Overview

The scheme for managing and implementation of CAN IDs is covered. The main goals are

- Avoid assignment and usage of duplicate CAN ID

- Provide a reference to the CAN ID for a brief description, type of payload, usage

(usage: early the term “function type” was used to what is now called “function” and the original “function” is now called “instance”. Net—when viewing the .sql files beware of the terminology.)

Starting basic layout starts with CANID\_INSERT.sql a Derby database file. CANID names and id hex assignments are made here. Java programs generate a gen\_db.h file for use in C programs, and a file for dealing with parameters in high flash locations of the STM32.

A. Parameters

Parameters are values used by a routine that are not fixed, e.g. calibration of ADC inputs, and CAN IDs. Most programs for the STM32 will use multiple instances of some function, usually one instance per processor (one instance per winch drum), but it could be multiple instances of a function in the same processor (e.g. two strain gauge channels).

Rewording, a parameter list is required for every instance of every function. The program remains the same for all instances, but uses a parameter list for the particular instance.

The basic scheme is that a a working struct (in sram of course) is loaded with the parameters when the program boots up and initializes. There are several ways this is being done.

Two approaches are in use-

1. Database maintains parameters

There are two .sql files that specify the parameters for a function and function instances. A java program converts those into a .c file that is compiled. The compiled list is loaded into the STM32 no differently than any program, but in this case it occupies flash near the top end.

These parameter lists can be loaded either directly (STLink) or over CAN. The loading doesn’t affect lower memory positions.

Program modifications can be made and loaded (either directly or over CAN), and the loading doesn’t affect the parameters.

When the program boots up, and the functions initialize the parameters by copying an essentially a flat file from their section in high flash into the working struct.

2. Subroutine loads parameters

In this scheme the parameters the working struct with the parameters is loaded via a subroutine call during initialization. The subroutine loads the working struct and the result is not different than if the parameters had been copied from high flash.

For the CAN IDs assignment, the database is still useful. CAN IDs are referenced by name and the #define generated from the database assigns the hex value. Referencing by name keeps the assignment of the hex value in one place—the CANID\_INSERT.sql file—and by doing so avoid conflicts.

B. Using a CAN msg in the STM32--

1. Parameter list

The convention has been to have a files named--

<function name>\_idx\_v\_struc.h

<function name>\_idx\_v\_struc.c

The naming originates from the parameter values in high flash are essentially a flat file of four byte entries and the index in the file relates to the struct in .h file.

The ‘idx\_v\_struct.h’ file contains a struct that for historical reasons has the convention of naming--

<struct FUNCTIONLC>

Where, LC refers to “local copy”.

Usually the .c file would then have an instance designated--

<struct FUNCTIONLC functionlc;>

And likely would be made ‘extern’ in the .h file should other routines use it.

a. Parameters from high flash

The ‘idx\_v\_struct.c’ routine would be a copy routine that takes values from high flash and, using the indices generated by the java processing of the database, stores them in the struct.

b. Subroutines loads

The ‘idx\_v\_struct.c’ routine loads the parameters directly, e.g.

p->cid\_drum\_tst\_stepcmd = CANID\_TST\_STEPCMD;

The convention has been to prefix the CAN id parameters with ‘cid’ and use lower case from the name in CANID\_INSERT.sql file. Including the hex assignment from the .sql file in the comments can save looking up the hex value when looking at a cangate list of msgs. (The java cangate version works directly with the database.)

Note that each instance of a function will have its own subroutine.

[At the time of this writing the scheme for selecting the subroutine instance has not been completely worked out. The general idea is a command line parameter for instance causes the Makefile and compiling to use a specific instance subroutine. The database/high flash approach has this built in, but the setup of the parameter values list is somewhat more tedious than the subroutine approach.]

2. CAN msgs

CAN IDs are uint32\_t parameters, handled the same as other uint32\_t parameters.

There are two uses of CAN IDs in a routine. One is for sending and the other for receiving. For sending the CAN node only needs to have the msg payload updated and msg queued for sending by the can\_iface routine. For reception, the CAN hardware filter is loaded with a list of CAN IDs to be allowed to interrupt.

a. structs

**CANRCVBUF**

This is an ancient struct used “everywhere,” and somewhat of a misnomer as it is used for transmit as well as receive. A union was specified in the payload, based on the expectation of little endian between the sending and receiver. For devices sending multi-byte values big endian the union is of little value as the individual byte order needs to be reversed.

/\* Buffering incoming CAN messages \*/

union CANDATA // Unionize for easier cooperation amongst types

{

unsigned long long ull;

signed long long sll;

double dbl;

u32 ui[2];

u16 us[4];

u8 uc[8];

u8 u8[8];

s32 si[2];

s16 ss[4];

s8 sc[8];

float f[2];

};

struct CANRCVBUF // Combine CAN msg ID and data fields

{ // offset name: verbose desciption

u32 id; // 0x00 CAN\_TIxR: mailbox receive register ID p 662

u32 dlc; // 0x04 CAN\_TDTxR: time & length p 660

union CANDATA cd; // 0x08,0x0C CAN\_TDLxR,CAN\_TDLxR: Data payload (low, high)

};

**CANTXQMSG**

This is used to convey the info to the CAN routine that queues and sends the CAN msgs (can\_iface.c). CanTask.c takes structs from the queue and passes the arguments to can\_iface.c The struct includes the basic CAN msg struct, but adds a pointer to the control block for the CAN module (CAN1, CAN2, CAN3) to be used, a retry count, and some bits (described below).

/\* CAN msg passed on queue from a Task sending a CAN msg \*/

struct CANTXQMSG

{

struct CANRCVBUF can; // CAN msg

struct CAN\_CTLBLOCK\* pctl; // Pointer to control block for this CAN

uint8\_t maxretryct;

uint8\_t bits;

};

The ‘bits’ are specified in can\_iface.h. Note that the byte is used differently for TX and RX, hence the union CAN\_X.

/\* In the following RX uses 'xw' and TX uses 'xb[]' \*/

union CAN\_X

{

uint32\_t xw; // RX (DTW)

u8 xb[4]; // TX

};

/\* xb usage for TX \*/

// retryct xb[0] // Counter for number of retries for TERR errors

// maxretryct xb[1] // Maximum number of TERR retry counts

// bits xb[2] // Use these bits to set some conditions (see below)

// nosend xb[3] // Do not send: 0 = send; 1 = do NOT send on CAN bus (internal use only)

/\* bit definition within bits, xb[2] \*/

#define SOFTNART 0x01 // 1 = No retries (including arbitration); 0 = retries

#define NOCANSEND 0x02 // 1 = Do not send to the CAN bus

#define CANMSGLOOPBACKBIT 0x04 // 1 = Loopback: copy of outgoing msg appears in incoming

The SOFTNART bit (soft no automatic retry) is used for CAN msgs where retry is not desired, e.g. sending CAN msgs where the receivers are phase locking to the time of the msg.

The CANMSGLOOPBACKBIT causes the CAN transmit interrupt to send a copy of the msg that has completed the sending to be loaded in the circular buffer for incoming CAN msgs.

**CANRCVBUFN**

Located in can\_iface.h, this is used for received CAN msgs. To the basic CAN msg it adds the CAN module that the CAN msgs came from, plus the 32b system DTW counter at the time the interrupt was serviced.

/\* Received CAN msg plus CAN module identification \*/

// This allows "someone" to associate the msg with the CAN module

struct CANRCVBUFN

{

struct CANRCVBUF can; // Our standard CAN msg

struct CAN\_CTLBLOCK\* pctl; // Pointer to control block for this CAN

uint32\_t toa; // Time-Of-Arrival: CAN msg arrival

};

**MAILBOXCAN**

Received (and loopback) CAN msgs are placed in a circular buffer by can\_iface.c interrupt handling. MailboxTask is notified. MailboxTask checks the CAN id against the list of subscribed CAN ids and makes notification to tasks. It can also notify tasks that subscribe to access to the circular buffer and those tasks have their own pointer into the circular buffer and do their own “mail sorting.” MailboxTask will also notify the gateway function if the gateway function code has been included via a #ifdef in main.h.

/\* CAN readings mailbox \*/

struct MAILBOXCAN

{

struct CANRCVBUFN ncan; // CAN msg plus DTW and CAN control block pointer (pctl)

struct MAILBOXREADINGS mbx; // Readings extracted from CAN msg

struct CANNOTIFYLIST\* pnote; // Pointer to notification block; NULL = none

uint32\_t ctr; // Update counter (increment each update)

uint8\_t paytype; // Code for payload type

uint8\_t newflag; // Set to 1 when new msg loaded; user resets if desired.

};

**X. Database update**

When changes are made the database needs to be updated. The .sql files serve as input to build the database.

1. Be sure repo is up-to-date

cd ~/GliderWinchCommons/embed

git pull

2. Start database engine

cd ~/GliderWinchCommons/embed/svn\_common/trunk/db

./db\_start

This may take some time.

2. Run java programs

./db\_updatej

This can take some time during the initial connection and sometimes looks like it is hung.

3. Run java programs and check for errors

./db\_updatej | tee z

cat z | grep ERROR

If there are errors, one way to find and correct them is to import the temp file ‘z’ into an editor and do a search on ERROR. Typical problems:

A field was copied and is a duplicate where duplicates are not permitted. Description fields are easily forgotten as needing to be different.

4. Update repo on github

~/GliderWinchCommons/embed

git commit -m 'changes to database' -a

git push