DBC Format

From Embedded Systems Learning Academy

DBC file is a proprietary format that describes the data over a CAN bus.

In this article, you will learn the basic syntax of a DBC file that defines up to 8 bytes of CAN message data. A lot of CAN bus related tools can read the DBC file and display values next to each "signal" that you define in the DBC file.

The second part of this article discusses how the auto-generated code can help you read and write the CAN message data. Essentially, each "message" defined in a DBC becomes a C structure with the signals being the members of the C structure.

Contents

DBC Data

Simple DBC Message

A simple DBC message contains the Message ID (MID), and at least one signal. Let's demonstrate by showing a message that contains a single 8-bit signal. **Spaces and the syntax is really strict**, so if you get a single space incorrect, the auto-generation script will likely fail.

```
BO_ 500 IO_DEBUG: 4 IO
 SG_ IO_DEBUG_test_unsigned : 0|8@1+ (1,0) [0|0] "" DBG
```

Observations:

- The message name is **`IO_DEBUG`** and MID is **`500`** (decimal), and the length is **`4`** bytes (though we only need 1 for 8-bit signal)

- The sender is `IO`
- **`0|8`**: The unsigned signal starts at bit position 0, and the size of this signal is 8
- **`(1,0)`**: The scale and offset (discussed later)
- **`[0|0]`**: Min and Max is not defined (discussed later)
- **`""`**: There are no units (it could be, for instance "inches")
- **`@1+`**: Defines that the signal is little-endian, and unsigned: Never change this!

Signed Signal

A signed signal can be sent by simply applying a negative offset to a signal. Let's add a signed signal to the previous message.

```
BO_ 500 IO_DEBUG: 4 IO
 SG_ IO_DEBUG_test_unsigned : 0|8@1+ (1,0) [0|0] "" DBG
 SG_ IO_DEBUG_test_signed : 8|8@1- (1,-128) [0|0] "" DBG
```

Fractional Signals (float)

A floating point variable can be sent by deciding the range, and the precision that you require. For example, if we choose 8-bits, with **`0.1`** as a fraction, we can send the data range of **`0.0 -> 25.5`**. On the other hand, if we want more precision and negative representation, we could use 12-bits with **`0.01`** as a fraction, and an offset. The second fractional signal also contains an explicit minimum and maximum, which is limited by 12-bit that can represent 4096 different numbers, and by factoring in the offset, and using half of the range for negative representation, it ends up with the limited range of **`-20.48 -> 20.47`**

```
BO_ 500 IO_DEBUG: 4 IO
 SG_ IO_DEBUG_test_unsigned : 0|8@1+ (1,0) [0|0] "" DBG
 SG_ IO_DEBUG_test_signed : 8|8@1- (1,-128) [0|0] "" DBG
 SG_ IO_DEBUG_test_float1 : 16|8@1+ (0.1,0) [0|0] "" DBG
 SG_ IO_DEBUG_test_float2 : 24|12@1+ (0.01,-20.48) [-20.48|20.47] "" DBG
```

Enumeration Types

An enumeration type is used where the user wishes to use or see names, instead of numbers. For example, instead of a state machine showing up as "0, 1, 2", we could see it as "stopped, running, paused". It is accomplished by adding two new lines in the DBC file.

A "BA_" field needs to be added, and for the sake of simplicity, you can follow the example below to list an enumeration as a "FieldType" first. Then, you need a "VAL_" field that actually defines the enumeration values.

```
BO_ 500 IO_DEBUG: 4 IO
 SG_ IO_DEBUG_test_enum : 8|8@1+ (1,0) [0|0] "" DBG

BA_ "FieldType" SG_ 500 IO_DEBUG_test_enum "IO_DEBUG_test_enum";

VAL_ 500 IO_DEBUG_test_enum 2 "IO_DEBUG_test2_enum_two" 1 "IO_DEBUG_test2_enum_one" ;
```

Multiplexed Message

A multiplexed message can be used (indirectly) to send more than 8 bytes using a single message ID. For example, if we use a 2-bit MUX, we can send 62-bits of data with four multiplexers (M0, M1, M2, M3). In other words, your message could state that:

- If first 2-bits are 0 (M0), then 62-bits of data is for the car's front sensors
- If first 2-bits are 1 (M1), then 62-bits of data is for the car's rear sensors

Likewise, we could use 8-bit multiplexer, and then we can send 7 bytes of unique data * 256 multiplexers using the same MID. Multiplexed messages are used quite often when:

- Certain information should be grouped under a single MID.

  If there are 100 sensors that use 32-bit value, it is better to use a multipexer rather than 100 different message IDs.

- 11-bit MID does not provide any space to send information under a different MID
- We wish to use the same MID for CAN priority purposes

Observe the following things in the example below:

- There are two multiplexed messages, m0, and m1.

  m0 has the value of 0b0000 for the MUX, and m1 has the value of 0b0001
  We could have a theoretical m15 with value 0b1111 since there is only a 4 bit MUX.

- The 4-bit "M" needs to be defined first.
- In this rather advanced example, we also have a non-mux'd signal called **SENSOR_SONARS_err_count** that is sent with all multiplexed messages
- There are four sensor values sent with multiplexor m0
- There are four "un filtered" sensor values sent with multipexor m1

In conclusion, you can define a multiplexed message that uses a single message ID, however, they are treated, and decoded differently depending on which multipexed value was sent. **In order to send a multiplexed message below, you will have to send two separate messages, one for the m0 and one for the m1**.

```
BO_ 200 SENSOR_SONARS: 8 SENSOR
 SG_ SENSOR_SONARS_mux M : 0|4@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_err_count : 4|12@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_left m0 : 16|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_middle m0 : 28|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_right m0 : 40|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_rear m0 : 52|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_no_filt_left m1 : 16|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_middle m1 : 28|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_right m1 : 40|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_rear m1 : 52|12@1+ (0.1,0) [0|0] "" DBG
```

DBC Example

```
VERSION ""

NS_ :
    BA_
    BA_DEF_
    BA_DEF_DEF_
    BA_DEF_DEF_REL_
    BA_DEF_REL_
    BA_DEF_SGTYPE_
    BA_REL_
    BA_SGTYPE_
    BO_TX_BU_
    BU_BO_REL_
    BU_EV_REL_
    BU_SG_REL_
    CAT_
    CAT_DEF_
    CM_
    ENVVAR_DATA_
    EV_DATA_
    FILTER
    NS_DESC_
    SGTYPE_
    SGTYPE_VAL_
    SG_MUL_VAL_
    SIGTYPE_VALTYPE_
    SIG_GROUP_
    SIG_TYPE_REF_
    SIG_VALTYPE_
    VAL_
    VAL_TABLE_

BS_:

BU_: DBG DRIVER IO MOTOR SENSOR


BO_ 100 DRIVER_HEARTBEAT: 1 DRIVER
 SG_ DRIVER_HEARTBEAT_cmd : 0|8@1+ (1,0) [0|0] "" SENSOR,MOTOR

BO_ 500 IO_DEBUG: 4 IO
 SG_ IO_DEBUG_test_unsigned : 0|8@1+ (1,0) [0|0] "" DBG
 SG_ IO_DEBUG_test_enum : 8|8@1+ (1,0) [0|0] "" DBG
 SG_ IO_DEBUG_test_signed : 16|8@1- (1,0) [0|0] "" DBG
 SG_ IO_DEBUG_test_float : 24|8@1+ (0.5,0) [0|0] "" DBG

BO_ 101 MOTOR_CMD: 1 DRIVER
 SG_ MOTOR_CMD_steer : 0|4@1- (1,-5) [-5|5] "" MOTOR
 SG_ MOTOR_CMD_drive : 4|4@1+ (1,0) [0|9] "" MOTOR

BO_ 400 MOTOR_STATUS: 3 MOTOR
 SG_ MOTOR_STATUS_wheel_error : 0|1@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ MOTOR_STATUS_speed_kph : 8|16@1+ (0.001,0) [0|0] "kph" DRIVER,IO

BO_ 200 SENSOR_SONARS: 8 SENSOR
 SG_ SENSOR_SONARS_mux M : 0|4@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_err_count : 4|12@1+ (1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_left m0 : 16|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_middle m0 : 28|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_right m0 : 40|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_rear m0 : 52|12@1+ (0.1,0) [0|0] "" DRIVER,IO
 SG_ SENSOR_SONARS_no_filt_left m1 : 16|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_middle m1 : 28|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_right m1 : 40|12@1+ (0.1,0) [0|0] "" DBG
 SG_ SENSOR_SONARS_no_filt_rear m1 : 52|12@1+ (0.1,0) [0|0] "" DBG




CM_ BU_ DRIVER "The driver controller driving the car";
CM_ BU_ MOTOR "The motor controller of the car";
CM_ BU_ SENSOR "The sensor controller of the car";
CM_ BO_ 100 "Sync message used to synchronize the controllers";
```

```
BA_DEF_ "BusType" STRING ;
BA_DEF_ BO_ "GenMsgCycleTime" INT 0 0;
BA_DEF_ SG_ "FieldType" STRING ;

BA_DEF_DEF_ "BusType" "CAN";
BA_DEF_DEF_ "FieldType" "";
BA_DEF_DEF_ "GenMsgCycleTime" 0;

BA_ "GenMsgCycleTime" BO_ 100 1000;
BA_ "GenMsgCycleTime" BO_ 500 100;
BA_ "GenMsgCycleTime" BO_ 101 100;
BA_ "GenMsgCycleTime" BO_ 400 100;
BA_ "GenMsgCycleTime" BO_ 200 100;
BA_ "FieldType" SG_ 100 DRIVER_HEARTBEAT_cmd "DRIVER_HEARTBEAT_cmd";
BA_ "FieldType" SG_ 500 IO_DEBUG_test_enum "IO_DEBUG_test_enum";


VAL_ 100 DRIVER_HEARTBEAT_cmd 2 "DRIVER_HEARTBEAT_cmd_REBOOT" 1 "DRIVER_HEARTBEAT_cmd_SYNC" 0 "DRIVER_HEARTBEAT_cmd_NOOP" ;
VAL_ 500 IO_DEBUG_test_enum 2 "IO_DEBUG_test2_enum_two" 1 "IO_DEBUG_test2_enum_one" ;
```

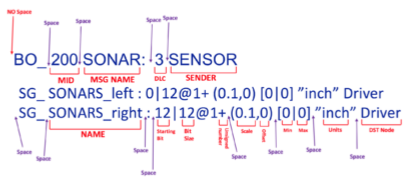
Auto-Generated Code

In a nutshell, the AGC (Auto-Generated-Code) allows a C/C++ application to deal with structures, and avoid the work of packing and unpacking data from a CAN message. Here is an example of a message defined in the DBC, and its auto-generated code artifact:

```
BO_ 101 MOTOR_CMD: 1 DRIVER
 SG_ MOTOR_CMD_steer : 0|4@1- (1,-5) [-5|5] "" MOTOR
 SG_ MOTOR_CMD_drive : 4|4@1+ (1,0) [0|9] "" MOTOR
```

-->

```
typedef struct {
    int8_t MOTOR_CMD_steer;
    uint8_t MOTOR_CMD_drive;
} MOTOR_CMD_t;
```



```
BO_ 200 SONAR: 3 SENSOR
 SG_ SONARS_left : 0|12@1+ (0.1,0) [0|0] "inch" Driver
 SG_ SONARS_right : 12|12@1+ (0.1,0) [0|0] "inch" Driver
```

CAN Communication Handling in C

I created auto-generation tool that operates on the DBC file that produces the AGC. AGC contains useful artifacts such as:

- C structures that store data variables as described in the DBC message
  - Each signal that your controller is a recipient of will occupy a member variable of the struct
  - If your controller is the sender of the message, then all signals of the DBC message will appear in the structs
- Receive handling to convert a CAN message to the C structure
- Transmission handling to convert the C structure into the CAN message's data bytes

The AGC essentially removes the burden of encoding and decoding the CAN message data. For example, you don't have to parse fields individually by doing something such as "bit 5 of byte 6 means foo", and instead, you will simply have a C structure where the CAN message data can be parsed and stored upon.

MIA Handling

MIA means "Missing in Action", and the objective is that if a periodic message is not received as expected, then the contents of the parsed message can be defaulted to the data of your choice. For example, if a temperature sensor reading is not received, then we can tell the AGC to default the sensor reading value to 68 degrees Fahrenheit. This reduces code clutter because each time you use the temperature sensor's value, you don't have to check if the corresponding CAN data was received in the last few seconds.

CAN RX

To handling the messages that are to be received, we must have "glue code" that pieces together the data of a received messages to the target structure. Unfortunately the AGC cannot generate the glue code because it is decoupled from the CAN driver and furthermore, it leaves it up the application to provide the destination structure to convert the CAN data onto.

```
#include "_can_dbc/generated_can.h"
#include "can.h"

// The MIA functions require that you define the:
//   - Time when the handle_mia() functions will replace the data with the MIA
//   - The MIA data itself (ie: MOTOR_STATUS__MIA_MSG)
const uint32_t            SENSOR_SONARS_m0__MIA_MS = 3000;
const SENSOR_SONARS_m0_t  SENSOR_SONARS_m0__MIA_MSG = { 0 };
const uint32_t            SENSOR_SONARS_m1__MIA_MS = 3000;
const SENSOR_SONARS_m1_t  SENSOR_SONARS_m1__MIA_MSG = { 0 };
const uint32_t            MOTOR_STATUS__MIA_MS = 3000;
const MOTOR_STATUS_t      MOTOR_STATUS__MIA_MSG = { 0 };

// For the sake of example, we use global data storage for messages that we receive
SENSOR_SONARS_t sensor_can_msg = { 0 };
MOTOR_STATUS_t motor_status_msg = { 0 };

// Sample periodic task that receives and stores messages
void period_100Hz(void)
{
    can_msg_t can_msg;

    // Empty all of the queued, and received messages within the last 10ms (100Hz callback frequency)
    while (CAN_rx(can1, &can_msg, 0))
    {
        // Form the message header from the metadata of the arriving message
        dbc_msg_hdr_t can_msg_hdr;
        can_msg_hdr.dlc = can_msg.frame_fields.data_len;
        can_msg_hdr.mid = can_msg.msg_id;

        // Attempt to decode the message (brute force, but should use switch/case with MID)
        dbc_decode_SENSOR_SONARS(&sensor_can_msg, can_msg.data.bytes, &can_msg_hdr);
        dbc_decode_MOTOR_STATUS(&motor_status_msg, can_msg.data.bytes, &can_msg_hdr);
    }

    // Service the MIA counters of the MUX'd message
    // successful decoding resets the MIA counter, otherwise it will increment to
    // its MIA value and upon the MIA trigger, it will get replaced by your MIA struct
    dbc_handle_mia_SENSOR_SONARS_m0(&sensor_can_msg.m0, 10);  // 10ms due to 100Hz
    dbc_handle_mia_SENSOR_SONARS_m1(&sensor_can_msg.m1, 10);

    // Service the MIA counter of a regular (non MUX'd) message
    dbc_handle_mia_MOTOR_STATUS(&motor_status_msg, 10);
}
```

CAN TX

The transmission side is simpler than receive because you simply provide an 8-byte data storage for the CAN message, and the encode_ functions convert the C structure onto the 8-bytes that you can send out as part of the CAN message. The encode_ functions also return the message ID and the length of the CAN message and this was done in an effort to decouple the CAN driver from the AGC.

```
void period_10Hz(void)
{
    // Send out Motor command at 10Hz
    MOTOR_CMD_t motor_cmd = { 0 };
    motor_cmd.MOTOR_CMD_drive = 3;
    motor_cmd.MOTOR_CMD_steer = 0;

    can_msg_t can_msg = { 0 };

    // Encode the CAN message's data bytes, get its header and set the CAN message's DLC and length
    dbc_msg_hdr_t msg_hdr = dbc_encode_MOTOR_CMD(can_msg.data.bytes, &motor_cmd);
    can_msg.msg_id = msg_hdr.mid;
    can_msg.frame_fields.data_len = msg_hdr.dlc;

    // Queue the CAN message to be sent out
    CAN_tx(can1, &can_msg, 0);
}
```

CAN TX with callback

There is an alternate method of sending CAN messages that may make your life easier by defining a callback function to send a CAN message. This way, your application doesn't have to deal with a CAN message at all!

```
// This method needs to be defined once, and AGC will call it for all dbc_encode_and_send_FOO() functions
bool dbc_app_send_can_msg(uint32_t mid, uint8_t dlc, uint8_t bytes[8])
{
    can_msg_t can_msg = { 0 };
    can_msg.msg_id                = mid;
    can_msg.frame_fields.data_len = dlc;
    memcpy(can_msg.data.bytes, bytes, dlc);

    return CAN_tx(can1, &can_msg, 0);
}

void period_10Hz(void)
{
    // Send out Motor command at 10Hz
    MOTOR_CMD_t motor_cmd = { 0 };
    motor_cmd.MOTOR_CMD_drive = 3;
    motor_cmd.MOTOR_CMD_steer = 0;

    // This function will encode the CAN data bytes, and send the CAN msg using dbc_app_send_can_msg()
    dbc_encode_and_send_MOTOR_CMD(&motor_cmd);
}
```

Retrieved from "http://socialledge.com/sjsu/index.php?title=DBC_Format&oldid=40046"

Category: Pages with syntax highlighting errors