CBUS Library for Arduino

Introduction

This set of libraries implements a complete CBUS module using the Arduino environment. A minimum of additional code is required to create a fully-functional FLiM-compliant module.

CBUS - an abstract base class containing the common methods
CBUS2515 - an implementation of CBUS specifically for the MCP2515 CAN bus controller
CBUSLED - non-blocking LED management
CBUSswitch - non-blocking pushbutton switch management
CBUSconfig - event and node variable storage in on-chip or external EEPROM

No additional code need be written to integrate with FCU or JMRI or to learn events.

Features include:

- CBUS switch and FLiM/SLiM LEDs
- transition to and from FLiM/SLiM, tested with FCU and JMRI Node Manager
- event learning and storage, with configurable number of event variables
- node variable (NV) modconfiguration and storage
- storage can use the on-chip 1K EEPROM or external I2C EEPROM up to 64K
- reset capability to return the module to an empty configuration
- a user-assignable function to be called when a previously learned event is received
- a user-assignable function to be called when selected CAN bus frames are received
- beta support for RFC0005 long CBUS messages

All five libraries a required although the LED and switch libraries can be used standalone in other projects if you find them useful. The download links are:

CBUS - https://github.com/MERG-DEV/CBUS
CBUS2515 - https://github.com/MERG-DEV/CBUS2515
CBUSLED - https://github.com/MERG-DEV/CBUSLED
CBUSswitch - https://github.com/MERG-DEV/CBUSswitch
CBUSconfig - https://github.com/MERG-DEV/CBUSmodconfig

You can now also download these from the Library Manager in the Arduino IDE. This will also prompt you when updated versions are released.

You will also need to install the following two 3rd party libraries:

ACAN2515 - https://github.com/pierremolinaro/acan2515
Streaming - https://github.com/janelia-arduino/Streaming

For people who may have used other CAN bus libraries (e.g. MCP_CAN), note that the ACAN2515 library implements interrupt handling and configurable send/receive buffers, so there is no need to code this yourself.

Other implementations of the CBUS library are available for ESP32 and SAM3X8E (Arduino Due). This CBUS2515 library supports other processors such as AVR-DA, Raspberry Pi Pico and ESP8266.

Hardware

The minimum hardware required to create a CBUS module is:

- an Arduino processor board, e.g. Uno, Nano, Mega, Pro Mini, etc
- a CAN bus module based on the MCP2515 chip (available from multiple eBay sellers)
- two LEDs (green and yellow/amber) with 1K resistors
- a pushbutton switch

As an alternative, I have designed a generic through-hole PCB containing all the above, as well as a 64K EEPROM chip. It has no module-specific components, but all spare IO pins are brought out to headers. The design files are available on the MERG wiki at:
https://www.merg.org.uk/merg_wiki/doku.php?id=projects:canxmas

If using separate components, connect up as follows:

Arduino Uno pin	CAN bus module pin
5V	Vcc
GND	GND
10 (SS)	CS
12 (MISO)	SO
11 (MOSI)	SI
13 (SCK)	SCK
2 (INT0)	INT

Connect the green LED with its current-limiting resistor between Arduino pin 4 and GND
Connect the yellow LED with its current-limiting resistor between Arduino pin 5 and GND
Connect the pushbutton switch between Arduino pin 6 and GND

Pinouts for other modules (e.g. Mega) are well-documented on the Internet.

You may find a breadboard handy for the connections.

Using the library

The CBUS library includes a starter sketch in the example folder (CAN_empty.ino). This creates a complete but 'empty' module with no specific personality.

Here is a commentary on the example code:

1. include the required libraries

```
// 3rd party libraries
#include <Streaming.h>

// CBUS library header files
#include <CBUS2515.h>              // CAN controller and CBUS class
#include <CBUSswitch.h>            // pushbutton switch
```

```
#include <CBUSLED.h>                  // CBUS LEDs
#include <CBUSconfig.h>               // module configuration
#include <CBUSParams.h>               // CBUS parameters
#include <cbusdefs.h>                 // MERG CBUS constants
```

2. set constants

```
// constants
const byte VER_MAJ = 1;               // code major version
const char VER_MIN = 'a';             // code minor version
const byte VER_BETA = 0;              // code beta sub-version
const byte MODULE_ID = 99;            // CBUS module type

const byte LED_GRN = 4;               // CBUS green SLiM LED pin
const byte LED_YLW = 5;               // CBUS yellow FLiM LED pin
const byte SWITCH0 = 6;               // CBUS push button switch pin
```

3. create the CBUS objects

```
// CBUS objects
CBUSConfig modconfig;                 // configuration object
CBUS2515 CBUS(&modconfig);            // CBUS object
CBUSLED ledGrn, ledYlw;               // two LED objects
CBUSSwitch pb_switch;                 // switch object
```

3. set the module's name:

```
// module name, must be 7 characters, space padded.
unsigned char mname[7] = { 'E', 'M', 'P', 'T', 'Y', ' ', ' ' };
```

4. in the setup() function:

(a) call the sketch's CBUS setup function:

```
setupCBUS();
```

This function contains all the CBUS configuration and initialisation:

```
// set config layout parameters
modconfig.EE_NVS_START = 10;
modconfig.EE_NUM_NVS = 10;
modconfig.EE_EVENTS_START = 50;
modconfig.EE_MAX_EVENTS = 32;
modconfig.EE_NUM_EVS = 1;
modconfig.EE_BYTES_PER_EVENT = (modconfig.EE_NUM_EVS + 4);

// initialise and load configuration
modconfig.setEEPROMtype(EEPROM_INTERNAL);
modconfig.begin();                            // assign to CBUS

CBUS.setParams(params);
CBUS.setName(mname);

// set module parameters
CBUSParams params(modconfig);
params.setVersion(VER_MAJ, VER_MIN, VER_BETA);
params.setModuleId(MODULE_ID);
params.setFlags(PF_FLiM | PF_COMBI);

// assign to CBUS
```

```
CBUS.setParams(params.getParams());
CBUS.setName(mname);

// set CBUS LED pins and assign to CBUS
ledGrn.setPin(LED_GRN);
ledYlw.setPin(LED_YLW);
CBUS.setLEDs(ledGrn, ledYlw);

// initialise CBUS switch and assign to CBUS
pb_switch.setPin(SWITCH0, LOW);
pb_switch.run();
CBUS.setSwitch(pb_switch);

// module reset - if switch is depressed at startup and module is in SLiM mode
if (pb_switch.isPressed() && !modconfig.FLiM) {
  Serial << F("> switch was pressed at startup in SLiM mode") << endl;
  modconfig.resetModule(ledGrn, ledYlw, pb_switch);
}

// register our CBUS event handler, to receive event messages of learned events
CBUS.setEventHandler(eventhandler);

// register our CAN frame handler, to receive *every* CAN frame
CBUS.setFrameHandler(framehandler);

// set CBUS LEDs to indicate the current mode
CBUS.indicateMode(modconfig.FLiM);

// configure and start CAN bus and CBUS message processing
CBUS.setNumBuffers(2, 1);      // more buffers = more memory used, fewer = less
CBUS.setOscFreq(16000000UL);   // select the crystal frequency of the CAN module
CBUS.setPins(10, 2);           // select pins for CAN bus CE and interrupt connections

if (!CBUS.begin()) {
  Serial << F("> error starting CBUS") << endl;
}
```

4. implement a simple loop() function:

```
void loop() {

  //
  /// do CBUS message, switch and LED processing
  //

  CBUS.process();
}
```

5. implement a simple user-defined function to handle received events (note that EVs and NVs number from one, not zero):

```
void eventhandler(byte index, CANFrame *msg) {
  // as an example, display the opcode and the first EV of this event
  Serial << F("> event handler: index = ") << index << F(", opcode = 0x") << _HEX(msg-
>data[0]) << endl;
  Serial << F("> EV1 = ") << modconfig.getEventEVval(index, 1) << endl;
  return; }
```

6. implement a simple user-defined function to handle all received CAN bus frames:

```
void framehandler(CANFrame *msg) {
```

```
  // as an example, format and display the received frame

  Serial << "[ " << (msg->id & 0x7f) << "] [" << msg->len << "] [";

  for (byte d = 0; d < msg->len; d++) {
    Serial << " 0x" << _HEX(msg->data[d]);
  }

  Serial << " ]" << endl;
  return;
}
```

Adding the module's personality

The foregoing implements a module that has no useful functionality.

Consumer Modules

A user-defined event handler function is called whenever a CBUS accessory event that has previously been learned is received; this is the entry point for implementing a Consumer module. It goes without saying that this function will not be called until the module has been taught at least one event.

The following useful data is passed to this function when it is called:

```
byte index;
```

This is the index into the module's event table. The Event Variables (EVs) can then be located.

```
CANFrame *msg;
```

A pointer to a CANFrame object containing the following:

```
msg->id                     // the CANID of the sending node
msg->len                    // the number of data bytes in the frame's payload
msg->data[0]                // the opcode of the received message
msg->data[1] and [2]        // the Node Number (NN) of the sending module
msg->data[3] and [4]        // the Event Number (EN) of the received message
```

Bytes 5-7 are additional data bytes that are send by some opcodes. See the CBUS Developers' Guide for more information on opcodes.

A further user-defined function can be called for every CAN bus frame received. It receives a pointer to the CAN frame object, as above. This may prove useful for modules that need to be aware of more than just learned accessory events, for example a module that needs to read NVs from other modules.

Producer Modules

A Producer module will need to send CBUS messages when something of interest happens in 'the outside world', e.g. a switch is pressed, a loco is detected, etc. It is up to you to define the following items for any CBUS messages you wish to send:

- the opcode, depending on whether it is a simple on/off event (e.g. ACON/ACOF) or something more complex with additional data bytes (e.g. ACON3/ACOF3)
- the Event Number (EN)
- any additional data bytes

This code fragment (not in the example sketch) shows how to send a simple ON event message with event number 1 and no additional data bytes:

```
// create and initialise a message object
CANFrame msg;

// populate the object's parameters

// the size of the data payload
msg.len = 5;

// the opcode
msg.data[0] = OPC_ACON;

// the module's node number (NN)
msg.data[1] = highByte(modconfig.nodeNum);
msg.data[2] = lowByte(modconfig.nodeNum);

// the event number (EN)
msg.data[3] = 0;
msg.data[4] = 1;

// send the message
bool sent_ok = CBUS.sendMessage(&msg);
```

This function returns `true` if the message was successfully sent or `false` if not. Note that the CAN header (comprising the CAN ID and priority bits) is populated implicitly by the sendMessage() method.

Note also that the NN and EN are 16-bit integers and each occupy two (8-bit) bytes of the message. The Arduino macros highByte() and lowByte() return the appropriate byte part from a 16-bit integer. See the CBUS Developers' Guide for a full discussion of short and long events and the meaning of NN and EN.

Module Reset

The Arduino environment provides no easy way to pre-program the microcontroller's on-chip EEPROM. To ensure that the contents of the EEPROM are cleared and set to sensible defaults, the CBUSconfig library provides a simple method for resetting the module. This is shown in the example program included with the library.

Hold down the pushbutton switch as you power-on the module. Then, as a safety precaution, press and hold the switch for a further 5 seconds. The module will then reset the EEPROM contents and reboot. The Node Number and CANID will both be set to zero.

The module can only be reset whilst in SLiM mode (with the green LED illuminated). If, due to random EEPROM data, the module starts up in FLiM mode (with the yellow LED illuminated), or you want to reset the module at any time in the future, hold the switch down for 6 seconds to revert to SLiM mode. You can then proceed to reset the module.

CBUS SLiM Mode

The library does not currently support SLiM mode; that is, there is no provision for setting the node number or learning events by hardware switches. Therefore, you must use FCU or JMRI to modconfigure your module in FLiM mode. This is in common with most newer MERG CBUS modules.

Support for SLiM modconfiguration and event learning may be added as a future enhancement if demand exists.

It is of course possible to ‘hard-code’ events into the event table.

Arduino Serial Port

The library prints copious debug information but by default this code is commented out. Selected lines can be uncommented to help with code debugging and development, but note that this will increase the program size and memory consumption.