]<98><NN hi><NN lo><DN hi><DN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number

Indicates an 'ON' event using the short event number of 2 LS bytes.

153. 99 Accessory Short OFF (ASOF)

Format:
[<MjPri><MinPri=3><CANID>]<99><NN hi><NN lo><DN hi><DN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number

Indicates an 'OFF' event using the short event number of 2 LS bytes.

154. 9A Accessory Short Request Event (ASRQ)

Format:
[<MjPri><MinPri=3><CANID>]<9A><NN hi><NN lo><DN hi><DN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number

Indicates a 'request' event using the short event number of 2 LS bytes. A request event is used to elicit a response from a producer 'device' when it is required to know the 'state' of the device without producing an ON or OFF event and to trigger an event from a "combi" node.

155 9B Response to request for individual node parameter (PARAN)

Format:
[<MjPri><MinPri=3><CANID>]<9B><NN hi><NN lo><Para#><Para val>
NN is the node number of the sending node. Para# is the index of the parameter and Para val is the parameter value.

156 9C Request for read of an event variable (REVAL)

Format:
[<MjPri><MinPri=3><CANID>]<9C><NN hi><NN lo><EN#><EV#>
This request differs from B2 (REQEV) as it doesn't need to be in learn mode but does require the knowledge of the event index to which the EV request is directed.
EN# is the event index. EV# is the event variable index. Response is B5 (NEVAL)

157. 9D Accessory Short Response Event (ARSON)
Format:
[<MjPri><MinPri=3><CANID>]<9D><NN hi><NN lo><DN hi><DN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number

Indicates an 'ON' response event. A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

158. 9E Accessory Short Response Event (ARSOF)
Format:
[<MjPri><MinPri=3><CANID>]<9E><NN hi><NN lo><DN hi><DN lo>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number

Indicates an 'OFF' response event. A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

159. 9F Extended op-code with 3 additional bytes (EXTC3)
Format:
[<MjPri><MinPri=3><CANID>]<9F><Ext_OPC><byte1><byte2><byte3>
Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

A0-BF - 5 data byte packets

[<MjPri><MinPri><CAN ID>]<Opc><Dat1><Dat2><Dat3><Dat4><Dat5>

160. A0 Request 4-byte DCC Packet (*RDCC4*)
Format:
[<MjPri><MinPri=2><CANID>]<A0><REP><Byte0>..<Byte3>
<**Dat1(REP)**> is number of repetitions in sending the packet.
<**Dat2>..<Dat5>** 4 bytes of the DCC packet.
Allows a CAB or equivalent to request a 4 byte DCC packet to be sent to the track. The packet is sent <REP> times and is not refreshed on a regular basis.

161. A1 Reserved

162. A2 Write CV in Service mode (*WCVS*)
Format:
[<MjPri><MinPri=2><CANID>]<A2><Session><High CV#><LowCV#><Mode>
<CVval>
<**Dat1>** is the session number of the cab
<Dat2> is the MSB # of the CV to be written (supports CVs 1 - 65536)
<**Dat3>** is the LSB # of the CV to be written
<**Dat4>** is the service write mode
<**Dat5>** is the CV value to be written
Sent to the command station to write a DCC CV in service mode.

163. A3 Reserved

164. A4 Reserved

165. A5 Reserved

166. A6 Reserved

167. A7 Reserved

168. A8 Reserved

169. A9 Reserved

170. AA Reserved

171. AB Reserved

172. AC Reserved

173. AD Reserved

174. AE Reserved

175. AF Reserved

176. B0 Accessory ON (ACON1)

Format:
[<MjPri><MinPri=3><CANID>]<B0><NN hi><NN lo><EN hi>
<EN lo><data>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte

Indicates an 'ON' event using the full event number of
4 bytes with one additional data byte.

177. B1 Accessory OFF (ACOF1)

Format:
[<MjPri><MinPri=3><CANID>]<B1><NN hi><NN lo><EN hi>
<EN lo><data>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte

Indicates an 'OFF' event using the full event number of 4 bytes with one additional data byte.

178 B2 Read event variable in learn mode (REQEV)

Format:
[<MjPri><MinPri=3><CANID>]<B2><NN hi><NN lo><EN hi>
<EN lo><EV# >
Allows a configuration tool to read stored event variables from a node. EV# is the EV index. Reply is (EVANS)

179. B3 Accessory Response Event (ARON1)

Format:
[<MjPri><MinPri=3><CANID>]<B3><NN hi><NN lo><EN hi>
<EN lo><data>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is the additional data byte 1

Indicates an 'ON' response event with one additional data byte. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

180. B4 Accessory Response Event (AROF1)
Format:
[<MjPri><MinPri=3><CANID>]<B4><NN hi><NN lo><EN hi>
<EN lo><data>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is the additional data byte 1

Indicates an 'OFF' response event with one additional data byte. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

181. B5 Response to request for read of EV value (NEVAL)
Format:
[<MjPri><MinPri=3><CANID>]<B5><NN hi><NN lo><EN#>
<EV#><EVval>
NN is the node replying. EN# is the index of the event in that node. EV# is the index of the event variable. EVval is the value of that EV. This is response to 9C (REVAL)

182. B6 Response to Query Node (PNN)
Format:
[<MjPri><MinPri=3><CANID>]<B6><NN Hi><NN Lo><Manuf Id><Module Id><Flags>
<NN Hi**>** is the high byte of the node number
<NN Lo**>** is the low byte of the node number
<Manuf Id**>** is the Manufacturer id as defined in the node parameters
<Module Id**>** is the Module Type Id id as defined in the node parameters
<Flags> is the node flags as defined in the node parameters, see Section 7.2.3.

The Flags byte contains bit flags as follows:
Bit 0: Set to 1 for consumer node
Bit 1: Set to 1 for producer node
Bit 2: Set to 1 for FLiM mode
Bit 3: Set to 1 for Bootloader compatible

If a module is both a producer and a consumer then it is referred to as a "combi" node and both flags will be set.

Every node should send this message in response to a QNN message.

183. B7 Reserved

184. B8 Accessory Short ON (ASON1)
Format:
[<MjPri><MinPri=3><CANID>]<B8><NN hi><NN lo><DN hi><DN lo><data 1>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number
<**Dat5**> is the additional data byte 1

Indicates an 'ON' event using the short event number of 2 LS bytes with one added data byte.

185. B9 Accessory Short OFF (ASOF1)

Format:
[<MjPri><MinPri=3><CANID>]<B9><NN hi><NN lo><DN hi><DN lo><data 1>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number
<**Dat5**> is the additional data byte 1

Indicates an 'OFF' event using the short event number of 2 LS bytes with one added data byte.

186. BA Reserved

187. BB Reserved

188. BC Reserved

189. BD Accessory Short Response Event (ARSON1) with one data byte

Format:
[<MjPri><MinPri=3><CANID>]<BD><NN hi><NN lo><DN hi><DN lo><data 1>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number
<**Dat5**> is the additional data byte 1

Indicates an 'ON' response event with one added data byte. A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

190. BE Accessory Short Response Event (ARSOF1) with one data byte

Format:
[<MjPri><MinPri=3><CANID>]<BE><NN hi><NN lo><DN hi><DN lo><data 1>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number
<**Dat5**> is the additional data byte 1

Indicates an 'OFF' response event with one added data byte. A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

191. BF Extended op-code with 4 data bytes (EXTC4)

Format:
[<MjPri><MinPri=3><CANID>]<BF><Ext-OPC><byte1><byte2><byte3 > <byte4>

Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

C0-DF - 6 data byte packets

[<MjPri><MinPri><CAN ID>]<Opc><Dat1><Dat2><Dat3><Dat4><Dat5><Dat6>

192. C0 Request 5-byte DCC Packet (*RDCC5*)

Format:
[<MjPri><MinPri=2><CANID>]<C0><REP><Byte0>..<Byte4>
<**Dat1(REP)**> is # of repetitions in sending the packet.
<**Dat2>..<Dat6>** 5 bytes of the DCC packet.
Allows a CAB or equivalent to request a 5 byte DCC packet to be sent to the track. The packet is sent <REP> times and is not refreshed on a regular basis.

193. C1 Write CV (byte) in OPS mode by address (*WCVOA*)

Format:
[<MjPri><MinPri=2><CANID>]<C1><AddrH><AddrL><High CV#>
<Low CV#><Mode><Val>
<**Dat1**> and <**Dat2**> are [AddrH] and [AddrL] of the decoder, respectively.
7 bit addresses have (AddrH=0).
14 bit addresses have bits 7,8 of AddrH set to 1.
<Dat3> is the MSB # of the CV to be written (supports CVs 1 - 65536)
<**Dat4>** is the LSB # of the CV to be written
<**Dat5>** is the programming mode to be used
<**Dat6>** is the CV byte value to be written
Sent to the command station to write a DCC CV byte in OPS mode to specific loco (on the main). Used by computer based ops mode programmer that does not have a valid throttle handle.

194. C2 Cab Data (*CABDAT*)

Format:
[<MjPri><MinPri=2><CANID>]<0xC2><addrH><addrL><datcode><aspect1><aspect2><speed>

Transmitted by a layout control system to send data to a cab controlling a specific loco.

<**addrH**> and <**addrL**> are the loco address in the same format as RLOC and GLOC
7 bit (short) addresses have (addrH=0).
14 bit (long) addresses have bits 6,7 of addrH set to 1.

<**datcode**> defines the meaning of the remaining 3 bytes. Values of <datcode> may be defined as required. The following value has currently been defined:

01 - CABSIG - Transmitted by a layout control system to send signal aspects to be displayed on a cab handset as cab signalling

Where <datcode> is set to 01 CABSIG, the remaining 3 bytes are defined as follows:

<**aspect1**> is is signalling system independent, and is defined as follows (colours in brackets correspond to UK colour light signalling, the given aspect names may be displayed differently in other signalling systems):

Bits 0-1: 2 bit aspect code 00=danger (red), 01=caution (yellow),
10=preliminary caution (double yellow), 11=proceed (green)
Bit 2: 1 = calling on aspect (bits 0-1 are set to 00 for danger when calling on)
Bit 3: 0 indicates upper nibble is feather location, 1 for upper nibble is theatre type route indicator
Bits 5-8: 0 = no route indicated, 1 to 6 = feather position or 1 to 15 for theatre indication

<aspect1> should be set to 0xFF if no signal information is available. This can be used, for example, to indicate leaving a cab signalling area. A cab should extinguish any currently showing aspect on receipt of this code.

Note that because bits 0 and 1 should be set to zero when bit 2 is set, the code 0xFF is not otherwise a valid aspect.

<**aspect2**> may be used as required for other signalling systems.

For UK signalling, bit 0 set indicates a flashing aspect, applicable to caution, preliminary caution or proceed.

<**speed**> is a speed limit indication that a cab may optionally display to the driver. If <speed> is not implemented by a layout control system, or whenever speed limit information is not available, this byte should be set to 0xFF (255).

195. C3 Reserved

196. C4 Reserved

197. C5 Reserved

198. C6 Reserved

199. C7 Reserved

200. C8 Reserved

201. C9 Reserved

202. CA Reserved

203. CB Reserved

204. CC Reserved

205. CD Reserved

206. CE Reserved

207. CF Fast Clock (FCLK)
Format:
[<MjPri><MinPri=3><CANID>]<CF><mins><hrs><wdmon> <mday><temp>
<mins**>** is the minutes of the fast clock
<hrs**>** is the hours of the fast clock
<wdmon**>** bits 0-3 are the weekday (1=Sun, 2=Mon etc)
bits 4-7 are the month (1=Jan, 2=Feb etc)
<**div**> Set to 0 for freeze, 1 for real time
<mday> is day of the month 1-31
<temp> Temperature as twos complement -127 to +127

Used to implement a fast clock for the layout.

Note: This definition is at variance with the NMRA Addendum to RP 9.2.1, and is specific to CBUS.

This addendum defines a more complex time encoding as follows

The data bytes contains CCDDDDDD , there are four bytes in a time packet

CC = 00 DDDDDD = minutes in the range 0 – 59
CC = 10 DDDDDD = 0HHHHHH (sic) the hour in the rang 0-23 (note there is an extra H in the NMRA document)
CC = 01 DDDDDD = 000WWW the day of the week, 0 = Monday etc.
CC = 11 DDDDDD = 00FFFFF (sic) the acceleration factor, 1 means real time, 2 means real time * 2 etc (note there is an extra F in the NMRA document)

There is clearly some redundancy in the decodes, particularly CC = 01. There is no way to determine a date, so we could do something like the following

CC = 01 DDDDDD = 1ddddd where ddddd = the day of the month, range 1 to 31
CC = 01 DDDDDD = 01mmmm where mmmm = the month, January = 1

---

208. D0 Accessory ON (ACON2)

Format:
[<MjPri><MinPri=3><CANID>]<D0><NN hi><NN lo><EN hi><EN lo>
<data1><data2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2

Indicates an 'ON' event using the full event number of 4 bytes with two additional data bytes.

209. D1 Accessory OFF (ACOF2)

Format:
[<MjPri><MinPri=3><CANID>]<D1><NN hi><NN lo><EN hi><EN lo>
<data1><data2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2

Indicates an 'OFF' event using the full event number of 4 bytes with two additional data bytes.

210. D2 Teach an event in learn mode (EVLRN)

Format:
[<MjPri><MinPri=3><CANID>]<D2><NN hi><NN lo><EN hi><EN lo>
<EV#><EV val>
Sent by a configuration tool to a node in learn mode to teach it an event. Also teaches it the associated event variables (EVs) by the EV index (EV#). This command is repeated for each EV required.

211. D3 Response to a request for an EV value in a node in learn mode (EVANS)

Format:
[<MjPri><MinPri=3><CANID>]<D3><NN hi><NN lo><EN hi><EN lo>
<EV#><EV val>
A node response to a request from a configuration tool for the EVs associated with an event (REQEV). For multiple EVs, there will be one response per request.

212. D4 Accessory Response Event (ARON2)

Format:
[<MjPri><MinPri=3><CANID>]<D4><NN hi><NN lo><EN hi><EN lo>
<data1><data2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2

Indicates an 'ON' response event with two added data bytes. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

213. D5 Accessory Response Event (AROF2)

Format:
[<MjPri><MinPri=3><CANID>]<D5><NN hi><NN lo><EN hi><EN lo>
<data1><data2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2

Indicates an 'OFF' response event with two added data bytes. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

214. D6 Reserved

215. D7 Reserved

216. D8 Accessory Short ON (ASON2)

Format:
[<MjPri><MinPri=3><CANID>]<D8><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number
<**Dat5**> is the additional data byte 1
<**Dat6**> is additional data byte 2

Indicates an 'ON' event using the short event number of 2 LS bytes with two added data bytes.

217. D9 Accessory Short OFF (ASOF2)

Format:
[<MjPri><MinPri=3><CANID>]<D9><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number
<**Dat5**> is the additional data byte 1
<**Dat6**> is additional data byte 2

Indicates an 'OFF' event using the short event number of 2 LS bytes with two added data bytes.

218. DA Reserved

219. DB Reserved

220. DC Reserved

221. DD Accessory Short Response Event (ARSON2) with two data bytes

Format:
[<MjPri><MinPri=3><CANID>]<DD><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number
<**Dat5**> is the additional data byte 1
<**Dat6**> is the additional data byte 2

Indicates an 'ON' response event with two added data bytes.
A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

222. DE Accessory Short Response Event (ARSOF2) with two data bytes

Format:
[<MjPri><MinPri=3><CANID>]<DE><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number
<**Dat5**> is the additional data byte 1
<**Dat6**> is the additional data byte 2

Indicates an 'OFF' response event with two added data bytes.
A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

223. DF Extended op-code with 5 data bytes (EXTC5)

Format:
[<MjPri><MinPri=3><CANID>]<DF><Ext-OPC><byte1><byte2><byte3 >
<byte4><byte5>
Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

E0-FF - 7 data byte packets

[<MjPri><MinPri><CAN ID>]<OPC><Dat1>..<Dat7>

224. E0 Request 6-byte DCC Packet (*RDCC6*)

Format:
[<MjPri><MinPri=2><CANID>]<E0><REP><Byte0>..<Byte5>
<**Dat1(REP)**> is number of repetitions in sending the packet.
<**Dat2>..<Dat7>** 6 bytes of the DCC packet.
Allows a CAB or equivalent to request a 6 byte DCC packet to be sent to the track. The packet is sent <REP> times and is not refreshed on a regular basis.

225. E1 Engine report (*PLOC*)

Format:
[<MjPri><MinPri=2><CANID>]<E1><Session><AddrH><AddrL>
<Speed/Dir><Fn1><Fn2><Fn3>

<**Dat1**> Session for engine assigned by the command station. This session number is used in all referenced to the engine until it is released.
<**Dat2**> is the MS byte of the DCC address. For short addresses it is set to 0.
<**Dat3>** is the LS byte of the DCC address. If the engine is consisted, this is the consist address.
<**Dat4>** is the Speed/Direction value. Bit 7 is the direction bit and bits 0-6 are the speed value.
<**Dat5**> is the function byte F0 to F4
<**Dat6>** is the function byte F5 to F8
<**Dat7>** is the function byte F9 to F12

A report of an engine entry sent by the command station. Sent in response to QLOC or as an acknowledgement of acquiring an engine requested by a cab (RLOC or GLOC).

226. E2 Response to request for node name string (NAME)

Format:
[<MjPri><MinPri=3><CANID>]<E2><char1><char2><char3><char4>
<char5><char6><char7>

A node response while in ‘setup’ mode for its name string. Reply to (RQMN). The string for the module type is returned in char1 to char7, space filled to 7 bytes. The Module Name prefix , currently either CAN or ETH, depends on the Interface Protocol parameter, it is not included in the response, see section 7.2.3 for the definition of the parameters.

227. E3 Command Station status report (*STAT*)

Format:
[<MjPri><MinPri=2><CANID>]<E3><NN hi><NN lo><CS num><flags>
<Major rev><Minor rev><Build no.>
<NN hi> <NN lo> Gives node id of command station, so further info can be got from parameters or interrogating NVs
<CS num> For future expansion - set to zero at present
<flags> Flags as defined below
<Major rev> Major revision number
<Minor rev> Minor revision letter
<Build no.> Build number, always 0 for a released version.
<flags> is status defined by the bits below.

bits:
0 - Hardware Error (self test)
1 - Track Error
2 - Track On/ Off
3 - Bus On/ Halted
4 - EM. Stop all performed
5 - Reset done
6 - Service mode (programming) On/ Off
7 – reserved
Sent by the command station in response to RSTAT.

228. E4 Reserved

229. E5 Reserved

230. E6 Reserved

231. E7 Reserved

232. E8 Reserved for streaming protocol

233. E9 Reserved for streaming protocol

234. EA Reserved for streaming protocol

235. EB Reserved for streaming protocol

236. EC Reserved for streaming protocol

237. ED Reserved for streaming protocol

238. EE Reserved for streaming protocol

239. EF Response to request for node parameters (PARAMS)

Format:
[<MjPri><MinPri=3><CANID>]<EF><PARA 1><PARA 2><PARA 3>
<PARA 4><PARA 5><PARA 6><PARA 7>
A node response while in ‘setup’ mode for its parameter string. Reply to (RQNP)

240. F0 Accessory ON (ACON3)

Format:
[<MjPri><MinPri=3><CANID>]<F0><NN hi><NN lo><EN hi><EN lo>
<data1><data2><data3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2
<**Dat7**> is additional data byte 3

Indicates an 'ON' event using the full event number of 4 bytes with three additional data bytes.

241. F1 Accessory OFF (ACOF3)

Format:
[<MjPri><MinPri=3><CANID>]<F1><NN hi><NN lo><EN hi><EN lo>
<data1><data2><data3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2
<**Dat7**> is additional data byte 3

Indicates an 'OFF' event using the full event number of 4 bytes with three additional data bytes.

242. F2 Response to request to read node events (ENRSP)

Format:
[<MjPri><MinPri=3><CANID>]<F2><NN hi><NN lo>
<EN3><EN2><EN1><EN0><EN#>

Where the NN is that of the sending node. EN3 to EN0 are the four bytes of the stored event. EN# is the index of the event within the sending node. This is a response to either 57 (NERD) or 72 (NENRD)

243. F3 Accessory Response Event (ARON3)

Format:
[<MjPri><MinPri=3><CANID>]<F3><NN hi><NN lo><EN hi><EN lo>
<data1><data2><data3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2
<**Dat7**> is additional data byte 3

Indicates an 'ON' response event with three added data bytes. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

244. F4 Accessory Response Event (AROF3)

Format:
[<MjPri><MinPri=3><CANID>]<F4><NN hi><NN lo><EN hi><EN lo>
<data1><data2><data3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the event number
<**Dat4>** is the low byte of the event number
<**Dat5**> is an additional data byte 1
<**Dat6**> is additional data byte 2
<**Dat7**> is additional data byte 3

Indicates an 'OFF' response event with three added data bytes. A response event is a reply to a status request (AREQ) without producing an ON or OFF event.

245. F5 Teach an event in learn mode using event indexing (EVLRNI)

Format:
[<MjPri><MinPri=3><CANID>]<F5><NN hi><NN lo><EN hi><EN lo>
<EN#><EV#><EV val>
Sent by a configuration tool to a node in learn mode to teach it an event. The event index must be known. Also teaches it the associated event variables.(EVs). This command is repeated for each EV required.

246. F6 Accessory node data event (ACDAT)

Format:
[<MjPri><MinPri=3><CANID>]<F6><NN hi><NNlo>
<data1><data2><data3><data4><data5>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the first node data byte
<**Dat4>** is the second node data byte
<**Dat5**> is the third node data byte
<**Dat6**> is the fourth node data byte
<**Dat7**> is the fifth node data byte

Indicates an event from this node with 5 bytes of data.
For example, this can be used to send the 40 bits of an RFID tag. There is no event number in order to allow space for 5 bytes of data in the packet, so there can only be one data event per node.

247. F7 Accessory node data Response (ARDAT)

Format:
[<MjPri><MinPri=3><CANID>]<F7><NN hi><NN lo>
<data1><data2><data3><data4><data5>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the first node data byte
<**Dat4>** is the second node data byte
<**Dat5**> is the third node data byte
<**Dat6**> is the fourth node data byte
<**Dat7**> is the fifth node data byte

Indicates a node data response. A response event is a reply to a status request (RQDAT) without producing a new data event.

248. F8 Accessory Short ON (ASON3)

Format:
[<MjPri><MinPri=3><CANID>]<F8><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2><data 3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number
<**Dat5**> is the additional data byte 1
<**Dat6**> is additional data byte 2
<**Dat7**> is additional data byte 3

Indicates an 'ON' event using the short event number of 2 LS bytes with three added data bytes.

249. F9 Accessory Short OFF (ASOF3)

Format:
[<MjPri><MinPri=3><CANID>]<F9><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2><data 3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the Device Number
<**Dat4>** is the low byte of the Device Number
<**Dat5**> is the additional data byte 1
<**Dat6**> is additional data byte 2
<**Dat7**> is additional data byte 3

Indicates an 'OFF' event using the short event number of 2 LS bytes with three added data bytes.

250. FA Device data event (short mode) (DDES)

Format:
[<MjPri><MinPri=3><CANID>]<FA><DN hi><DN lo>
<data1><data2><data3><data4><data5>
<**Dat1>** is the high byte of the device number
<**Dat2>** is the low byte of the device number
<**Dat3>** is the first device data byte
<**Dat4>** is the second device data byte
<**Dat5**> is the third device data byte
<**Dat6**> is the fourth device data byte
<**Dat7**> is the fifth device data byte

Function is the same as F6 but uses device addressing so can relate data to a device attached to a node. e.g. one of several RFID readers attached to a single node.

251. FB Device data response (short mode) (DDRS)

Format:
[<MjPri><MinPri=3><CANID>]<FB><DN hi><DN lo>
<data1><data2><data3><data4><data5>
<**Dat1>** is the high byte of the device number
<**Dat2>** is the low byte of the device number
<**Dat3>** is the first device data byte
<**Dat4>** is the second device data byte
<**Dat5**> is the third device data byte
<**Dat6**> is the fourth device data byte
<**Dat7**> is the fifth device data byte

The response to a request for data from a device. (0x5B)

252. FC Reserved

253. FD Accessory Short Response Event (ARSON3)

Format:
[<MjPri><MinPri=3><CANID>]<FD><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2><data 3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number
<**Dat5**> is the additional data byte 1
<**Dat6**> is the additional data byte 2
<**Dat7**> is the additional data byte 3

Indicates an 'ON' response event with with three added data bytes. A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

254. FE Accessory Short Response Event (ARSOF3)

Format:
[<MjPri><MinPri=3><CANID>]<FE><NN hi><NN lo><DN hi><DN lo>
<data 1><data 2><data 3>
<**Dat1>** is the high byte of the node number
<**Dat2>** is the low byte of the node number
<**Dat3>** is the high byte of the device number
<**Dat4>** is the low byte of the device number
<**Dat5**> is the additional data byte 1
<**Dat6**> is the additional data byte 2
<**Dat7**> is the additional data byte 3

Indicates an 'OFF' response event with with three added data bytes. A response event is a reply to a status request (ASRQ) without producing an ON or OFF event.

255. FF Extended op-code with 6 data bytes (EXTC6)

Format:
[<MjPri><MinPri=3><CANID>]<FF><Ext-OPC><byte1><byte2><byte3 >
<byte4><byte5><byte6>
Used if the basic set of 32 OPCs is not enough. Allows an additional 256 OPCs

12.4. Accessory Module – Error codes

Error codes for CBUS accessory modules, these error codes are returned by OPC CMDERR 0x6F

1	Command Not Supported	- see note 1.
2	Not In Learn Mode	
3	Not in Setup Mode	- see note 1
4	Too Many Events	
5	Reserved	
6	Invalid Event variable index	
7	Invalid Event	
8	Reserved	- see note 2
9	Invalid Parameter Index	
10	Invalid Node Variable Index	
11	Invalid Event Variable Value	
12	Invalid Node Variable Value	

Note 1: Accessory modules do not return this error
Note 2: Currently used by code that processes OPC REVAL 0x9C but this code should be updated to use codes 6 & 7.

12.5. DCC – Error codes

These codes are returned by OPC ERR 0x63

1	Loco stack full	First two bytes are loco address, third is error number.
2	Loco address taken	- First two byes are loco address, third is error number.
3	Session not present	- First byte session id, second byte zero, third is error number.
4	Consist empty	- First byte consist id, second byte zero, third is error number.
5	Loco not found	- First byte session id, second byte zero,, third is error number.
6	CAN bus error	- Two data bytes set to zero (not used), third is error number. - This would be sent out in the unlikely event that the command station buffers overflow.
7	Invalid request	- First two bytes are loco address, third is error number. Indicates an invalid or inconsistent request. For example, a GLOC request with both steal and share flags set.
8	Session cancelled	- First byte session id, second byte zero, third is error numb Sent to a cab to cancel the session when another cab is stealing that session.

Appendix A – Fixed Node Numbers and CAN_IDs

The following fixed Node Numbers and CAN_IDs have been defined

Module	Fixed CAN ID		Fixed Node Number	
	Decimal	Hex	Decimal	Hex
CANCMD	114	0x72	65534	0xFFFE
CANUSB	124	0x7C	65532	0xFFFC
CANEther	125	0x7D		
CABS			65535	0xFFFF

Note that CANRS used to be in this list with a fixed CAN ID, but that firmware has never used it for anything, just passing on the received CAN ID, so it has been removed from this list.

CAN IDs 96 to 106 (0x64 to 0x7D) are used by modules with a fixed CAN ID. See the notes in section 6.2 about CAN IDs and self enumeration.

Node numbrers 0xFFF0 (65520) upwards are used by modules with a fixed default node number.

See the notes in section 7.2.2 about node numbers.