NUCAN ZERO-TO-HERO

NUCAN is the NU Teams standard use medium for CAN BUS, featuring a DBC file and C++ scripts which generate a library of short cuts to allow for easy and efficient use of CAN BUS within code.

Instead of writing the information of each CAN BUS signal, NUCAN allows for a message to be sent with a single line of code. Setting up the CAN BUS code is also easier and NUCAN comes with some added functionality like heartbeat, internal CPU temperature sensing on Teensys and WATCHDOGS.

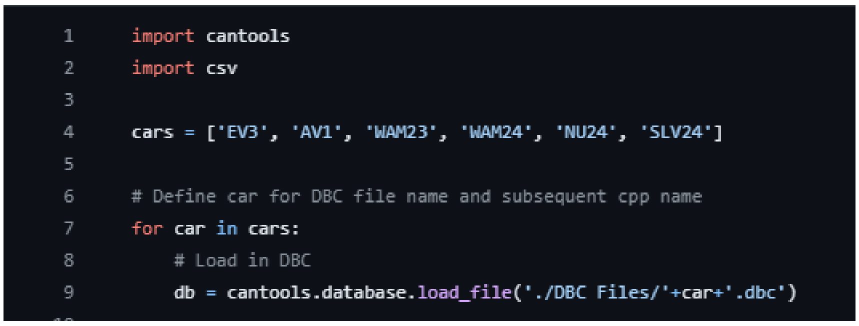
The WATCHDOG is built into NUCAN and does not need to be turned on, it will monitor the Teensy's heartbeat signal and reboot it if there is any problem. This means a heartbeat signal is needed for NUCAN to work.

The NUCAN GitHub:

The NUCAN GitHub repository is where all of NUCAN is stored for all teams.

There is a folder of all the DBC files as well as individual libraries for each DBC file within the folder.

The build_cpp_from_dbc.py file is a script which turns the DBC file, which humans can easily interact with, into C++ code that Arduino IDE uses. Whenever a new DBC file is added or removed this file needs to be updated. **Line 4 is the only line that needs to updated to represent the DBC files within the DBC Files folder.**



```
1   import cantools
2   import csv
3
4   cars = ['EV3', 'AV1', 'WAM23', 'WAM24', 'NU24', 'SLV24']
5
6   # Define car for DBC file name and subsequent cpp name
7   for car in cars:
8       # Load in DBC
9       db = cantools.database.load_file('./DBC Files/'+car+'.dbc')
```

Figure 1 build_cpp_from_dbc.py Script

Whenever this repository is 'pulled', the libraries which aren't being used by the Arduino code need to be deleted to avoid error when verifying or uploading the code to a teensy. An example of this is when using the NU24_CAN.h file, all other DBC created .h and .cpp files, e.g. WAM23.cpp, WAM23.h must be removed from the user's local device EXCEPT InternalTemperature, Watchdog_t4 and NUCAN.

HOWEVER, DO NOT push these deleted files as changes back to GitHub if changes to a DBC file are made. This will remove all the other files from the GitHub.

Whenever a change is made to a DBC file, NUCAN needs to be pushed to update the C++ files, then pulled. It may take a second for script to run fully and update the C++ files.

Creating and editing a DBC file:

To create a new DBC file or edit an existing one, either copy the format of an older DBC file (recommended) or create a new one from scratch using Kvaser Database Editor 3, which is a CAN message editor (DOWNLOAD: https://kvaser.com/single-download/?download_id=47183).

It is recommended to edit and create new DBC by copying what was already done before. There are many examples in GitHub within the DBC folder and it is important to get familiar with these before trying to edit and create new DBC files. Refer back to CAN BUS ZERO TO HERO to ensure that the DBC file format is understood.

Using the Kvaser Editor for DBC files:

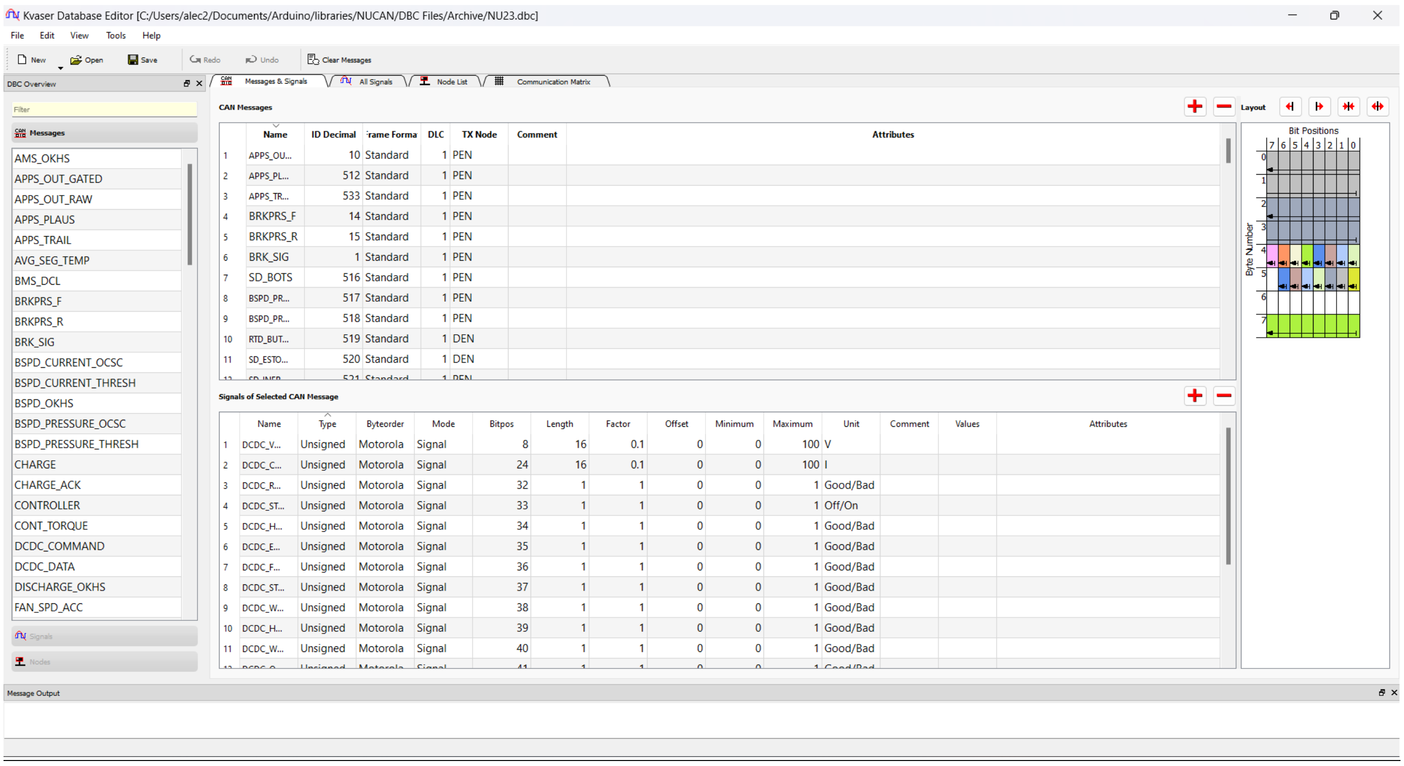
The Kvaser Editor will format everything, and it is recommended to look and understand an existing DBC file (revise back to CANBUS ZERO TO HERO) before making changes or creating a new file to minimise mistakes.

Once the format and interface is understood, create new CAN messages using the plus icon in the ‘Messages & Signals’ tab in the top box. To add signals within a single CAN message, use the bottom box (this is what is specific to each signal, so consideration needs to be made).

Ensure there is a heartbeat message for each node as the Teensy’s WATCHDOG needs a heartbeat signal to function.

Once the DBC file has been created or edited, it can then be saved/exported and save to the NUCAN DBC file folder and then added to the build_cpp_from_dbc.py file, before being pushed to GitHub and pulled shortly after.

Before using the DBC file and the generated libraries, double check everything was correctly created or use first in a very safe matter.


Kvaser Database Editor [C:/Users/alec2/Documents/Arduino/libraries/NUCAN/DBC Files/Archive/NU23.dbc]
File Edit View Tools Help
New Open Save Redo Undo Clear Messages
DBC Overview
Messages & Signals
All Signals
Node List
Communication Matrix
Filter
Messages
Signals
Nodes
Layout
Bit Positions
Byte Number
Message Output

AMS_OKHS
APPS_OUT_GATED
APPS_OUT_RAW
APPS_PLAUS
APPS_TRAIL
AVG_SEG_TEMP
BMS_DCL
BRKPRS_F
BRKPRS_R
BRK_SIG
BSPD_CURRENT_OCSC
BSPD_CURRENT_THRESH
BSPD_OKHS
BSPD_PRESSURE_OCSC
BSPD_PRESSURE_THRESH
CHARGE
CHARGE_ACK
CONTROLLER
CONT_TORQUE
DCDC_COMMAND
DCDC_DATA
DISCHARGE_OKHS
FAN_SPD_ACC

CAN Messages

	Name	ID Decimal	rame Forma	DLC	TX Node	Comment	Attributes
1	APPS_OU...	10	Standard	1	PEN		
2	APPS_PL...	512	Standard	1	PEN		
3	APPS_TR...	533	Standard	1	PEN		
4	BRKPRS_F	14	Standard	1	PEN		
5	BRKPRS_R	15	Standard	1	PEN		
6	BRK_SIG	1	Standard	1	PEN		
7	SD_BOTS	516	Standard	1	PEN		
8	BSPD_PR...	517	Standard	1	PEN		
9	BSPD_PR...	518	Standard	1	PEN		
10	RTD_BUT...	519	Standard	1	DEN		
11	SD_ESTO...	520	Standard	1	DEN		

Signals of Selected CAN Message

	Name	Type	Byteorder	Mode	Bitpos	Length	Factor	Offset	Minimum	Maximum	Unit	Comment	Values	Attributes
1	DCDC_V...	Unsigned	Motorola	Signal	8	16	0.1	0	0	100	V			
2	DCDC_C...	Unsigned	Motorola	Signal	24	16	0.1	0	0	100	I			
3	DCDC_R...	Unsigned	Motorola	Signal	32	1	1	0	0	1	Good/Bad			
4	DCDC_ST...	Unsigned	Motorola	Signal	33	1	1	0	0	1	Off/On			
5	DCDC_H...	Unsigned	Motorola	Signal	34	1	1	0	0	1	Good/Bad			
6	DCDC_E...	Unsigned	Motorola	Signal	35	1	1	0	0	1	Good/Bad			
7	DCDC_F...	Unsigned	Motorola	Signal	36	1	1	0	0	1	Good/Bad			
8	DCDC_ST...	Unsigned	Motorola	Signal	37	1	1	0	0	1	Good/Bad			
9	DCDC_W...	Unsigned	Motorola	Signal	38	1	1	0	0	1	Good/Bad			
10	DCDC_H...	Unsigned	Motorola	Signal	39	1	1	0	0	1	Good/Bad			
11	DCDC_W...	Unsigned	Motorola	Signal	40	1	1	0	0	1	Good/Bad			

Figure 2 NU23 DBC file in Kvaser Database Editor

Using NUCAN on Arduino IDE

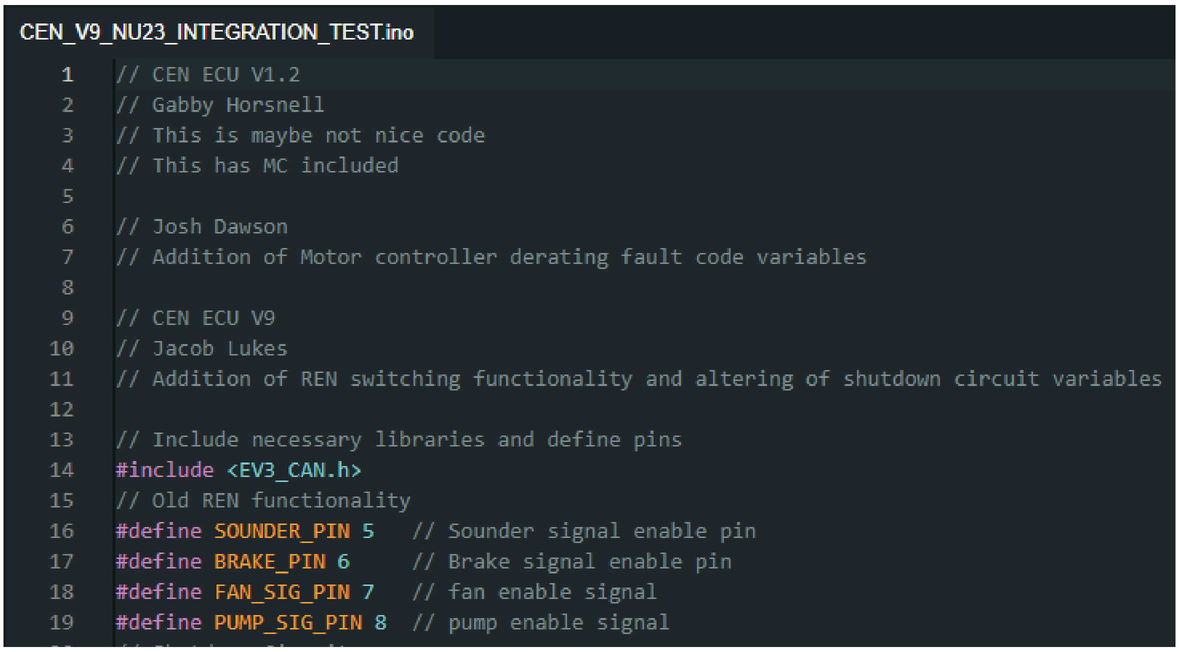
Set up:

Arduino requires the NUCAN library files to be in its libraries folder. The easiest way to ensure it can always find NUCAN is to put the whole NUCAN folder in the Arduino libraries folder (typically found in *User>Documents>Arduino*).

You also need to delete the other unused library files, e.g. SLV24.cpp and SLV24.h as described above now (if you are unsure, ask for help). You can see what files you need to delete by verifying your code, the trouble files will be displayed as an error.

Defining a Library:

Before reading or writing CAN messages with NUCAN, the respective header file for whichever variant of NUCAN you want to use must be included at the top of the Arduino script using '#include <(*file_name*.h)>'.



```
// CEN ECU V1.2
// Gabby Horsnell
// This is maybe not nice code
// This has MC included

// Josh Dawson
// Addition of Motor controller derating fault code variables

// CEN ECU V9
// Jacob Lukes
// Addition of REN switching functionality and altering of shutdown circuit variables

// Include necessary libraries and define pins
#include <EV3_CAN.h>
// Old REN functionality
#define SOUNDER_PIN 5   // Sounder signal enable pin
#define BRAKE_PIN 6     // Brake signal enable pin
#define FAN_SIG_PIN 7   // fan enable signal
#define PUMP_SIG_PIN 8  // pump enable signal
```

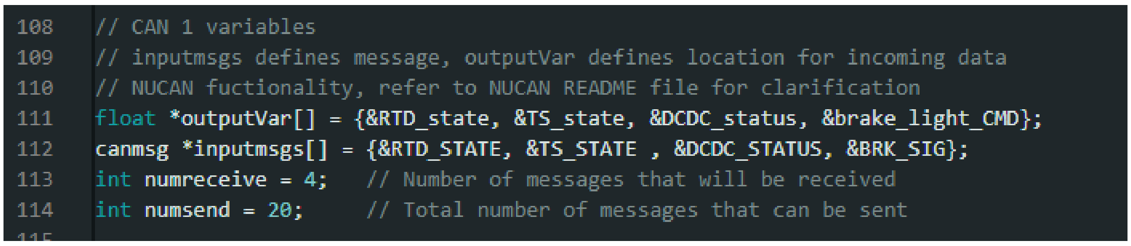
Figure 3 EV3_CAN.h Defined in CEN CODE

The example above uses the EV3 library.

Note: NUCAN does not need to be defined as it is included within each library.

CAN Variable:

Next the variables which NUCAN uses needs to be created. NUCAN requires the number of messages its receiving, sending, and the name and variable to store the value of the received messages.



```
// CAN 1 variables
// inputmsgs defines message, outputVar defines location for incoming data
// NUCAN fuctionality, refer to NUCAN README file for clarification
float *outputVar[] = {&RTD_state, &TS_state, &DCDC_status, &brake_light_CMD};
canmsg *inputmsgs[] = {&RTD_STATE, &TS_STATE , &DCDC_STATUS, &BRK_SIG};
int numreceive = 4;   // Number of messages that will be received
int numsend = 20;     // Total number of messages that can be sent
```

Figure 4 NUCAN Variables in CEN CODE

This is a good example of how to do this, as 4 messages are received, and their values are sorted into variables which were defined earlier in the code.

outputVar[] is an array of variables, these are where the received messages will be sorted for use within the code.

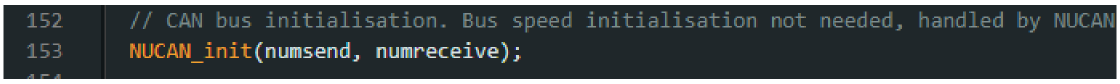
inputmsgs[] is an array of CAN BUS message names, these are the message to be read from the bus.

numreceive is the number of messages that NUCAN expects to read from the CAN BUS.

numsend is the number of messages that NUCAN expect to write onto the CAN BUS.

<u>NUCAN Initialisation:</u>

Now NUCAN variables have been created, NUCAN can be initialised in the 'void setup()' section of the code using 'NUCAN_init(numsend, numreceive)'.



```
// CAN bus initialisation. Bus speed initialisation not needed, handled by NUCAN
NUCAN_init(numsend, numreceive);
```

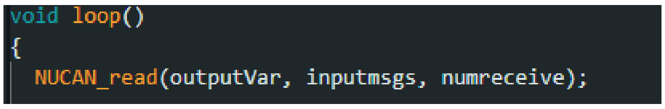
Figure 5 NUCAN initialised in CEN CODE

Above is an example of doing this. NOTE in this example the bus speed is set by NUCAN based on the CANBUS_speed.csv file in the DBC folder.

To set the bus speed manually, simply define the bus speed as a variable prior and include it as the third variable within the NUCAN_init function e.g. NUCAN_init(numsend, numreceive, bus_speed).

<u>Reading messages using NUCAN:</u>

As the number and names of the variables of the received messages have already been defined, NUCAN only needs to be told to read the BUS and update these variables using NUCAN_read(outputVar, inputmsgs, numreceive). This should be within 'void loop()'.

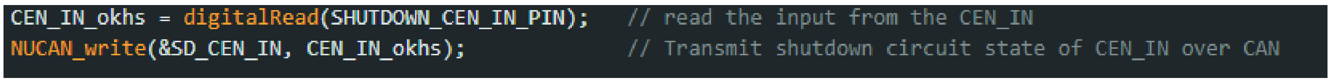


```
void loop()
{
  NUCAN_read(outputVar, inputmsgs, numreceive);
```

Figure 6 Using NUCAN_read in CEN CODE

<u>Writing messages using NUCAN:</u>

Once a variable is made and has a value, it is now ready to be sent using NUCAN_write(). To use this function, put the name of the signal you want this variable to update and then the variable. This name should be identical to the name used in the respective DBC file.



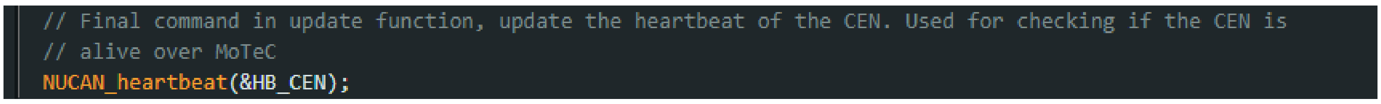
```
CEN_IN_okhs = digitalRead(SHUTDOWN_CEN_IN_PIN);    // read the input from the CEN_IN
NUCAN_write(&SD_CEN_IN, CEN_IN_okhs);              // Transmit shutdown circuit state of CEN_IN over CAN
```

Figure 7 Using NUCAN_write in CEN CODE

In the code above, a variable is defined from a digital read of a pin and then written to the bus using a NUCAN_write().

Using the NUCAN Heartbeat:

To call the NUCAN Heartbeat function, simply include NUCAN_heartbeat() within the void loop() section of the script.



```
// Final command in update function, update the heartbeat of the CEN. Used for checking if the CEN is
// alive over MoTeC
NUCAN_heartbeat(&HB_CEN);
```

Figure 8 Using NUCAN_heartbeat in CEN CODE

In this example, a heartbeat signal is sent to the CAN BUS using a message name specific to that node (i.e. '&HB_CEN').

IMPORTANT: A heartbeat signal is necessary for NUCAN to work as the WATCHDOG will reset the Teensy if it does not see a heartbeat.

Using the NUCAN internal temperature:

This function writes the current CPU temperature of the Teensy. To use it, call NUCAN_Core_Temp() with the specific CAN BUS message name from a respective DBC, just like NUCAN_heartbeat.

e.g. NUCAN_Core_Temp(&LVD_CORE_TEMP).

Include this line within the void loop() section of the code.

By Alec Chapman 3/7/24