```c
// Software CANbus implementation for rp2040
//
// Copyright (C) 2022  Kevin O'Connor <kevin@koconnor.net>
//
// This file may be distributed under the terms of the GNU GPLv3 license.

#include <stdint.h> // uint32_t
#include <string.h> // memset
#include "RP2040.h" // hw_set_bits
#include "can2040.h" // can2040_setup
#include "hardware/regs/dreq.h" // DREQ_PIO0_RX1
#include "hardware/structs/dma.h" // dma_hw
#include "hardware/structs/iobank0.h" // iobank0_hw
#include "hardware/structs/padsbank0.h" // padsbank0_hw
#include "hardware/structs/pio.h" // pio0_hw
#include "hardware/structs/resets.h" // RESETS_RESET_PIO0_BITS


/****************************************************************
 * rp2040 and low-level helper functions
 ****************************************************************/

// Helper compiler definitions
#define barrier() __asm__ __volatile__("": : :"memory")
#define likely(x)       __builtin_expect(!!(x), 1)
#define unlikely(x)     __builtin_expect(!!(x), 0)
#define ARRAY_SIZE(a) (sizeof(a) / sizeof(a[0]))
#define DIV_ROUND_UP(n,d) (((n) + (d) - 1) / (d))

// Helper functions for writing to "io" memory
static inline void writel(void *addr, uint32_t val) {
    barrier();
    *(volatile uint32_t *)addr = val;
}
static inline uint32_t readl(const void *addr) {
    uint32_t val = *(volatile const uint32_t *)addr;
    barrier();
    return val;
}

// rp2040 helper function to clear a hardware reset bit
static void
rp2040_clear_reset(uint32_t reset_bit)
{
    if (resets_hw->reset & reset_bit) {
        hw_clear_bits(&resets_hw->reset, reset_bit);
        while (!(resets_hw->reset_done & reset_bit))
            ;
    }
```

```
}

// Helper to set the mode and extended function of a pin
static void
rp2040_gpio_peripheral(uint32_t gpio, int func, int pull_up)
{
    padsbank0_hw->io[gpio] = (
        PADS_BANK0_GPIO0_IE_BITS
        | (PADS_BANK0_GPIO0_DRIVE_VALUE_4MA << PADS_BANK0_GPIO0_DRIVE_MSB)
        | (pull_up > 0 ? PADS_BANK0_GPIO0_PUE_BITS : 0)
        | (pull_up < 0 ? PADS_BANK0_GPIO0_PDE_BITS : 0));
    iobank0_hw->io[gpio].ctrl = func << IO_BANK0_GPIO0_CTRL_FUNCSEL_LSB;
}


/****************************************************************
 * rp2040 PIO support
 ****************************************************************/

#define PIO_CLOCK_PER_BIT 32
#define PIO_RX_WAKE_BITS 10

#define can2040_offset_sync_found_end_of_message 2u
#define can2040_offset_sync_signal_start 4u
#define can2040_offset_sync_entry 6u
#define can2040_offset_sync_end 13u
#define can2040_offset_shared_rx_read 13u
#define can2040_offset_shared_rx_end 15u
#define can2040_offset_match_load_next 18u
#define can2040_offset_match_end 25u
#define can2040_offset_tx_got_recessive 25u
#define can2040_offset_tx_start 26u
#define can2040_offset_tx_conflict 31u

static const uint16_t can2040_program_instructions[] = {
    0x0085, //  0: jmp    y--, 5
    0x0048, //  1: jmp    x--, 8
    0xe12a, //  2: set    x, 10                  [1]
    0x00cc, //  3: jmp    pin, 12
    0xc000, //  4: irq    nowait 0
    0x00c0, //  5: jmp    pin, 0
    0xc040, //  6: irq    clear 0
    0xe229, //  7: set    x, 9                   [2]
    0xf242, //  8: set    y, 2                   [18]
    0xc104, //  9: irq    nowait 4               [1]
    0x03c5, // 10: jmp    pin, 5                 [3]
    0x0307, // 11: jmp    7                      [3]
    0x0043, // 12: jmp    x--, 3
    0x20c4, // 13: wait   1 irq, 4
    0x4001, // 14: in     pins, 1
    0xa046, // 15: mov    y, isr
    0x00b2, // 16: jmp    x != y, 18
    0xc002, // 17: irq    nowait 2
    0x40eb, // 18: in     osr, 11
    0x4054, // 19: in     y, 20
    0xa047, // 20: mov    y, osr
    0x8080, // 21: pull   noblock
    0xa027, // 22: mov    x, osr
    0x0098, // 23: jmp    y--, 24
    0xa0e2, // 24: mov    osr, y
    0xa242, // 25: nop                       [2]
    0x6021, // 26: out    x, 1
    0xa001, // 27: mov    pins, x
    0x20c4, // 28: wait   1 irq, 4
    0x00d9, // 29: jmp    pin, 25
    0x023a, // 30: jmp    !x, 26             [2]
    0xc027, // 31: irq    wait 7
};

// Local names for PIO state machine IRQs
#define SI_MAYTX     PIO_IRQ0_INTE_SM0_BITS
#define SI_MATCHED   PIO_IRQ0_INTE_SM2_BITS
#define SI_ACKDONE   PIO_IRQ0_INTE_SM3_BITS
#define SI_RX_DATA   PIO_IRQ0_INTE_SM1_RXNEMPTY_BITS
#define SI_TXPENDING PIO_IRQ0_INTE_SM1_BITS // Misc bit manually forced

// Setup PIO "sync" state machine (state machine 0)
static void
pio_sync_setup(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    struct pio_sm_hw *sm = &pio_hw->sm[0];
    sm->execctrl = (
        cd->gpio_rx << PIO_SM0_EXECCTRL_JMP_PIN_LSB
        | (can2040_offset_sync_end - 1) << PIO_SM0_EXECCTRL_WRAP_TOP_LSB
        | can2040_offset_sync_signal_start << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB);
    sm->pinctrl = (
        1 << PIO_SM0_PINCTRL_SET_COUNT_LSB
        | cd->gpio_rx << PIO_SM0_PINCTRL_SET_BASE_LSB);
    sm->instr = 0xe080; // set pindirs, 0
    sm->pinctrl = 0;
    pio_hw->txf[0] = PIO_CLOCK_PER_BIT / 2 * 7 - 5 - 1;
    sm->instr = 0x80a0; // pull block
    sm->instr = can2040_offset_sync_entry; // jmp sync_entry
}

// Setup PIO "rx" state machine (state machine 1)
static void
```

```
pio_rx_setup(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    struct pio_sm_hw *sm = &pio_hw->sm[1];
    sm->execctrl = (
        (can2040_offset_shared_rx_end - 1) << PIO_SM0_EXECCTRL_WRAP_TOP_LSB
        | can2040_offset_shared_rx_read << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB);
    sm->pinctrl = cd->gpio_rx << PIO_SM0_PINCTRL_IN_BASE_LSB;
    sm->shiftctrl = 0; // flush fifo on a restart
    sm->shiftctrl = (PIO_SM0_SHIFTCTRL_FJOIN_RX_BITS
                     | PIO_RX_WAKE_BITS << PIO_SM0_SHIFTCTRL_PUSH_THRESH_LSB
                     | PIO_SM0_SHIFTCTRL_AUTOPUSH_BITS);
    sm->instr = can2040_offset_shared_rx_read; // jmp shared_rx_read
}

// Setup PIO "match" state machine (state machine 2)
static void
pio_match_setup(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    struct pio_sm_hw *sm = &pio_hw->sm[2];
    sm->execctrl = (
        (can2040_offset_match_end - 1) << PIO_SM0_EXECCTRL_WRAP_TOP_LSB
        | can2040_offset_shared_rx_read << PIO_SM0_EXECCTRL_WRAP_BOTTOM_LSB);
    sm->pinctrl = cd->gpio_rx << PIO_SM0_PINCTRL_IN_BASE_LSB;
    sm->shiftctrl = 0;
    sm->instr = 0xe040; // set y, 0
    sm->instr = 0xa0e2; // mov osr, y
    sm->instr = 0xa02a, // mov x, !y
    sm->instr = can2040_offset_match_load_next; // jmp match_load_next
}

// Setup PIO "tx" state machine (state machine 3)
static void
pio_tx_setup(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    struct pio_sm_hw *sm = &pio_hw->sm[3];
    sm->execctrl = cd->gpio_rx << PIO_SM0_EXECCTRL_JMP_PIN_LSB;
    sm->shiftctrl = (PIO_SM0_SHIFTCTRL_FJOIN_TX_BITS
                     | PIO_SM0_SHIFTCTRL_AUTOPULL_BITS);
    sm->pinctrl = (1 << PIO_SM0_PINCTRL_SET_COUNT_LSB
                   | 1 << PIO_SM0_PINCTRL_OUT_COUNT_LSB
                   | cd->gpio_tx << PIO_SM0_PINCTRL_SET_BASE_LSB
                   | cd->gpio_tx << PIO_SM0_PINCTRL_OUT_BASE_LSB);
    sm->instr = 0xe001; // set pins, 1
    sm->instr = 0xe081; // set pindirs, 1
}
```

```c
// Set PIO "sync" machine to signal "may transmit" (sm irq 0) on 11 idle bits
static void
pio_sync_normal_start_signal(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    uint32_t eom_idx = can2040_offset_sync_found_end_of_message;
    pio_hw->instr_mem[eom_idx] = 0xe12a; // set x, 10 [1]
}

// Set PIO "sync" machine to signal "may transmit" (sm irq 0) on 17 idle bits
static void
pio_sync_slow_start_signal(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    uint32_t eom_idx = can2040_offset_sync_found_end_of_message;
    pio_hw->instr_mem[eom_idx] = 0xa127; // mov x, osr [1]
}

// Test if PIO "rx" state machine has overflowed its fifos
static int
pio_rx_check_stall(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    return pio_hw->fdebug & (1 << (PIO_FDEBUG_RXSTALL_LSB + 1));
}

// Set PIO "match" state machine to raise a "matched" signal on a bit sequence
static void
pio_match_check(struct can2040 *cd, uint32_t match_key)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_hw->txf[2] = match_key;
}

// Calculate pos+bits identifier for PIO "match" state machine
static uint32_t
pio_match_calc_key(uint32_t raw_bits, uint32_t rx_bit_pos)
{
    return (raw_bits & 0x1fffff) | ((-rx_bit_pos) << 21);
}

// Cancel any pending checks on PIO "match" state machine
static void
pio_match_clear(struct can2040 *cd)
{
    pio_match_check(cd, 0);
}

// Flush and halt PIO "tx" state machine
```

```c
static void
pio_tx_reset(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_hw->ctrl = ((0x07 << PIO_CTRL_SM_ENABLE_LSB)
                    | (0x08 << PIO_CTRL_SM_RESTART_LSB));
    pio_hw->irq = (SI_MATCHED | SI_ACKDONE) >> 8; // clear PIO irq flags
    // Clear tx fifo
    struct pio_sm_hw *sm = &pio_hw->sm[3];
    sm->shiftctrl = 0;
    sm->shiftctrl = (PIO_SM0_SHIFTCTRL_FJOIN_TX_BITS
                     | PIO_SM0_SHIFTCTRL_AUTOPULL_BITS);
    // Must reset again after clearing fifo
    pio_hw->ctrl = ((0x07 << PIO_CTRL_SM_ENABLE_LSB)
                    | (0x08 << PIO_CTRL_SM_RESTART_LSB));
}

// Queue a message for transmission on PIO "tx" state machine
static void
pio_tx_send(struct can2040 *cd, uint32_t *data, uint32_t count)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_tx_reset(cd);
    pio_hw->instr_mem[can2040_offset_tx_got_recessive] = 0xa242; // nop [2]
    uint32_t i;
    for (i=0; i<count; i++)
        pio_hw->txf[3] = data[i];
    struct pio_sm_hw *sm = &pio_hw->sm[3];
    sm->instr = 0xe001; // set pins, 1
    sm->instr = can2040_offset_tx_start; // jmp tx_start
    sm->instr = 0x20c0; // wait 1 irq, 0
    pio_hw->ctrl = 0x0f << PIO_CTRL_SM_ENABLE_LSB;
}

// Set PIO "tx" state machine to inject an ack after a CRC match
static void
pio_tx_inject_ack(struct can2040 *cd, uint32_t match_key)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_tx_reset(cd);
    pio_hw->instr_mem[can2040_offset_tx_got_recessive] = 0xc023; // irq wait 3
    pio_hw->txf[3] = 0x7fffffff;
    struct pio_sm_hw *sm = &pio_hw->sm[3];
    sm->instr = 0xe001; // set pins, 1
    sm->instr = can2040_offset_tx_start; // jmp tx_start
    sm->instr = 0x20c2; // wait 1 irq, 2
    pio_hw->ctrl = 0x0f << PIO_CTRL_SM_ENABLE_LSB;

    pio_match_check(cd, match_key);
}

// Did PIO "tx" state machine unexpectedly finish a transmit attempt?
static int
pio_tx_did_fail(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    // Check for passive/dominant bit conflict without parser noticing
    if (pio_hw->sm[3].addr == can2040_offset_tx_conflict)
        return !(pio_hw->intr & SI_RX_DATA);
    // Check for unexpected drain of transmit queue without parser noticing
    return (!(pio_hw->flevel & PIO_FLEVEL_TX3_BITS)
            && (pio_hw->intr & (SI_MAYTX | SI_RX_DATA)) == SI_MAYTX);
}

// Enable host irqs for state machine signals
static void
pio_irq_set(struct can2040 *cd, uint32_t sm_irqs)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_hw->inte0 = sm_irqs | SI_RX_DATA;
}

// Return current host irq mask
static uint32_t
pio_irq_get(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    return pio_hw->inte0;
}

// Raise the txpending flag
static void
pio_signal_set_txpending(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_hw->irq_force = SI_TXPENDING >> 8;
}

// Clear the txpending flag
static void
pio_signal_clear_txpending(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_hw->irq = SI_TXPENDING >> 8;
}

// Setup PIO state machines
static void
```

```c
pio_sm_setup(struct can2040 *cd)
{
    // Reset state machines
    pio_hw_t *pio_hw = cd->pio_hw;
    pio_hw->ctrl = PIO_CTRL_SM_RESTART_BITS | PIO_CTRL_CLKDIV_RESTART_BITS;
    pio_hw->fdebug = 0xffffffff;
    pio_hw->irq = 0xff;
    pio_signal_set_txpending(cd);

    // Load pio program
    uint32_t i;
    for (i=0; i<ARRAY_SIZE(can2040_program_instructions); i++)
        pio_hw->instr_mem[i] = can2040_program_instructions[i];

    // Set initial state machine state
    pio_sync_setup(cd);
    pio_rx_setup(cd);
    pio_match_setup(cd);
    pio_tx_setup(cd);

    // Start state machines
    pio_hw->ctrl = 0x07 << PIO_CTRL_SM_ENABLE_LSB;
}

// Initial setup of gpio pins and PIO state machines
static void
pio_setup(struct can2040 *cd, uint32_t sys_clock, uint32_t bitrate)
{
    // Configure pio0 clock
    uint32_t rb = cd->pio_num ? RESETS_RESET_PIO1_BITS : RESETS_RESET_PIO0_BITS;
    rp2040_clear_reset(rb);

    // Setup and sync pio state machine clocks
    pio_hw_t *pio_hw = cd->pio_hw;
    uint32_t div = (256 / PIO_CLOCK_PER_BIT) * sys_clock / bitrate;
    int i;
    for (i=0; i<4; i++)
        pio_hw->sm[i].clkdiv = div << PIO_SM0_CLKDIV_FRAC_LSB;

    // Configure state machines
    pio_sm_setup(cd);

    // Map Rx/Tx gpios
    uint32_t pio_func = cd->pio_num ? 7 : 6;
    rp2040_gpio_peripheral(cd->gpio_rx, pio_func, 1);
    rp2040_gpio_peripheral(cd->gpio_tx, pio_func, 0);
}
```

```c
/*****************************************************************
 * CRC calculation
 ****************************************************************/

// Calculated 8-bit crc table (see scripts/crc.py)
static const uint16_t crc_table[256] = {
    0x0000,0x4599,0x4eab,0x0b32,0x58cf,0x1d56,0x1664,0x53fd,0x7407,0x319e,
    0x3aac,0x7f35,0x2cc8,0x6951,0x6263,0x27fa,0x2d97,0x680e,0x633c,0x26a5,
    0x7558,0x30c1,0x3bf3,0x7e6a,0x5990,0x1c09,0x173b,0x52a2,0x015f,0x44c6,
    0x4ff4,0x0a6d,0x5b2e,0x1eb7,0x1585,0x501c,0x03e1,0x4678,0x4d4a,0x08d3,
    0x2f29,0x6ab0,0x6182,0x241b,0x77e6,0x327f,0x394d,0x7cd4,0x76b9,0x3320,
    0x3812,0x7d8b,0x2e76,0x6bef,0x60dd,0x2544,0x02be,0x4727,0x4c15,0x098c,
    0x5a71,0x1fe8,0x14da,0x5143,0x73c5,0x365c,0x3d6e,0x78f7,0x2b0a,0x6e93,
    0x65a1,0x2038,0x07c2,0x425b,0x4969,0x0cf0,0x5f0d,0x1a94,0x11a6,0x543f,
    0x5e52,0x1bcb,0x10f9,0x5560,0x069d,0x4304,0x4836,0x0daf,0x2a55,0x6fcc,
    0x64fe,0x2167,0x729a,0x3703,0x3c31,0x79a8,0x28eb,0x6d72,0x6640,0x23d9,
    0x7024,0x35bd,0x3e8f,0x7b16,0x5cec,0x1975,0x1247,0x57de,0x0423,0x41ba,
    0x4a88,0x0f11,0x057c,0x40e5,0x4bd7,0x0e4e,0x5db3,0x182a,0x1318,0x5681,
    0x717b,0x34e2,0x3fd0,0x7a49,0x29b4,0x6c2d,0x671f,0x2286,0x2213,0x678a,
    0x6cb8,0x2921,0x7adc,0x3f45,0x3477,0x71ee,0x5614,0x138d,0x18bf,0x5d26,
    0x0edb,0x4b42,0x4070,0x05e9,0x0f84,0x4a1d,0x412f,0x04b6,0x574b,0x12d2,
    0x19e0,0x5c79,0x7b83,0x3e1a,0x3528,0x70b1,0x234c,0x66d5,0x6de7,0x287e,
    0x793d,0x3ca4,0x3796,0x720f,0x21f2,0x646b,0x6f59,0x2ac0,0x0d3a,0x48a3,
    0x4391,0x0608,0x55f5,0x106c,0x1b5e,0x5ec7,0x54aa,0x1133,0x1a01,0x5f98,
    0x0c65,0x49fc,0x42ce,0x0757,0x20ad,0x6534,0x6e06,0x2b9f,0x7862,0x3dfb,
    0x36c9,0x7350,0x51d6,0x144f,0x1f7d,0x5ae4,0x0919,0x4c80,0x47b2,0x022b,
    0x25d1,0x6048,0x6b7a,0x2ee3,0x7d1e,0x3887,0x33b5,0x762c,0x7c41,0x39d8,
    0x32ea,0x7773,0x248e,0x6117,0x6a25,0x2fbc,0x0846,0x4ddf,0x46ed,0x0374,
    0x5089,0x1510,0x1e22,0x5bbb,0x0af8,0x4f61,0x4453,0x01ca,0x5237,0x17ae,
    0x1c9c,0x5905,0x7eff,0x3b66,0x3054,0x75cd,0x2630,0x63a9,0x689b,0x2d02,
    0x276f,0x62f6,0x69c4,0x2c5d,0x7fa0,0x3a39,0x310b,0x7492,0x5368,0x16f1,
    0x1dc3,0x585a,0x0ba7,0x4e3e,0x450c,0x0095
};

// Update a crc with 8 bits of data
static uint32_t
crc_byte(uint32_t crc, uint32_t data)
{
    return (crc << 8) ^ crc_table[((crc >> 7) ^ data) & 0xff];
}

// Update a crc with 8, 16, 24, or 32 bits of data
static inline uint32_t
crc_bytes(uint32_t crc, uint32_t data, uint32_t num)
{
    switch (num) {
    default: crc = crc_byte(crc, data >> 24); /* FALLTHRU */
    case 3:  crc = crc_byte(crc, data >> 16); /* FALLTHRU */
    case 2:  crc = crc_byte(crc, data >> 8);  /* FALLTHRU */
    case 1:  crc = crc_byte(crc, data);
    }
    return crc;
}


/****************************************************************
 * Bit unstuffing
 ****************************************************************/

// Add 'count' number of bits from 'data' to the 'bu' unstuffer
static void
unstuf_add_bits(struct can2040_bitunstuffer *bu, uint32_t data, uint32_t count)
{
    uint32_t mask = (1 << count) - 1;
    bu->stuffed_bits = (bu->stuffed_bits << count) | (data & mask);
    bu->count_stuff = count;
}

// Reset state and set the next desired 'num_bits' unstuffed bits to extract
static void
unstuf_set_count(struct can2040_bitunstuffer *bu, uint32_t num_bits)
{
    bu->unstuffed_bits = 0;
    bu->count_unstuff = num_bits;
}

// Clear bitstuffing state (used after crc field to avoid bitstuffing ack field)
static void
unstuf_clear_state(struct can2040_bitunstuffer *bu)
{
    uint32_t lb = 1 << bu->count_stuff;
    bu->stuffed_bits = (bu->stuffed_bits & (lb - 1)) | (lb << 1);
}

// Restore raw bitstuffing state (used to undo unstuf_clear_state() )
static void
unstuf_restore_state(struct can2040_bitunstuffer *bu, uint32_t data)
{
    uint32_t cs = bu->count_stuff;
    bu->stuffed_bits = (bu->stuffed_bits & ((1 << cs) - 1)) | (data << cs);
}

// Pull bits from unstuffer (as specified in unstuf_set_count() )
static int
unstuf_pull_bits(struct can2040_bitunstuffer *bu)
{
    uint32_t sb = bu->stuffed_bits, edges = sb ^ (sb >> 1);
    uint32_t e2 = edges | (edges >> 1), e4 = e2 | (e2 >> 2), rm_bits = ~e4;
    uint32_t cs = bu->count_stuff, cu = bu->count_unstuff;
    if (!cs)
        // Need more data
        return 1;
    for (;;) {
        uint32_t try_cnt = cs > cu ? cu : cs;
        for (;;) {
            uint32_t try_mask = ((1 << try_cnt) - 1) << (cs + 1 - try_cnt);
            if (likely(!(rm_bits & try_mask))) {
                // No stuff bits in try_cnt bits - copy into unstuffed_bits
                bu->count_unstuff = cu = cu - try_cnt;
                bu->count_stuff = cs = cs - try_cnt;
                bu->unstuffed_bits |= ((sb >> cs) & ((1 << try_cnt) - 1)) << cu;
                if (! cu)
                    // Extracted desired bits
                    return 0;
                break;
            }
            bu->count_stuff = cs = cs - 1;
            if (rm_bits & (1 << (cs + 1))) {
                // High bit is a stuff bit
                if (unlikely(rm_bits & (1 << cs))) {
                    // Six consecutive bits - a bitstuff error
                    if (sb & (1 << cs))
                        return -1;
                    return -2;
                }
                break;
            }
            // High bit not a stuff bit - limit try_cnt and retry
            bu->count_unstuff = cu = cu - 1;
            bu->unstuffed_bits |= ((sb >> cs) & 1) << cu;
            try_cnt /= 2;
        }
        if (likely(!cs))
            // Need more data
            return 1;
    }
}

// Return most recent raw (still stuffed) bits
static uint32_t
unstuf_get_raw(struct can2040_bitunstuffer *bu)
{
    return bu->stuffed_bits >> bu->count_stuff;
}


/****************************************************************
```

```c
 * Bit stuffing
 ****************************************************************/

// Stuff 'num_bits' bits in '*pb' - upper bits must already be stuffed
static uint32_t
bitstuff(uint32_t *pb, uint32_t num_bits)
{
    uint32_t b = *pb, count = num_bits;
    for (;;) {
        uint32_t try_cnt = num_bits, edges = b ^ (b >> 1);
        uint32_t e2 = edges | (edges >> 1), e4 = e2 | (e2 >> 2), add_bits = ~e4;
        for (;;) {
            uint32_t try_mask = ((1 << try_cnt) - 1) << (num_bits - try_cnt);
            if (!(add_bits & try_mask)) {
                // No stuff bits needed in try_cnt bits
                if (try_cnt >= num_bits)
                    goto done;
                num_bits -= try_cnt;
                try_cnt = (num_bits + 1) / 2;
                continue;
            }
            if (add_bits & (1 << (num_bits - 1))) {
                // A stuff bit must be inserted prior to the high bit
                uint32_t low_mask = (1 << num_bits) - 1, low = b & low_mask;
                uint32_t high = (b & ~(low_mask >> 1)) << 1;
                b = high ^ low ^ (1 << (num_bits - 1));
                count += 1;
                if (num_bits <= 4)
                    goto done;
                num_bits -= 4;
                break;
            }
            // High bit doesn't need stuff bit - accept it, limit try_cnt, retry
            num_bits--;
            try_cnt /= 2;
        }
    }
done:
    *pb = b;
    return count;
}

// State storage for building bit stuffed transmit messages
struct bitstuffer_s {
    uint32_t prev_stuffed, bitpos, *buf;
};

// Push 'count' bits of 'data' into stuffer without performing bit stuffing
static void
```

```c
bs_pushraw(struct bitstuffer_s *bs, uint32_t data, uint32_t count)
{
    uint32_t bitpos = bs->bitpos;
    uint32_t wp = bitpos / 32, bitused = bitpos % 32, bitavail = 32 - bitused;
    uint32_t *fb = &bs->buf[wp];
    if (bitavail >= count) {
        fb[0] |= data << (bitavail - count);
    } else {
        fb[0] |= data >> (count - bitavail);
        fb[1] |= data << (32 - (count - bitavail));
    }
    bs->bitpos = bitpos + count;
}

// Push 'count' bits of 'data' into stuffer
static void
bs_push(struct bitstuffer_s *bs, uint32_t data, uint32_t count)
{
    data &= (1 << count) - 1;
    uint32_t stuf = (bs->prev_stuffed << count) | data;
    uint32_t newcount = bitstuff(&stuf, count);
    bs_pushraw(bs, stuf, newcount);
    bs->prev_stuffed = stuf;
}

// Pad final word of stuffer with high bits
static uint32_t
bs_finalize(struct bitstuffer_s *bs)
{
    uint32_t bitpos = bs->bitpos;
    uint32_t words = DIV_ROUND_UP(bitpos, 32);
    uint32_t extra = words * 32 - bitpos;
    if (extra)
        bs->buf[words - 1] |= (1 << extra) - 1;
    return words;
}


/****************************************************************
 * Transmit state tracking
 ****************************************************************/

// Transmit states (stored in cd->tx_state)
enum {
    TS_IDLE = 0, TS_QUEUED = 1, TS_ACKING_RX = 2, TS_CONFIRM_TX = 3
};

// Calculate queue array position from a transmit index
static uint32_t
```

```c
tx_qpos(struct can2040 *cd, uint32_t pos)
{
    return pos % ARRAY_SIZE(cd->tx_queue);
}

// Queue the next message for transmission in the PIO
static uint32_t
tx_schedule_transmit(struct can2040 *cd)
{
    if (cd->tx_state == TS_QUEUED && !pio_tx_did_fail(cd))
        // Already queued or actively transmitting
        return 0;

#if CAN_PICO_MULTI_CORE == 1
    uint32_t save = spin_lock_blocking( cd -> tx_queue_lock.spin_lock );
#endif

    if (cd->tx_push_pos == cd->tx_pull_pos) {
        // No new messages to transmit
        cd->tx_state = TS_IDLE;
        pio_signal_clear_txpending(cd);

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif
        return SI_TXPENDING;
    }
    cd->tx_state = TS_QUEUED;
    struct can2040_transmit *qt = &cd->tx_queue[tx_qpos(cd, cd->tx_pull_pos)];
    pio_tx_send(cd, qt->stuffed_data, qt->stuffed_words);

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif
    return 0;
}

// Setup PIO state for ack injection
static void
tx_inject_ack(struct can2040 *cd, uint32_t match_key)
{
    cd->tx_state = TS_ACKING_RX;
    pio_tx_inject_ack(cd, match_key);
}

// Check if the current parsed message is feedback from current transmit
static int
tx_check_local_message(struct can2040 *cd)
{
    if (cd->tx_state != TS_QUEUED)
        return 0;

#if CAN_PICO_MULTI_CORE == 1
    uint32_t save = spin_lock_blocking( cd -> tx_queue_lock.spin_lock );
#endif

    struct can2040_transmit *qt = &cd->tx_queue[tx_qpos(cd, cd->tx_pull_pos)];
    struct can2040_msg *pm = &cd->parse_msg, *tm = &qt->msg;
    if (qt->crc == cd->parse_crc && tm->id == pm->id && tm->dlc == pm->dlc
        && tm->data32[0] == pm->data32[0] && tm->data32[1] == pm->data32[1]) {
        // This is a self transmit
        cd->tx_state = TS_CONFIRM_TX;

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif

        return 1;
    }

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif
    return 0;
}


/****************************************************************
 * Notification callbacks
 ****************************************************************/

// Report state flags (stored in cd->report_state)
enum {
    RS_NEED_EOF_FLAG = 1<<2,
    // States
    RS_IDLE = 0, RS_NEED_RX_ACK = 1, RS_NEED_TX_ACK = 2,
    RS_NEED_RX_EOF = RS_NEED_RX_ACK | RS_NEED_EOF_FLAG,
    RS_NEED_TX_EOF = RS_NEED_TX_ACK | RS_NEED_EOF_FLAG,
};

// Report error to calling code (via callback interface)
static void
report_callback_error(struct can2040 *cd, uint32_t error_code)
{
    struct can2040_msg msg = {};
    cd->rx_cb(cd, CAN2040_NOTIFY_ERROR | error_code, &msg);
}
```

```c
// Report a received message to calling code (via callback interface)
static void
report_callback_rx_msg(struct can2040 *cd)
{
    cd->rx_cb(cd, CAN2040_NOTIFY_RX, &cd->parse_msg);
}

// Report a message that was successfully transmited (via callback interface)
static void
report_callback_tx_msg(struct can2040 *cd)
{

#if CAN_PICO_MULTI_CORE == 1
    uint32_t save = spin_lock_blocking( cd -> tx_queue_lock.spin_lock );
#endif

    cd->tx_pull_pos++;

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif

    cd->rx_cb(cd, CAN2040_NOTIFY_TX, &cd->parse_msg);
}

// EOF phase complete - report message (rx or tx) to calling code
static void
report_handle_eof(struct can2040 *cd)
{
    if (cd->report_state & RS_NEED_EOF_FLAG) { // RS_NEED_xX_EOF
        // Successfully processed a new message - report to calling code
        pio_sync_normal_start_signal(cd);
        if (cd->report_state == RS_NEED_TX_EOF)
            report_callback_tx_msg(cd);
        else
            report_callback_rx_msg(cd);
    }
    cd->report_state = RS_IDLE;
    pio_match_clear(cd);
}

// Check if in an rx message is being processed
static int
report_is_rx_eof_pending(struct can2040 *cd)
{
    return cd->report_state == RS_NEED_RX_EOF;
}

// Parser found a new message start
```

```c
static void
report_note_message_start(struct can2040 *cd)
{
    pio_irq_set(cd, SI_MAYTX);
}

// Setup for ack injection (if receiving) or ack confirmation (if transmit)
static void
report_note_crc_start(struct can2040 *cd)
{
    int ret = tx_check_local_message(cd);
    if (ret) {
        // This is a self transmit - setup tx eof "matched" signal
        cd->report_state = RS_NEED_TX_ACK;
        uint32_t bits = (cd->parse_crc_bits << 9) | 0x0ff;
        pio_match_check(cd, pio_match_calc_key(bits, cd->parse_crc_pos + 9));
        return;
    }

    // Setup for ack inject (after rx fifos fully drained)
    cd->report_state = RS_NEED_RX_ACK;
    pio_signal_set_txpending(cd);
    pio_irq_set(cd, SI_MAYTX | SI_TXPENDING);
}

// Parser successfully found matching crc
static void
report_note_crc_success(struct can2040 *cd)
{
    if (cd->report_state == RS_NEED_TX_ACK)
        // Enable "matched" irq for fast back-to-back transmit scheduling
        pio_irq_set(cd, SI_MAYTX | SI_MATCHED);
}

// Parser found successful ack
static void
report_note_ack_success(struct can2040 *cd)
{
    if (cd->report_state == RS_IDLE)
        // Got "matched" signal already
        return;
    // Transition RS_NEED_xX_ACK to RS_NEED_xX_EOF
    cd->report_state |= RS_NEED_EOF_FLAG;
}

// Parser found successful EOF
static void
report_note_eof_success(struct can2040 *cd)
{
    if (cd->report_state == RS_IDLE)
        // Got "matched" signal already
        return;
    report_handle_eof(cd);
    pio_irq_set(cd, SI_TXPENDING);
}

// Parser found unexpected data on input
static void
report_note_parse_error(struct can2040 *cd)
{
    if (cd->report_state != RS_IDLE) {
        cd->report_state = RS_IDLE;
        pio_match_clear(cd);
    }
    pio_sync_slow_start_signal(cd);
    pio_irq_set(cd, SI_MAYTX | SI_TXPENDING);
}

// Received PIO rx "ackdone" irq
static void
report_line_ackdone(struct can2040 *cd)
{
    // Setup "matched" irq for fast rx callbacks
    uint32_t bits = (cd->parse_crc_bits << 8) | 0x7f;
    pio_match_check(cd, pio_match_calc_key(bits, cd->parse_crc_pos + 8));
    // Schedule next transmit (so it is ready for next frame line arbitration)
    uint32_t check_txpending = tx_schedule_transmit(cd);
    pio_irq_set(cd, SI_MAYTX | SI_MATCHED | check_txpending);
}

// Received PIO "matched" irq
static void
report_line_matched(struct can2040 *cd)
{
    // A match event indicates an ack and eof are present
    if (cd->report_state != RS_IDLE) {
        // Transition RS_NEED_xX_ACK to RS_NEED_xX_EOF (if not already there)
        cd->report_state |= RS_NEED_EOF_FLAG;
        report_handle_eof(cd);
    }
    // Implement fast back-to-back tx scheduling (if applicable)
    uint32_t check_txpending = tx_schedule_transmit(cd);
    pio_irq_set(cd, check_txpending);
}

// Received 10+ passive bits on the line (between 10 and 17 bits)
static void
report_line_maytx(struct can2040 *cd)
```

```c
{
    // Line is idle - may be unexpected EOF, missed ack injection,
    // or missed "matched" signal.
    if (cd->report_state != RS_IDLE)
        report_handle_eof(cd);
    uint32_t check_txpending = tx_schedule_transmit(cd);
    pio_irq_set(cd, check_txpending);
}

// Schedule a transmit
static void
report_line_txpending(struct can2040 *cd)
{
    if (cd->report_state == RS_NEED_RX_ACK) {
        // Ack inject request from report_note_crc_start()
        uint32_t mk = pio_match_calc_key(cd->parse_crc_bits, cd->parse_crc_pos);
        tx_inject_ack(cd, mk);
        pio_irq_set(cd, SI_MAYTX | SI_ACKDONE);
        return;
    }
    // Tx request from can2040_transmit(), report_note_eof_success(),
    // or report_note_parse_error().
    uint32_t check_txpending = tx_schedule_transmit(cd);
    pio_irq_set(cd, (pio_irq_get(cd) & ~SI_TXPENDING) | check_txpending);
}


/****************************************************************
 * Input state tracking
 ****************************************************************/

// Parsing states (stored in cd->parse_state)
enum {
    MS_START, MS_HEADER, MS_EXT_HEADER, MS_DATA0, MS_DATA1,
    MS_CRC, MS_ACK, MS_EOF0, MS_EOF1, MS_DISCARD
};

// Transition to the next parsing state
static void
data_state_go_next(struct can2040 *cd, uint32_t state, uint32_t num_bits)
{
    cd->parse_state = state;
    unstuf_set_count(&cd->unstuf, num_bits);
}

// Transition to the MS_DISCARD state - drop all bits until 6 passive bits
static void
data_state_go_discard(struct can2040 *cd)
{
    report_note_parse_error(cd);

    if (pio_rx_check_stall(cd)) {
        // CPU couldn't keep up for some read data - must reset pio state
        cd->raw_bit_count = cd->unstuf.count_stuff = 0;
        pio_sm_setup(cd);
        report_callback_error(cd, 0);
    }

    data_state_go_next(cd, MS_DISCARD, 32);
}

// Received six dominant bits on the line
static void
data_state_line_error(struct can2040 *cd)
{
    data_state_go_discard(cd);
}

// Received six unexpected passive bits on the line
static void
data_state_line_passive(struct can2040 *cd)
{
    if (cd->parse_state != MS_DISCARD && cd->parse_state != MS_START) {
        // Bitstuff error
        data_state_go_discard(cd);
        return;
    }

    uint32_t stuffed_bits = unstuf_get_raw(&cd->unstuf);
    uint32_t dom_bits = ~stuffed_bits;
    if (!dom_bits) {
        // Counter overflow in "sync" state machine - reset it
        cd->unstuf.stuffed_bits = 0;
        cd->raw_bit_count = cd->unstuf.count_stuff = 0;
        pio_sm_setup(cd);
        data_state_go_discard(cd);
        return;
    }

    // Look for sof after 10 passive bits (most "PIO sync" will produce)
    if (!(dom_bits & 0x3ff)) {
        data_state_go_next(cd, MS_START, 1);
        return;
    }

    data_state_go_discard(cd);
}
```

```c
// Transition to MS_CRC state - await 16 bits of crc
static void
data_state_go_crc(struct can2040 *cd)
{
    cd->parse_crc &= 0x7fff;

    // Calculate raw stuffed bits after crc and crc delimiter
    uint32_t crcstart_bitpos = cd->raw_bit_count - cd->unstuf.count_stuff - 1;
    uint32_t crc_bits = (unstuf_get_raw(&cd->unstuf) << 15) | cd->parse_crc;
    uint32_t crc_bitcount = bitstuff(&crc_bits, 15 + 1) - 1;
    cd->parse_crc_bits = (crc_bits << 1) | 0x01; // Add crc delimiter
    cd->parse_crc_pos = crcstart_bitpos + crc_bitcount + 1;

    report_note_crc_start(cd);
    data_state_go_next(cd, MS_CRC, 16);
}

// Transition to MS_DATA0 state (if applicable) - await data bits
static void
data_state_go_data(struct can2040 *cd, uint32_t id, uint32_t data)
{
    if (data & (0x03 << 4)) {
        // Not a supported header
        data_state_go_discard(cd);
        return;
    }
    cd->parse_msg.data32[0] = cd->parse_msg.data32[1] = 0;
    uint32_t dlc = data & 0x0f;
    cd->parse_msg.dlc = dlc;
    if (data & (1 << 6)) {
        dlc = 0;
        id |= CAN2040_ID_RTR;
    }
    cd->parse_msg.id = id;
    if (dlc)
        data_state_go_next(cd, MS_DATA0, dlc >= 4 ? 32 : dlc * 8);
    else
        data_state_go_crc(cd);
}

// Handle reception of first bit of header (after start-of-frame (SOF))
static void
data_state_update_start(struct can2040 *cd, uint32_t data)
{
    cd->parse_msg.id = data;
    report_note_message_start(cd);
    data_state_go_next(cd, MS_HEADER, 17);
}
```

```c
// Handle reception of next 17 header bits
static void
data_state_update_header(struct can2040 *cd, uint32_t data)
{
    data |= cd->parse_msg.id << 17;
    if ((data & 0x60) == 0x60) {
        // Extended header
        cd->parse_msg.id = data;
        data_state_go_next(cd, MS_EXT_HEADER, 20);
        return;
    }
    cd->parse_crc = crc_bytes(0, data, 3);
    data_state_go_data(cd, (data >> 7) & 0x7ff, data);
}

// Handle reception of additional 20 bits of "extended header"
static void
data_state_update_ext_header(struct can2040 *cd, uint32_t data)
{
    uint32_t hdr1 = cd->parse_msg.id;
    uint32_t crc = crc_bytes(0, hdr1 >> 4, 2);
    cd->parse_crc = crc_bytes(crc, ((hdr1 & 0x0f) << 20) | data, 3);
    uint32_t id = (((hdr1 << 11) & 0x1ffc0000) | ((hdr1 << 13) & 0x3e000)
                   | (data >> 7) | CAN2040_ID_EFF);
    data_state_go_data(cd, id, data);
}

// Handle reception of first 1-4 bytes of data content
static void
data_state_update_data0(struct can2040 *cd, uint32_t data)
{
    uint32_t dlc = cd->parse_msg.dlc, bits = dlc >= 4 ? 32 : dlc * 8;
    cd->parse_crc = crc_bytes(cd->parse_crc, data, dlc);
    cd->parse_msg.data32[0] = __builtin_bswap32(data << (32 - bits));
    if (dlc > 4)
        data_state_go_next(cd, MS_DATA1, dlc >= 8 ? 32 : (dlc - 4) * 8);
    else
        data_state_go_crc(cd);
}

// Handle reception of bytes 5-8 of data content
static void
data_state_update_data1(struct can2040 *cd, uint32_t data)
{
    uint32_t dlc = cd->parse_msg.dlc, bits = dlc >= 8 ? 32 : (dlc - 4) * 8;
    cd->parse_crc = crc_bytes(cd->parse_crc, data, dlc - 4);
    cd->parse_msg.data32[1] = __builtin_bswap32(data << (32 - bits));
    data_state_go_crc(cd);
}
```

```c
// Handle reception of 16 bits of message CRC (15 crc bits + crc delimiter)
static void
data_state_update_crc(struct can2040 *cd, uint32_t data)
{
    if (((cd->parse_crc << 1) | 1) != data) {
        data_state_go_discard(cd);
        return;
    }

    report_note_crc_success(cd);
    unstuf_clear_state(&cd->unstuf);
    data_state_go_next(cd, MS_ACK, 2);
}

// Handle reception of 2 bits of ack phase (ack, ack delimiter)
static void
data_state_update_ack(struct can2040 *cd, uint32_t data)
{
    if (data != 0x01) {
        // Undo unstuf_clear_state() for correct SOF detection in
        // data_state_line_passive()
        unstuf_restore_state(&cd->unstuf, (cd->parse_crc_bits << 2) | data);

        data_state_go_discard(cd);
        return;
    }
    report_note_ack_success(cd);
    data_state_go_next(cd, MS_EOF0, 4);
}

// Handle reception of first four end-of-frame (EOF) bits
static void
data_state_update_eof0(struct can2040 *cd, uint32_t data)
{
    if (data != 0x0f || pio_rx_check_stall(cd)) {
        data_state_go_discard(cd);
        return;
    }
    unstuf_clear_state(&cd->unstuf);
    data_state_go_next(cd, MS_EOF1, 5);
}

// Handle reception of end-of-frame (EOF) bits 5-7 and first two IFS bits
static void
data_state_update_eof1(struct can2040 *cd, uint32_t data)
{
    if (data >= 0x1c || (data >= 0x18 && report_is_rx_eof_pending(cd)))
        // Message is considered fully transmitted
        report_note_eof_success(cd);

    if (data == 0x1f)
        data_state_go_next(cd, MS_START, 1);
    else
        data_state_go_discard(cd);
}

// Handle data received while in MS_DISCARD state
static void
data_state_update_discard(struct can2040 *cd, uint32_t data)
{
    data_state_go_discard(cd);
}

// Update parsing state after reading the bits of the current field
static void
data_state_update(struct can2040 *cd, uint32_t data)
{
    switch (cd->parse_state) {
    case MS_START: data_state_update_start(cd, data); break;
    case MS_HEADER: data_state_update_header(cd, data); break;
    case MS_EXT_HEADER: data_state_update_ext_header(cd, data); break;
    case MS_DATA0: data_state_update_data0(cd, data); break;
    case MS_DATA1: data_state_update_data1(cd, data); break;
    case MS_CRC: data_state_update_crc(cd, data); break;
    case MS_ACK: data_state_update_ack(cd, data); break;
    case MS_EOF0: data_state_update_eof0(cd, data); break;
    case MS_EOF1: data_state_update_eof1(cd, data); break;
    case MS_DISCARD: data_state_update_discard(cd, data); break;
    }
}


/****************************************************************
 * Input processing
 ****************************************************************/

// Process incoming data from PIO "rx" state machine
static void
process_rx(struct can2040 *cd, uint32_t rx_data)
{
    unstuf_add_bits(&cd->unstuf, rx_data, PIO_RX_WAKE_BITS);
    cd->raw_bit_count += PIO_RX_WAKE_BITS;

    // undo bit stuffing
    for (;;) {
        int ret = unstuf_pull_bits(&cd->unstuf);
        if (likely(ret > 0)) {
            // Need more data
            break;
        } else if (likely(!ret)) {
            // Pulled the next field - process it
            data_state_update(cd, cd->unstuf.unstuffed_bits);
        } else {
            if (ret == -1)
                // 6 consecutive high bits
                data_state_line_passive(cd);
            else
                // 6 consecutive low bits
                data_state_line_error(cd);
        }
    }
}

// Main API irq notification function
void
can2040_pio_irq_handler(struct can2040 *cd)
{
    pio_hw_t *pio_hw = cd->pio_hw;
    uint32_t ints = pio_hw->ints0;
    while (likely(ints & SI_RX_DATA)) {
        uint32_t rx_data = pio_hw->rxf[1];
        process_rx(cd, rx_data);
        ints = pio_hw->ints0;
        if (likely(!ints))
            return;
    }

    if (ints & SI_ACKDONE)
        // Ack of received message completed successfully
        report_line_ackdone(cd);
    else if (ints & SI_MATCHED)
        // Transmit message completed successfully
        report_line_matched(cd);
    else if (ints & SI_MAYTX)
        // Bus is idle, but not all bits may have been flushed yet
        report_line_maytx(cd);
    else if (ints & SI_TXPENDING)
        // Schedule a transmit
        report_line_txpending(cd);
}


/****************************************************************
 * Transmit queuing
 ****************************************************************/
```

```
// API function to check if transmit space available
int
can2040_check_transmit(struct can2040 *cd)
{

#if CAN_PICO_MULTI_CORE == 1
    uint32_t save = spin_lock_blocking( cd -> tx_queue_lock.spin_lock );
#endif

    uint32_t tx_pull_pos = readl(&cd->tx_pull_pos);
    uint32_t tx_push_pos = cd->tx_push_pos;
    uint32_t pending = tx_push_pos - tx_pull_pos;

#if CAN_PICO_MULTI_CORE == 1
    spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif

    return pending < ARRAY_SIZE(cd->tx_queue);
}

// API function to transmit a message
int
can2040_transmit(struct can2040 *cd, struct can2040_msg *msg)
{

#if CAN_PICO_MULTI_CORE == 1
    uint32_t save = spin_lock_blocking( cd -> tx_queue_lock.spin_lock );
#endif

    uint32_t tx_pull_pos = readl(&cd->tx_pull_pos);
    uint32_t tx_push_pos = cd->tx_push_pos;
    uint32_t pending = tx_push_pos - tx_pull_pos;
    if (pending >= ARRAY_SIZE(cd->tx_queue)) {
        // Tx queue full

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif

        return -1;
    }

    // Copy msg into transmit queue
    struct can2040_transmit *qt = &cd->tx_queue[tx_qpos(cd, tx_push_pos)];
    uint32_t id = msg->id;
    if (id & CAN2040_ID_EFF)
        qt->msg.id = id & ~0x20000000;
    else
        qt->msg.id = id & (CAN2040_ID_RTR | 0x7ff);
```

```c
    qt->msg.dlc = msg->dlc & 0x0f;
    uint32_t data_len = qt->msg.dlc > 8 ? 8 : qt->msg.dlc;
    if (qt->msg.id & CAN2040_ID_RTR)
        data_len = 0;
    qt->msg.data32[0] = qt->msg.data32[1] = 0;
    memcpy(qt->msg.data, msg->data, data_len);

    // Calculate crc and stuff bits
    uint32_t crc = 0;
    memset(qt->stuffed_data, 0, sizeof(qt->stuffed_data));
    struct bitstuffer_s bs = { 1, 0, qt->stuffed_data };
    uint32_t edlc = qt->msg.dlc | (qt->msg.id & CAN2040_ID_RTR ? 0x40 : 0);
    if (qt->msg.id & CAN2040_ID_EFF) {
        // Extended header
        uint32_t id = qt->msg.id;
        uint32_t h1 = ((id & 0x1ffc0000) >> 11) | 0x60 | ((id & 0x3e000) >> 13);
        uint32_t h2 = ((id & 0x1fff) << 7) | edlc;
        crc = crc_bytes(crc, h1 >> 4, 2);
        crc = crc_bytes(crc, ((h1 & 0x0f) << 20) | h2, 3);
        bs_push(&bs, h1, 19);
        bs_push(&bs, h2, 20);
    } else {
        // Standard header
        uint32_t hdr = ((qt->msg.id & 0x7ff) << 7) | edlc;
        crc = crc_bytes(crc, hdr, 3);
        bs_push(&bs, hdr, 19);
    }
    uint32_t i;
    for (i=0; i<data_len; i++) {
        uint32_t v = qt->msg.data[i];
        crc = crc_byte(crc, v);
        bs_push(&bs, v, 8);
    }
    qt->crc = crc & 0x7fff;
    bs_push(&bs, qt->crc, 15);
    bs_pushraw(&bs, 1, 1);
    qt->stuffed_words = bs_finalize(&bs);

    // Submit
    writel(&cd->tx_push_pos, tx_push_pos + 1);

    // Wakeup if in TS_IDLE state
    pio_signal_set_txpending(cd);

#if CAN_PICO_MULTI_CORE == 1
        spin_unlock( cd -> tx_queue_lock.spin_lock, save );
#endif

    return 0;
```

```c
}


/****************************************************************
 * Setup
 ****************************************************************/

// API function to initialize can2040 code
void
can2040_setup(struct can2040 *cd, uint32_t pio_num)
{

#if CAN_PICO_MULTI_CORE == 1
    lock_init( &cd -> tx_queue_lock, next_striped_spin_lock_num( ));
#endif

    memset(cd, 0, sizeof(*cd));
    cd->pio_num = !!pio_num;
    cd->pio_hw = cd->pio_num ? pio1_hw : pio0_hw;
}

// API function to configure callback
void
can2040_callback_config(struct can2040 *cd, can2040_rx_cb rx_cb)
{
    cd->rx_cb = rx_cb;
}

// API function to start CANbus interface
void
can2040_start(struct can2040 *cd, uint32_t sys_clock, uint32_t bitrate
              , uint32_t gpio_rx, uint32_t gpio_tx)
{
    cd->gpio_rx = gpio_rx;
    cd->gpio_tx = gpio_tx;
    pio_setup(cd, sys_clock, bitrate);
    data_state_go_discard(cd);
}

// API function to stop and uninitialize can2040 code
void
can2040_shutdown(struct can2040 *cd)
{
    // XXX
}
```