Zephyr® Project

Developer Summit

Zephyr® Project
Developer Summit

Overview of CAN Subsystem in Zephyr

Navin Sankar Nallampatti Velliangiri, *Next Big Thing AG*

Agenda

- What is CAN
- Features and Application
- CAN Bus Architecture
- CAN Subsystem in Zephyr
- Sample application
- Userspace tools

What is CAN

- Controller Area Network
- Developed by Robert Bosch
- ISO 11898
- Multi-master serial and broadcast bus
- Message based protocol
- Speed: 125kbps - 1Mbps
- https://en.wikipedia.org/wiki/CAN_bus

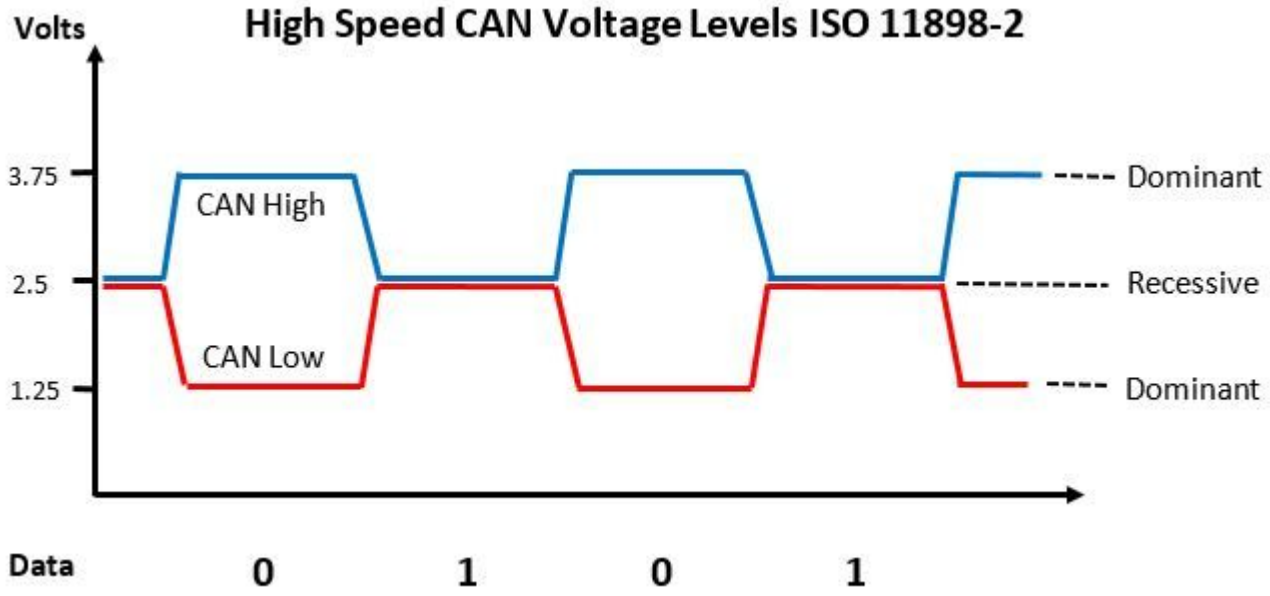
CAN Features

- Multi-Master and Multi-Node Architecture
- Reliable Message Transmission
- Arbitration and prioritization
- Built in Error Detection
- Low cost
- Uses twisted pair wires

CAN Applications

- Automotive
- Aerospace and Aviation
- Industrial Automation
- Building Automation
- Medical Devices

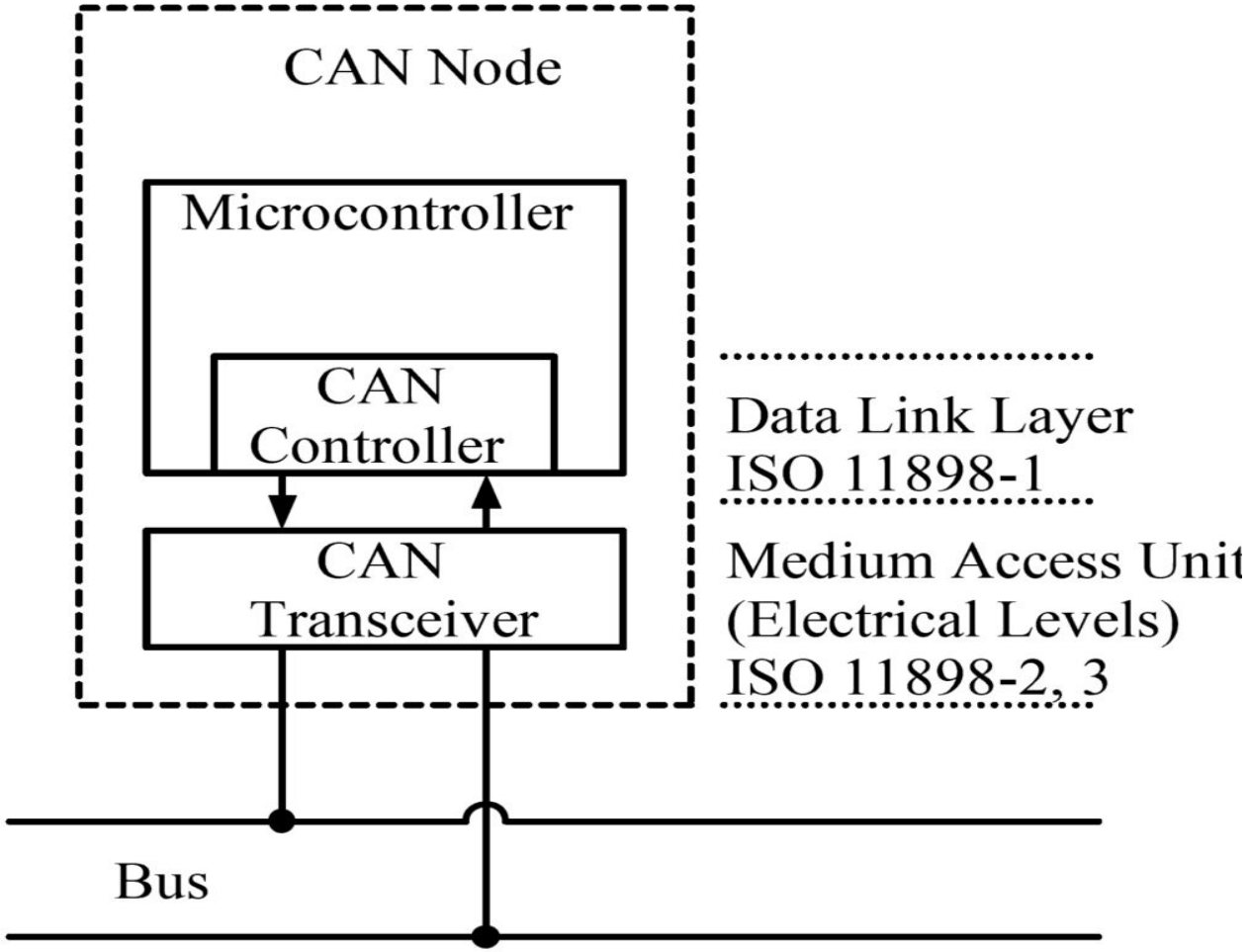
Signalling



- Dominant - Logic 0
- Recessive - Logic 1

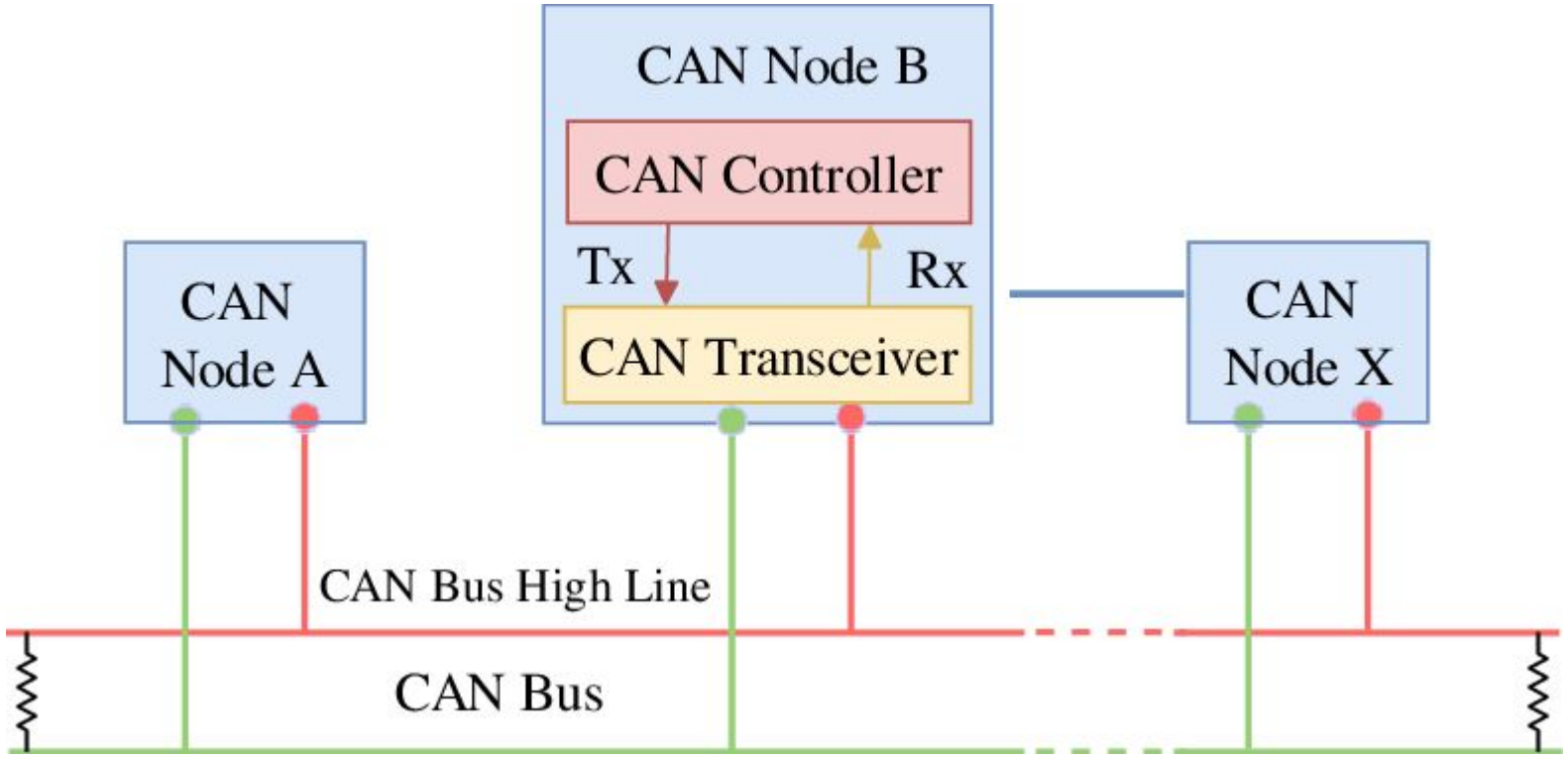
https://www.picotech.com/images/uploads/library/topics/_med/can-voltage-levels.jpg

Node



https://en.wikipedia.org/wiki/CAN_bus#/media/File:CAN_Node.png

CAN Bus Architecture

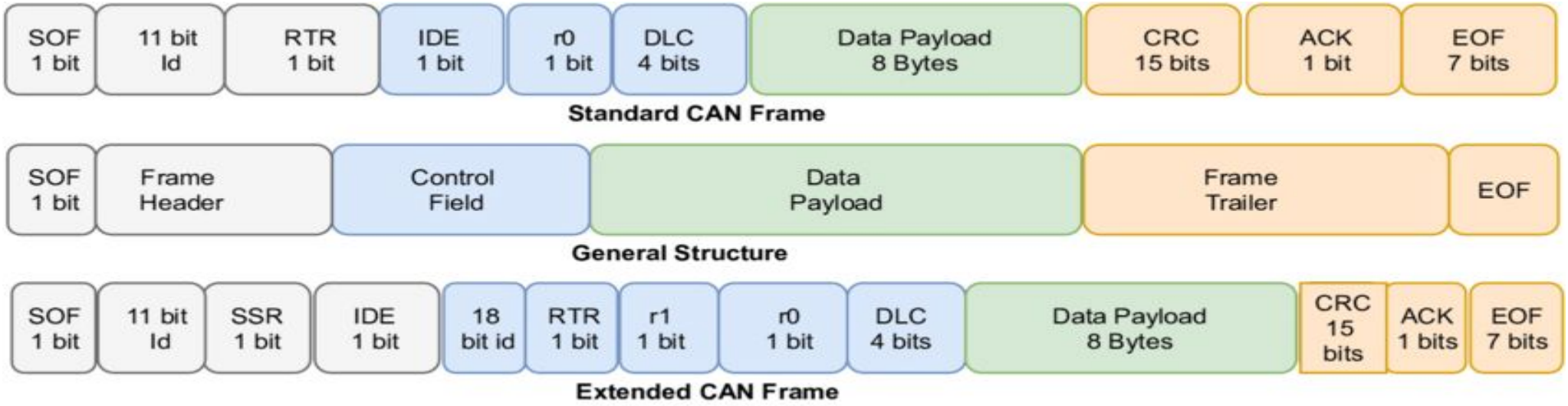


https://www.researchgate.net/publication/353208825/figure/fig1/AS:1044950389104640@1626146995429/The-standard-CAN-bus-node-architecture.png

CAN Message

- The CAN bus is a broadcast type of bus
- All nodes can hear all transmissions.
- Provides local filtering to listen to intended messages
- Four different types of message frame
  - Data Frame
  - Remote Frame
  - Error Frame
  - Overload Frame

CAN Message Frame

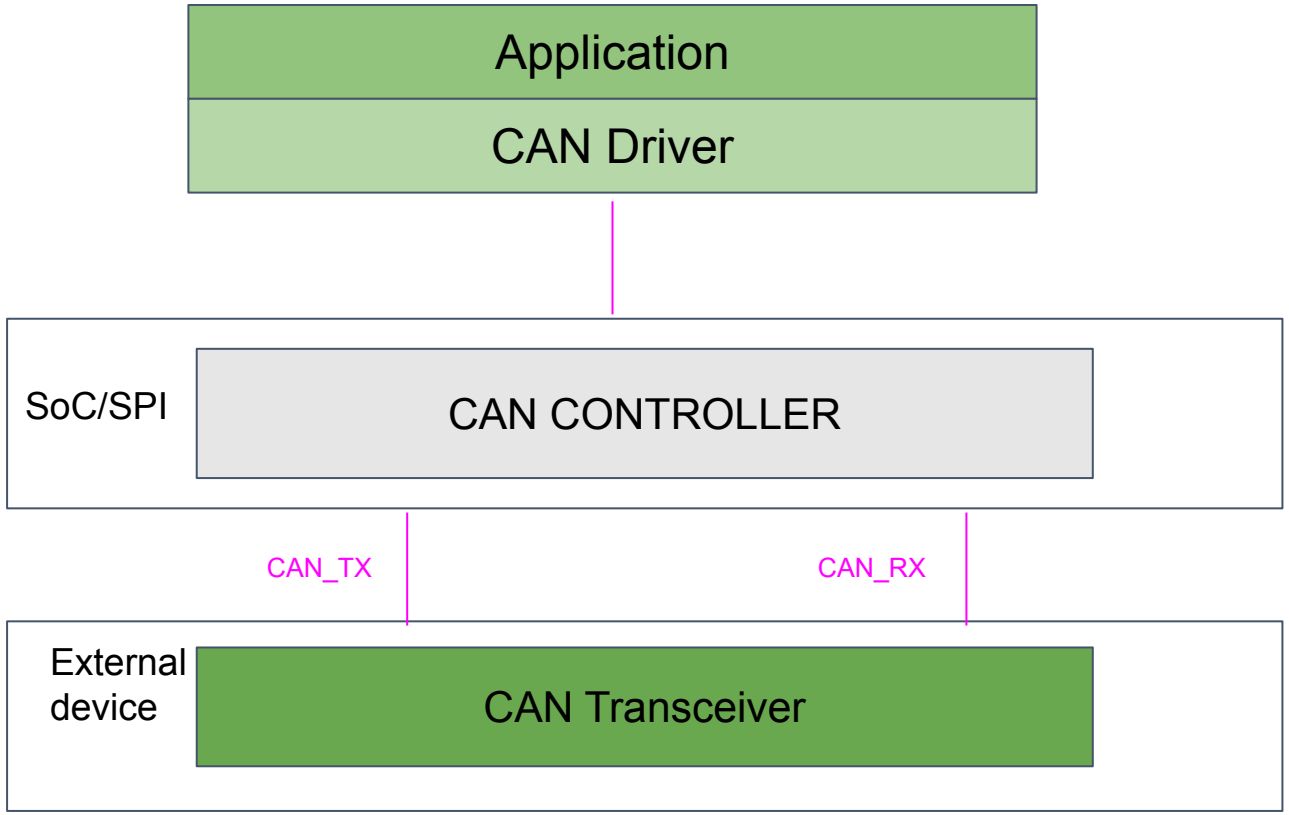


https://www.researchgate.net/profile/Charith-Perera-2/publication/340883976/figure/fig1/AS:883687482732545@1587698922820/Standard-and-extended-Controller-Area-Network-CAN-bus-frame-showing-the-Frame-header.png

Classical CAN vs CAN FD

Features	Classical CAN	CAN FD
Data Rate	upto 1Mbps	upto 8 Mbps
Payload Size	8 bytes payload/frame	64 bytes payload/frame
Network Length	upto 40 Meters	Limited to few meters
Compatibility	Classical CAN is not compatible with CAN FD	CAN FD is backward compatible with Classical CAN

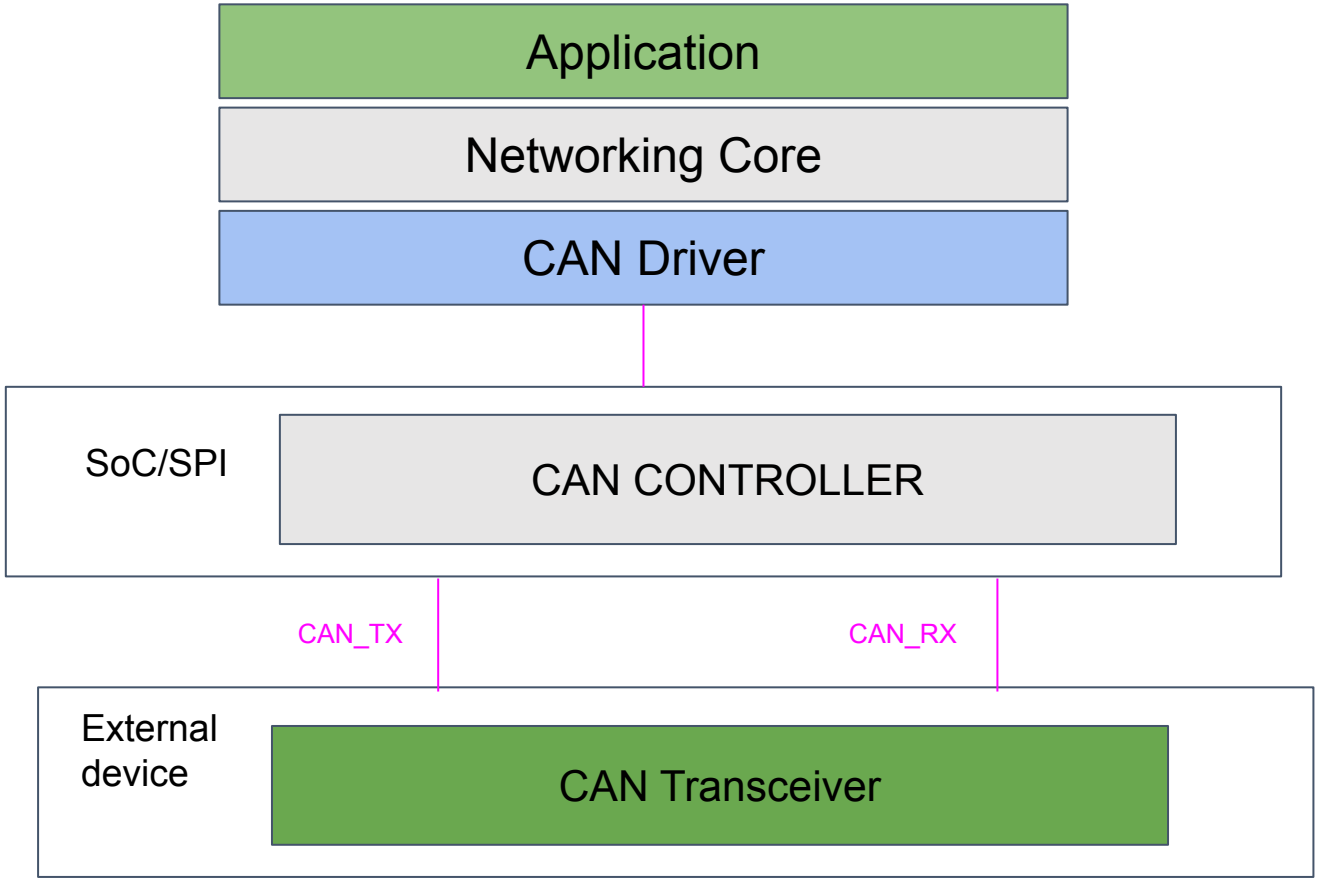
CAN in the Zephyr Driver Model



SocketCAN

- Support’s BSD Socket API
- Compatible with Linux SocketCAN implementation
- Integrated with Zephyr Networking stack

SocketCAN


Application
Networking Core
CAN Driver
SoC/SPI
CAN CONTROLLER
CAN_TX
CAN_RX
External device
CAN Transceiver

Add a CAN controller in Zephyr

- Compatibility
- Devicetree/shield
- Driver implementation
- Sample and test cases

Device tree example

```
&arduino_spi {
        status = "okay";
        cs-gpios = <&arduino_header 16 GPIO_ACTIVE_LOW>; /* D10 */

        mcp2515_dfrobot_can_bus_v2_0: can@0 {
                compatible = "microchip,mcp2515";
                spi-max-frequency = <1000000>;
                int-gpios = <&arduino_header 8 GPIO_ACTIVE_LOW>; /* D2 */
                status = "okay";
                reg = <0x0>;
                osc-freq = <16000000>;
                bus-speed = <125000>;
                sjw = <1>;
                sample-point = <875>;

                can-transceiver {
                        max-bitrate = <1000000>;
                };
        };
};
```

Device driver: declare the driver

```
#define MCP2515_INIT(inst)                                                              \
        static K_KERNEL_STACK_DEFINE(mcp2515_int_thread_stack_##inst,                   \
                                     CONFIG_CAN_MCP2515_INT_THREAD_STACK_SIZE);         \
                                                                                        \
        static struct mcp2515_data mcp2515_data_##inst = {                              \
                .int_thread_stack = mcp2515_int_thread_stack_##inst,                    \
                .tx_busy_map = 0U,                                                      \
                .filter_usage = 0U,                                                     \
        };                                                                              \
                                                                                        \
        static const struct mcp2515_config mcp2515_config_##inst = {                   \
                .bus = SPI_DT_SPEC_INST_GET(inst, SPI_WORD_SET(8), 0),                  \
                .int_gpio = GPIO_DT_SPEC_INST_GET(inst, int_gpios),                     \
                .int_thread_stack_size = CONFIG_CAN_MCP2515_INT_THREAD_STACK_SIZE,      \
                .int_thread_priority = CONFIG_CAN_MCP2515_INT_THREAD_PRIO,              \
                .tq_sjw = DT_INST_PROP(inst, sjw),                                      \
                .tq_prop = DT_INST_PROP_OR(inst, prop_seg, 0),                          \
                .tq_bs1 = DT_INST_PROP_OR(inst, phase_seg1, 0),                         \
                .tq_bs2 = DT_INST_PROP_OR(inst, phase_seg2, 0),                         \
                .bus_speed = DT_INST_PROP(inst, bus_speed),                             \
                .osc_freq = DT_INST_PROP(inst, osc_freq),                               \
                .sample_point = DT_INST_PROP_OR(inst, sample_point, 0),                 \
                .phy = DEVICE_DT_GET_OR_NULL(DT_INST_PHANDLE(inst, phys)),              \
                .max_bitrate = DT_INST_CAN_TRANSCEIVER_MAX_BITRATE(inst, 1000000),      \
        };                                                                              \
                                                                                        \
        DEVICE_DT_INST_DEFINE(inst, &mcp2515_init, NULL, &mcp2515_data_##inst,          \
                              &mcp2515_config_##inst, POST_KERNEL, CONFIG_CAN_INIT_PRIORITY,  \
                              &can_api_funcs);

DT_INST_FOREACH_STATUS_OKAY(MCP2515_INIT)
```

Device driver: init function

```
static int mcp2515_init(const struct device *dev)
{
        // 1. get the config and device data
        // 2. verify the spi bus is ready
        // 3. configure gpio interrupt
        // 4. create thread
        // 5. device specific initialization
}
```

Device driver: api’s

```
static const struct can_driver_api can_api_funcs = {
        .get_capabilities = mcp2515_get_capabilities,
        .set_timing = mcp2515_set_timing,
        .start = mcp2515_start,
        .stop = mcp2515_stop,
        .set_mode = mcp2515_set_mode,
        .send = mcp2515_send,
        .add_rx_filter = mcp2515_add_rx_filter,
        .remove_rx_filter = mcp2515_remove_rx_filter,
        .get_state = mcp2515_get_state,
#ifndef CONFIG_CAN_AUTO_BUS_OFF_RECOVERY
        .recover = mcp2515_recover,
#endif
        .set_state_change_callback = mcp2515_set_state_change_callback,
        .get_core_clock = mcp2515_get_core_clock,
        .get_max_filters = mcp2515_get_max_filters,
        .get_max_bitrate = mcp2515_get_max_bitrate,
        .timing_min = {
                // .
        },
        .timing_max = {
                // .
        }
};
```

Application: Obtaining the device

```
void main(void)
{
        const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));

        if (!device_is_ready(can_dev)) {
                printf("CAN: Device %s not ready.\n", can_dev->name);
                return;
        }

        ret = can_set_mode(can_dev, CAN_MODE_LOOPBACK);
        if (ret != 0) {
                printf("Error setting CAN mode [%d]", ret);
                return;
        }

        ret = can_start(can_dev);
        if (ret != 0) {
                printf("Error starting CAN controller [%d]", ret);
                return;
        }

        return;
}
```

Application: Set Timing

```
ret = can_calc_timing(can_dev, &timing, 250000, 875);
if (ret < 0) {
        LOG_ERR("Failed to calc a valid timing");
}

ret = can_stop(can_dev);
if (ret != 0) {
        LOG_ERR("Failed to stop CAN controller");
}

ret = can_set_timing(can_dev, &timing);
if (ret != 0) {
        LOG_ERR("Failed to set timing");
}

ret = can_start(can_dev);
if (ret != 0) {
        LOG_ERR("Failed to start CAN controller");
}
```

Application: Sending (blocking api)

```
int send_function(const struct device *can_dev)
{
        struct can_frame frame = {
                .flags = 0,
                .id = 0x123,
                .dlc = 8,
                .data = {1,2,3,4,5,6,7,8}
        };

        return can_send(can_dev, &frame, K_MSEC(100), NULL, NULL);
}
```

Application: Sending (non blocking api)

```c
void tx_callback(const struct device *dev, int error, void *user_data)
{
        char *sender = (char *)user_data;

        if (error != 0) {
                LOG_ERR("Sending failed [%d]\nSender: %s\n", error, sender);
        }
}

int send_function(const struct device *can_dev)
{
        struct can_frame frame = {
                .flags = CAN_FRAME_IDE, // uses Extended CAN ID
                .id = 0x1234567,
                .dlc = 2
        };

        frame.data[0] = 1;
        frame.data[1] = 2;

        return can_send(can_dev, &frame, K_FOREVER, tx_irq_callback, "Sender 1");
}
```

Application: Receiving (callback)

```
void rx_callback_function(const struct device *dev, struct can_frame *frame,
                          void *user_data)
{
        // ... do something with the frame ...
}

int set_rx_filter(can_dev)
{
        int filter_id;
        const struct can_filter my_filter = {
                .flags = CAN_FILTER_DATA,
                .id = 0x123,
                .id_mask = CAN_STD_ID_MASK
        };

        filter_id = can_add_rx_filter(can_dev, rx_callback_function, NULL, &my_filter);
        if (filter_id < 0) {
                LOG_ERR("Unable to add rx filter [%d]", filter_id);
        }

        return filter_id;
}
```

Application: Receiving (msgq)

```
CAN_MSGQ_DEFINE(my_can_msgq, 2);

void main(void) {
        int filter_id;
        struct can_frame rx_frame;
        const struct device *const can_dev = DEVICE_DT_GET(DT_CHOSEN(zephyr_canbus));

        // set the mode and start

        const struct can_filter my_filter = {
                .flags = CAN_FILTER_DATA | CAN_FILTER_IDE,
                .id = 0x1234567,
                .id_mask = CAN_EXT_ID_MASK
        };

        filter_id = can_add_rx_filter_msgq(can_dev, &my_can_msgq, &my_filter);
        if (filter_id < 0) {
                LOG_ERR("Unable to add rx msgq [%d]", filter_id);
                return;
        }

        while (true) {
                k_msgq_get(&my_can_msgq, &rx_frame, K_FOREVER);

                // ... do something with the frame ...
        }
}
```

Userspace tools in linux

- The **can-utils** package provides basic tools to display, record, generate and replay CAN traffic.
- Useful for debugging, testing, simple prototyping
- Key Tools:
  - cansend
  - candump
  - cangen
  - canplayer
  - cansniffer
  - canlogserver

cansend: send a single frame

```
$ cansend can0 123#112233
$ cansend can0 456#112233445566
$ cansend can0 456#1122334455667788
```

candump: display, filter and log data to files

```
$ candump can0
can0  123   [3]  11 22 33
can0  456   [8]  11 22 33 44 55 66
can0  789   [8]  11 22 33 44 55 66 77 88
```

Thank you!

#EMBEDDEDOSSUMMIT