1. Contents

2. Overview 1
3. General 1
4. File naming convention 3
5. File Format: 4
6. top-level section 5
7. nodeParameters section 6
8. nodeVariables & eventVariables sections 6
10. Overloaded labels 13
11. Logic elements 15
12. Channel Names 17

2. Overview

The CBUS protocol provides commands to allow the configuration of a module by a networked computer. These commands allow the read & write functionality of numbered variables, but doesn't describe what those variables do – as this is different across different modules
Thus, modifying a computer program to allow users to configure new modules has been a very time-consuming process, and typically not possible by the module developer
This document aims to provide a way to describe what these variables mean, in a form that can be digested by a computer program to provide a meaningful user interface, without changing the software programming, thus available to all module developers

In some firmware, the meaning of a variable may change depending on the value of another variable, so there are features to support this. To support dynamically changing the meaning of variables, there are two features, a 'logic element' which controls if an element is displayed (or not) and 'overloaded' labels which is supported by suitable properties

This format is intended to describe what the module variables mean, not dictate how the user interface is implemented. However, pragmatically some things are best described in user interface terms

3. General

CBUS Modules are identified by manufacturer/model numbers, which are stored in the firmware. So the module actually refers to the firmware, and not the physical board the firmware resides on, although most of the firmware has been written for a specific type of board. There is, however, some firmware that will run on different hardware, and will report the same module

identification irrespective of which hardware they are running on. Equally, there is specific hardware that have had different variants of firmware written for them, so such hardware will report a different module ID depending which variant of firmware is loaded

In this file the module is described using key-value pairs, the key identifies the data, and the value is the data content, which can be various types of literal (fixed) data, such as string, numeric or an array to 'nest' further key-value pairs
Key-value pairs can be represented in many data formats, with JSON being the preferred format widely supported by many languages

Camel case is the preferred naming format for keys, unless specified otherwise
Camel case is the practice of starting each word with a capital letter, except for the first word.

There is no direct support for comments, but a key-value pair can be used where necessary
E.g. "comment":"comment added for clarity"

4. File naming convention

The filename is generated from the module name, the manufacturer/module identification and the firmware version. Leading with the module name makes the files a little easier to sort by eye
Kebab case formatting (- sign separator) is used to separate the 3 items
Note the firmware version is kept in the same format as used in firmware documentation, i.e. number followed by character

AAAAAAA-BBCC-DE<Options>

Where
AAAAAAA : variable length module name
BB : Manufacturer ID in hexadecimal - two hex digits
CC : Module ID in hexadecimal - two hex digits
D : Firmware major version in decimal - 1 to 3 digits
E : Firmware minor version in single ascii character
<Options> : not mandatory - described below
Example (without options):
CANACC4-A501-2q.json

Module name

The module name is registered against the manufacturer ID & Module ID in CbusDefs, and as the manufacturer & module ID is also present, is just included for human readability

Manufacturer ID & Module ID

The manufacturer & module ID provide the unique identity of the module
For personal development of modules, it is recommended that the NMRA Manufacturer ID of '13' (development) is used, with a module ID & module name of personal choice, thus avoiding the need to have a module added to 'CbusDefs' – avoiding conflict with other modules would then be the responsibility of the user of that module

Options

The use of different microprocessor devices in the same module allows for different capabilities of that module. To reflect this, an optional suffix can be added to the file name to identify the processor type.
The option format for processor type is two hyphens, followed by a capital P then the 'processor type number' as defined by "CbusDefs" – e.g. **"—P13"**
For the majority of modules, that do not support multiple devices, or that have the same capabilities irrespective of processor type, then this option would be omitted – i.e. the option would only be used in a limited number of cases

For a recent version of CANMIO-Universal firmware, one example would be

CANMIO-A520-3d—P13.json

Where the –P13 indicates it's for a module with the P18F25K80 microprocessor

Why use a 'double hyphen' ?

As we cannot dictate that a hyphen cannot be in the module name, then a hyphen may appear in that, so in order to be able to identify the optional segment, the 'double hyphen' is used, as this is most unlikely to appear in the name

Using the filename

The combination of Manufacturer ID & Module ID uniquely identifies the module, so it is pragmatic to match the filename without the 'Module Name' portion

This means that the match can be performed from the data provided from 'Node Parameters', read from the module. The module NAME can only be read from a CBUS module when in setup mode, which isn't possible from an already configured module

It is recommended that a file match for a stored module descriptor file is initially performed with the processor option, falling back to matching the filename without the processor option

The version numbers from modules are not consistent in using either upper or lower case, and it would be pragmatic to not assume the filenames were consistent in this regard either, so the matching should be implemented to be case insensitive for both

5. File Format:

Style

Whilst not mandatory, for readability it is strongly recommended that the title & subtitle properties are the first in any element, followed by the type property.

```
{
  "displayTitle": "Time delay between response messages",
  "displaySubTitle": "1 millisecond steps",
  "type": "NodeVariableSlider",
  "nodeVariableIndex": 5,
  "displayUnits": "milliseconds"
},
```

Unfortunately, early examples do not follow this convention

6. top-level section

The following table lists just the top level elements, more information about each element is in its own section

Element	Brief description	Could/should/must exist
channelNames	Names for the channels this module has	could
eventVariableInformation	Displayable Text about event variables	could
eventVariables	Collection of EV descriptors	should
moduleDescriptorFilename	Filename of this descriptor	could
moduleName	Registered module name	should
nodeParameters	Collection of Node Parameters	could
nodeVariableInformation	Displayable Text about node variables	could
nodeVariables	Collection of NV descriptors	should
numberOfChannels	Number of channels this module has	could
NVsetNeedsLearnMode	Specific modules need this	Could
timestamp	Last commit time	should

moduleDescriptorFilename

This property is the filename of this descriptor. This is an important property that is used to identify the descriptor when the content of the file is being as a data object programmatically (i.e. not as a file with a filename). This could be added dynamically by any consumer of the file when first read from storage if it's needed
"moduleDescriptorFilename": "CANPAN-A51D-4c.json"

moduleName

This optional property in the root section is the name registered against the manufacturer ID & Module ID for this specific module. The value returned from the CBUS command NAME is typically a subset of this module name due to data restrictions
"moduleName":"CANACC5"

timestamp

A timestamp that shows when a module descriptor file was last committed. This is used to check if a user provided file is older than a system file. If the user provided file is older then it may need to be updated, or be removed so that the newer system file is used. A user provided file takes precedence over a system file with the same name and can thus hide updates in the system file.

The format of the timestamp string is <year><month><day><hour><minute> without separators. Each part is zero-padded. The timestamp shows the time in UTC.

NVsetNeedsLearnMode

One family of firmware based on original CANSERVO8 code needs to be put into 'learn' mode before node variables can be programmed. Setting the NVsetNeedsLearnMode key to a value of true in the root section will indicate if this is required for this specific module. The processing application is expected to assume false if this property is not present, so it's only required if it needs to be set to true

"NVsetNeedsLearnMode": true

7. nodeParameters section

This optinal section contains values expected to be returned from the module when requesting the Node Parameters. This data has multiple uses, e.g can be used if the module is not online, or can be used to verify the output of the module against these expected values (conformance testing)

The number of parameters can vary depending on module and firmware version, but typically is at least 20, and the meaning is defined in the CBUS standard

The name field is optional, as currently the parameters have fixed meaning, but having the name field does allow for informing the application about the meaning of new parameters that have not yet been added to the standard

In the case of any discrepancy between this list and the parameters actually read from the module, the application will decide which has precedence

```
"nodeParameters": {
   "1": {
      "value": 165,
      "name": "Manufacturer's Id"
   },
      <more content>
}
```

8. nodeVariables & eventVariables sections

These sections are the descriptors for node & event variables. Both of these have types and properties that work in the same way.

The major difference between the two is that there is only one instance of the node variables for each module, where there can be multiple instances of the event variables, so the actual types reflect that difference, but share the same properties

A group type element is defined to allow the grouping of elements using an array

For many modules, the meaning of certain variables change depending on the value of another variable. To cater for this, a visibilityLogic property has been created. This allows more than one descriptor for a single variable to be created, but controls which of these descriptors actually gets displayed by the result 'visibilityLogic' element (only display is logic returns true), this should be supported on all types
Another option is Overloaded Labels, which allows different labels to be presented depending on another variable, all within a single descriptor. Note this is only available on certain types

Types

The type property indicates what the variable represents and how should be handled
Note not all types duplicated for both, as created on an 'as needed' basis

EventVariableBitArray NodeVariableBitArray	Represents an 8 bit node variable where each bit can be selected independently - also known as flags, bitfield or multi-select Uses bitCollection to define the bits & their labels Uses displayTitle, displaySubTitle Supports Overloaded labels
EventVariableBitSingle NodeVariableBitSingle	Represent a single bit in a node variable Uses the bitPosition property to identify which bit Uses displayTitle, displaySubTitle
NodeVariableDual	Represents a two byte variable as a simple numeric input value Uses displayTitle, displaySubTitle Be aware which node variable Index is the most significant byte nodeVariableIndexHigh should be the index of the most significant byte, which may not be the highest index number
EventVariableGroup NodeVariableGroup	Allows a collection of types to be logically grouped together, differs from Tabs in that multiple groups can be visible at the same time, and expected to have less content Uses the groupItems property, which can contain any of the other types including the 'Tabs' types Doesn't use any other properties
EventVariableNumber NodeVariableNumber	Represents a variable as a simple numeric input value Option to use min & max to limit user input Option to use startBit & endBit to use a subset of the bits in a variable Option to use displayScale, displayUnits and displayOffset to adjust displayed values Uses displayTitle, displaySubTitle
EventVariableSlider NodeVariableSlider	Represents a variable as a slider control Option to use displayScale, displayUnits and displayOffset to adjust displayed values Uses displayTitle, displaySubTitle

	Option to use min & max to limit user input Option to use startBit & endBit to use a subset of the bits in a variable NodeVariableSlider supports the option to use outputOnWrite to show output is set immediately on a write (can implement a 'test' feature)
EventVariableSelect NodeVariableSelect	Represents a control to select a single value from the array of options Option to use bitMask to define a subset of the bits to use Option to use displayScale to adjust displayed values Option to use displayUnits to display units of measure Uses displayTitle, displaySubTitle Supports Overloaded labels
EventVariableTabs NodeVariableTabs	Defines the logical grouping of a set of variables, differs from 'groups' in that only the contents of one tab (tabPanel) is visible at a time, typically the full width of the display area, and would typically have more content than a group Uses the tabPanels property to define a set of tabs and the content of the associated tab panels, the content is any of the other types including the 'group' types. Doesn't use any other properties

Properties for nodeVariable & eventVariable

property (key)	type	requirement	default
Type	string	mandatory	Not Applicable
nodeVariableIndex eventVariableIndex	numeric	mandatory	Not Applicable
nodeVariableIndexHigh nodeVariableIndexLow	numeric	Mandatory for NodeVariableDual	Not Applicable
bit	numeric	Mandatory for some types	Not Applicable
bitCollection	array	Mandatory for some types	Not Applicable
Bitmask	numeric	optional	255
linkedVariables	array	optional	Not Applicable
Min	numeric	optional	0
Max	numeric	optional	maximum size of the variable type
startBit	numeric	optional	0
endBit	numeric	optional	8
groupItems	array	Mandatory for some types	Not Applicable
displayTitle	string	Mandatory for some types	Not Applicable
displaySubTitle	string	optional	Not Applicable
displayScale	numeric	optional	1
displayUnits	string	optional	blank
displayOffset	numeric	optional	0
Options	array	optional	Not Applicable
outputOnWrite	boolean	optional	false
tabPanels	array	Mandatory for some types	Not Applicable
visibilityLogic	Logic	optional	Not Applicable

	element		

bit

The bit position within the variable, 0 to 7, of an individual bit

bitCollection

An array of bit positions and associated labels used to define a collection of a variable number of bits and their labels used in the BitArray types. bitPositions start from 0
Each array entry of the form {“bitPosition”: 1, "label": "bit description"}

bitMask

A bit value of 1 in the bitMask indicates that the corresponding bit position in the variable should be modified, a value of 0 shows the corresponding bit position in the variable should keep its original value. This allows a type to modify just part of a variable, and leave the remainder for another type to modify. See options description for an example of how it can be used

linkedVariables

This defines an array of other variables that may change when this variable is changed. This allows these other variables to be re-read from the module when a change is made
An example is CANMIO-Universal firmware, which when the channel type is changed (e.g. ‘input’ to ‘servo’), the other variables associated with that channel are reset to default by the module itself, so need to be read from the module again. This element can contain both node variables (NV) and event variables (EV), or just one of either

```
“linkedVariables”:{
  “EV”: [5,6],
  “NV”: [11,12,13]
}
```

min/max

This pair usually relates to the raw value in the node variable, not the display value, unless stated otherwise in the type description above

startBit/endBit

Describes the starting and ending bits of a value that doesn’t use all 8 bits of a variable. Typically used to create a bit mask to use to ensure that unused bits are not modified when this value is updated

displayScale & displayUnits

For numeric values, this pair allows the variable to be displayed in a ‘friendly’ fashion, e.g. a time delay in 100mS intervals would have a displayScaling of 100 and a displayUnits of ‘mS’ These do not affect the underlying ‘raw’ variable

displayOffset

Used in special circumstances where the value the variable represents doesn’t start at 0. E.g. a time delay may have a minimum of 500mS (i.e. the variable value of 0 represents 500mS), but intervals of 100mS could have a display offset of 500. Another example would be to use an offset of 1 to display channel numbers 1 to 8, that’s stored in 3 bits which have a range of 0 to 7. Can also be used to display negative starting values, whilst keeping the raw value unsigned, e.g. using an offset of -100 to display -100 to +100 with the raw value in the variable being 0 to 200 - however, probably less useful in this application

displayTitle

The main description of the item

displaySubTitle

An optional element that can be used to add further information about the item - e.g. “Range 50 to 25500 mS”

groupItems

An array used by the NodeVariableGroup type to logically group other types together, e.g. to group more than one node variable to a single channel

options

Array of labels with values to be used in the NodeVariableSelect type.
Each array entry of the form {"label": "Options 1", "value": 0}
The value field maps onto the bits in the variable, for example, if the top 2 bits are used (bits 6 & 7), then the array will take the form

```
{"label": "event sent at ON end", "value": 0},          – bits 6 & 7 clear
{"label": "event sent when at OFF end", "value": 64},   – bit 6 set, 7 clear
{"label": "event sent at mid travel", "value": 128},    – bit 6 clear, bit 7 set
{"label": "Start of Day (SoD) event", "value": 192}     – bits 6 & 7 set
```

The bitMask option can be used to limit modifications to the specific bits, in this case a value of 192 would be used (bits 6 & 7 set to only allow those to be modified)

outputOnWrite

If true, this indicates that when this variable is written to, the associated output immediately takes on this new value.
In some modules this is seen in the servo position variables, and has been used to provide a ‘test’ feature by re-writing the current variable value, e.g. to move the servo to the ‘on’ position
Currently only supported in the NodeVariableSlider type.

tabPanels

An array used by the NodeVariableTabs & EventVariableTabs types to logically group other types together in tabbed panels
Each entry in the array contains the displayTitle of the tab, and a further array of items that form the content of the tab panel

```
{ "displayTitle": "Tab number one",
  "items": [
    {
      "type": "EventVariableNumber",
      "eventVariableIndex": 1,
      "displayTitle": "Output 1-1"
    }
  ]
}
```

visibilityLogic

This optional property uses a logic element (see below for more detail) to only display the descriptor if the logic returns true.
This is typically used where the meaning of a variable changes depending on another variable - e.g. on some modules, event variables change meaning (or not used) if the event is marked as a ‘produced’ event, or ‘consumed’ event
Having this “visibilityLogic” property allows multiple descriptors to be defined for the same variable, but then select which one gets displayed depending on the result of the logic element

10. Overloaded labels

In more complex modules, the same variable has different meaning depending on another variable - an example being the following partial extract for a configuration for EV#1

Produced(EV#1)									
Happening id		I/O	TYPE						
Hex	Decimal		INPUT	OUTPUT	SERVO	BOUNCE	MULTI	ANALOGUE	MAGNET
0	0	Global	No Happening is mapped to the event.						
1	1		Produced startup event						
2	2								
3	3								
4	4								
5	5								
6	6								
7	7								
8	8	1	Changed	Changed	Reached OFF	Changed	AT1		
9	9		TWO_ON		Reached MID		AT2		
A	10				Reached ON		AT3		
B	11						AT4		
C	12	2	Changed	Changed	Reached OFF	Changed	AT1		
D	13		TWO_ON		Reached MID		AT2		
E	14				Reached ON		AT3		
F	15						AT4		

Where the variable is an encoding (i.e. each value has a different meaning), then only the label needs to be changed to describe a different meaning for that value
This isn't suitable for variables that aren't encodings, hence only certain types support it

The following element describes how to 'overload' the label in types that support it
"overload"{"nv",labels["value":0, "label": "first label", "value":1, "label": "2nd label",.....]

For example, the following shows how the table above is represented

```
{
 "type": "EventVariableSelect",
 "eventVariableIndex": 1,
 "displayTitle": "Produced event",
 "displaySubTitle": "EV1",
 "options": [
  {"value": 0, "label": "no event (0)"},
  {"value": 1, "label": "Startup event (1)"},
  {"value": 8, "overload":{"nv": "16", "labels": [
       {"value": 0, "label": "CH1 - Input Changed"},
       {"value": 1, "label": "CH1 - Input Changed"},
       {"value": 2, "label": "CH1 - Reached OFF"},
       {"value": 3, "label": "CH1 - Input Changed"},
       {"value": 4, "label": "CH1 - AT1"}
     ]
   }
  },
  {"value": 9, "overload":{"nv": "16", "labels": [
       {"value": 0, "label": "CH1 - TWO_ON"},
       {"value": 2, "label": "CH1 - Reached MID"},
       {"value": 4, "label": "CH1 - AT2"}
     ]
```

```
        }
      },
...
```

11. Logic elements

A logic element allows simple logic to be embedded into the descriptor
The logic follows the syntax
“Named Property”:<expression>
The logic expression returns a true or false condition when evaluated for the named property
It is now recommended that jsonLogic is used to evaluated the expression, but compatibility with earlier versions is retained – these are both described below

The only property currently defined is “visibilityLogic”, which if added to an element, will decide if the element is displayed depending if the expression returns true or false
This then allows the application to show or hide an element depending on the condition of other variables, using this logic. There can thus be multiple elements for the same variable, and the logic used to determine which element to display

jsonLogic expressions

jsonLogic is implemented in a library that supports many different programming languages
https://jsonlogic.com/
There are numerous benefits to using this, including logical combinations (and, or etc..) and a much wider level of logical operations
The basic syntax is
{"operator" : ["values" ...]},
but please refer to the documentation for the actual supported operations, and the syntax for each operation

However, by itself, the library is not aware of external data, such as the module variables, so the ‘add operation’ feature needs to be used to add access to this data.
The custom operations needed are
EV, EVbit
NP, NPbit
NV, NVbit
Where the operations return the value of the specified variable that’s been read from the module
And the syntax for EV is
{“EV”, <index number>}
EVbit is
{“EVbit”, [<index number>, <bit number>]}
And the same for NP and NV

For the application implementing these, adding operations to the library is very easy, and good examples exist in the jsonLogic documentation. It will be specific to that application, as its giving access to however that application stores it’s module data

In order to differentiate these expressions from the earlier implementation, the jsonLogic expression is wrapped in a “JLL” element (json logic literal)
Here’s a simple example that tests if NV1 is equal to 9

```
"visibilityLogic":{ "JLL":{ "==" : [ {"NV" : [1]}, 9] } }
```

Here’s an example of ‘nested’ or ‘combinational’ logic that will only present true if either (or) bits 0 and 1 of EV3 are equal (‘==’) to 0 (false)

```
"visibilityLogic":{"JLL":{ "or" : [
            { "==" : [ {"EVbit" : [3, 0]}, false ] },
            { "==" : [ {"EVbit" : [3, 1]}, false ] }
          ]}
        },
```

It’s also possible to use these new operations as parameters to each other, e.g.

```
"visibilityLogic":{“JLL”: { "in" : [ { "NV": {"EV" : 1}}, [5,6,7] ] } }
```

Earlier expression syntax

The use of this is now deprecated, in favour of jsonLogic (above), but should be supported for backwards compatibility
The logic follows the syntax
“Named Property”:{ “argument”, “condition”}
The intent is that the “argument” always evaluates to a simple numeric value, and the condition just needs to compare two simple values
For example, in the following property, “visibilityLogic”, the result of the logic element is used to control if a variable is displayed or not
And in this case, the variable would only be displayed if the value of ev3 bit 7 is equal to 1

```
 "visibilityLogic":{
  "evBit": {“index”:3, “bit”: 7}
  "equals": 1
 },
```

Logic conditions

condition	description
“equals”:<value>	Equal to, with numerical value, will return true or false
“in”:[<value>,...]	Equal to any value in the array, will return true or false

Logic arguments

item	description

“evBit”: {
“index”:<ev index>,
“bit”: <0 to 7>
}
	Describes a single event variable bit with numerical values for event index and bit position (0 to 7). Will evaluate to 0 or 1

“nv”:<nv index>	Describes a node variable by it’s index number, will evaluate to a number 0 to 255

12. Channel Names

One of the characteristics of the majority of modules is that they support multiple input/output (I/O) ‘channels’
By convention, these channels are numbered, but it can become a problem remembering what a particular channel (e.g. node 300, channel 2) actually connects to, especially on large networks
The ‘channel names’ feature allows a management tool to easily replace channel ‘tokens’ in the file with user supplied names, that give more meaning (e.g. node 300, channel 2 could be named ‘turnout 1’)
If the user doesn’t choose to add a channel name, the management tool will supply a ‘system’ default in its place – this is expected to be ‘channel xx’ where xx is the channel number

There are three aspects to this feature

numberOfChannels’

A ‘numberOfChannels’ element to inform the management tool how many channels this module has, so it can offer the user the correct number of channels in the list to be edited
Note that this can be omitted if the optional channel names array is used for all the channels – see below

The token

The token itself, which is of the form ${channelxx} where xx is the channel number
This token will then be replaced by the user entered name for that channel number, where ever it is placed in the MDF – this is expected to be used to replace fixed text fields.
The management tool should accept either case for the text ‘channel’, and also allow whitespace between ‘channel’ and the actual channel number

channelNames array

An optional ‘channelName’ array, which will override the ‘system’ default name for the channel that is shown if there is no user name

This is useful where the channel is a fixed I/O type, so the default can reflect that (e.g. LEDxx or Switchxx) rather than the system generic ‘channelxx’

It is not necessary to provide a name for every channel here, e.g. you could just supply a name for channel 1 and channel 7
Note the management tool should find the highest channel number in the channelname array, and then use either the ‘numberOfChannels’ or the highest number from this array, whichever is greater, to determine how many channel names can be edited.

The order of precedence for the channel name would then be

1. User entered channel name – stored for that layout
2. MDF supplied channel name (e.g. switch XX, LED XX, Servo XX etc..)
3. System supplied default channel name (channel xx)

The starting fragment of an MDF is shown below, which shows the placement of the ‘numberOfChannels’, ‘channelNames’ & the first instance of a token - ${channel1}, which is used in the displayTitle element of a ‘NodeVariableSelect’ control

```
{
 "timestamp": "202505290839",
 "moduleName": "CAN4IN4OUT",
 “numberOfChannels”:8,
 "channelNames": {
  "1" : "Switch 1",
  "2" : "Switch 2",
  "3" : "Switch 3",
  "4" : "Switch 4",
  "5" : "LED 1",
  "6" : "LED 2",
  "7" : "LED 3",
  "8" : "LED 4"
 },
 "nodeVariables": [
  {
   "type": "NodeVariableSelect",
   "nodeVariableIndex": 1,
   "displayTitle": "${channel1}",
   "options": [
    {
```