Fix Me! Elaborate...

Using libcan

What is libcan ?

libcan is library to use the ECAN feature of the PIC18F2682/2685/4682/4685. CAN (for Controller Area Network) is a communication bus for micro-controllers and, hence, enables easy communication between CAN-enabled PIC micro-controllers, such as th PIC18F4685.

Announcement forum post: http://www.midibox.org/forum/index.php?topic=10338.0

libcan has nothing specific to MIOS so you could also use it outside MIOS.

Requirements

libcan uses the AutoTools application skeleton. So if you want to recompile it, you need all the related stuff.

The examples in this page also uses the AutoTools application skeleton.

CAN Wirings

Here we describe some CAN wirings.

Single wire CAN

Dual wire CAN with MCP2551 transceiver

Getting started with libcan

Message flows overview

Initialize CAN

You first have to initialize the CAN module of the PIC. This is done via the *can_init()* function of libcan.

```
void can_init();
```

Sending a message

Sending a data message is done via the *can_transmit_data(...)* function.

```
can_transmit_buffer_control * can_transmit_data(unsigned char local_id,
unsigned char priority, unsigned char length, data_ptr data_field);
```

libcan then processes the message in the following way :

1. Select a message transmit buffer
2. Convert the application-local message ID to the corresponding bus-global message ID
3. Fill the message buffer with the global ID and data
4. Order the PIC to send the message

Receiving messages

Dispatching of data messages is done via the *can_dispatch()* function.

```
void can_dispatch(void);
```

Selected messages are then forwarded to the application via the *can_receive_data_handler()* user handler.

```
extern void can_receive_data_handler(unsigned char local_id, unsigned char
length, data_ptr data_field);
```

Thus to receive messages, one has to call the *can_dispatch()* from its main loop (say, for MIOS, in the *Tick()* user handler), and implement the *can_receive_data_handler()* function.

libcan processes incoming data messages in the following way :

1. Fetch receive buffers until none is full
    1. Select a full receive buffer
    2. Convert the bus-global message ID to the corresponding application-local message ID
    3. If this id has been recognized, call *can_receive_data_handler()* with the local ID and data
    4. Empty the receive buffer

Message IDs

Local IDs are specific to the application. They are the IDs the application uses to reference the types of message it can send and receive.

Global IDs are bus-wide. They are the IDs that are communicated through the bus.

Local IDs are *unsigned char*s whereas global IDs are of type *can_id_t*, a specific PIC-optimized representation of IDs. You can create *can_id_t* instances by using the provided *CAN_STD_ID(id)* and *CAN_EXT_ID(id)* macros.

```
#define CAN_STD_ID(id) \
    (can_id_t) ... // preprocessing of id to make it a standard ID
#define CAN_EXT_ID(id) \
    (can_id_t) ... // preprocessing of id to make it an extended ID
```

> In fact, the *can_id_t* structure is an *unsigned long* typedef. However it does not contain a simple *unsigned long*. The ID is split in a standard and an extended part conforming to the CAN specification, and arranged in a manner suitable to direct consumption by the PIC micro-controller.

libcan uses local IDs to enable easy maintainability of the application. Thus one can change the local IDs or the global IDs independently. That is neat if you have many different applications that needs to communicate over a CAN bus. Also local/global mapping could be changed dynamically, thus enabling reconfiguration of the CAN nodes when a new CAN node enters the bus (see libcannelloni for an example of dynamic reconfiguration of the CAN network).

Conversion between local and global IDs are done via the *can_local_to_global_id(...)* and *can_global_to_local_id(...)* user handlers.

```
extern can_id_t can_local_to_global_id(unsigned char local_id);
extern unsigned char can_global_to_local_id(can_id_t internal_id);
```

The library provides three helper functions *can_helper_table_ltog(...)*, *can_helper_table_gtol(...)* and *can_helper_bdt_gtol(...)* to ease the implementation of the user handlers.

```
can_id_t can_helper_table_ltog(unsigned char local_id,
        const can_id_mapping_element_t const mapping_table[]);
unsigned char can_helper_table_gtol(can_id_t internal_id,
        const can_id_mapping_element_t const mapping_table[]);
```

But let's jump into some examples, and you'll better understand the big picture...

Sending message example

In this example, we send a message for each change of the first encoder of the first shift register.

```
#include <cmios.h>
#include <can.h>

/* Encoder definition table */

MIOS_ENC_TABLE {
        MIOS_ENC_ENTRY(1, , MIOS_ENC_MODE_DETENTED3),
        MIOS_ENC_EOT
};

/* Message ID definitions */
```

```
#define ENCODER_CHANGE 1

const can_id_mapping_element_t outgoingMessageIDs[] = {
        {ENCODER_CHANGE, CAN_STD_ID(10)}
};

/* Application code */

void Init(void) __wparam
{
    MIOS_SRIO_UpdateFrqSet(1);
    MIOS_SRIO_NumberSet(1);
    MIOS_SRIO_DebounceSet();
    MIOS_ENC_SpeedSet(, MIOS_ENC_SPEED_FAST, 2);

    can_init();
}

typedef __data struct {
    unsigned char id;
    char incrementer;
} encoder_change_message_t;

encoder_change_message_t encoderChangeMessage;

void ENC_NotifyChange(unsigned char encoder, char incrementer) __wparam
{
    can_transmit_buffer_control_t * bc;

    encoderChangeMessage.id = encoder;
    encoderChangeMessage.incrementer = incrementer;

    bc = can_transmit_data(ENCODER_CHANGE, ,
            sizeof(encoder_change_message_t),
            (data_ptr) &encoderChangeMessage);
}

can_id_t can_local_to_global_id(unsigned char local_id)
{
    return can_helper_table_ltog(local_id, outgoingMessageIDs);
}
```

The important bits of this example could be summarized as follows :

1. We create an *outgoingMessageIDs* table with a single message with local ID 1 and global ID 10
2. We init the CAN module in our *Init()* user handler
3. We define a structure for the message data and instanciate it for the current message pending transmission
4. In the *ENC_NotifyChange()* user handler, we send a message by calling *can_transmit_data()* with the following arguments :
    - The message local ID : ENCODER_CHANGE

- The message priority (among other messages to send) : 0
- The size of the data to send : sizeof(encoder_change_message_t)
- The data itself : (data_ptr) &encoderChangeMessage);

5. We convert local to global IDs in the *can_local_to_global_id()* by using the helper function *can_helper_table_ltog()* and our *outgoingMessageIDs* table.

Receiving messages example

In the following example, we receive messages that tells us to toggle DOUT pins.

```
#include <cmios.h>
#include <can.h>

/* Message ID definitions */

#define DOUT_TOGGLE 1

const can_id_mapping_element_t incomingMessageIDs[] = {
        {DOUT_TOGGLE, CAN_STD_ID(40)}
};

/* Application code */

void Init(void) __wparam
{
    MIOS_SRIO_UpdateFrqSet(1);
    MIOS_SRIO_NumberSet(1);
    MIOS_SRIO_DebounceSet();
    MIOS_ENC_SpeedSet(, MIOS_ENC_SPEED_FAST, 2);

    can_init();
}

typedef __data struct {
    unsigned char id;
    unsigned char value;
} dout_toggle_message_t;

void doutToggle(dout_toggle_message_t * doutToggleMessage)
{
    MIOS_DOUT_PinSet(doutToggleMessage->id, doutToggleMessage->value);
}

void Tick(void) __wparam
{
    can_dispatch();
}

void can_receive_data_handler(
        unsigned char local_id,
```

```
        unsigned char length,
        data_ptr data_field)
{
    switch (local_id) {
        case DOUT_TOGGLE:
            doutToggle((dout_toggle_message_t *) data_field);
            break;
        default:
            break;
    }
}

unsigned char can_global_to_local_id(can_id_t global_id)
{
    return can_helper_table_gtol(global_id, incomingMessageIDs);
}
```

The important bits of this example could be summarized as follows :

1. We create an *incomingMessageIDs* table with a single message with local ID 1 and global ID 40
2. We init the CAN module in our *Init()* user handler
3. We define a structure for the message data
4. In the *Tick()* user handler, we call *can_dispatch()* to let libcan process incoming messages
5. In the *can_receive_data_handler()* user handler, we fetch the local_id and call the corresponding function to process the message data casted adequately
6. We convert global to local IDs in the *can_global_to_local_id()* by using the helper function *can_helper_table_gtol()* and our *incomingMessageIDs* table.

> Please note the ***data*** storage class in the definition of the *dout_toggle_message_t* message data structure. This one is **really important** due to the way libcan call your *can_receive_data_handler()*. In fact, no data copy is made, and you get a direct pointer to the message receive buffer. Thus not putting a *data* storage class would make sdcc believe your pointer is general-purpose (21bit) pointer instead of a data memory (12 bit) pointer. **Omitting it would lead to a PIC crash and restart!**

You should process the message data before the return of *can_receive_data_handler()*. In fact, after that the receive buffer is freed and you would lose the data. If you want to copy the data to another part of the memory, you can use the *can_helper_copy_data(...)* provided by the library. (Note : this is nothing more than a reimplemented *mem_copy()*!)

```
void can_helper_copy_data(data_ptr from, data_ptr to, unsigned char length);
```

Detailed review of libcan

Local and global message IDs

ID conversion helpers

Message ID tables

Message ID binary decision tree

can_transmit_buffer_control type