CAN SDK
Reference Manual

Document number	2208
Version	2
Issue date	2023-03-06

1 Introduction

1.1 Overview

The CAN SDK described in this document provide three levels of functionality:

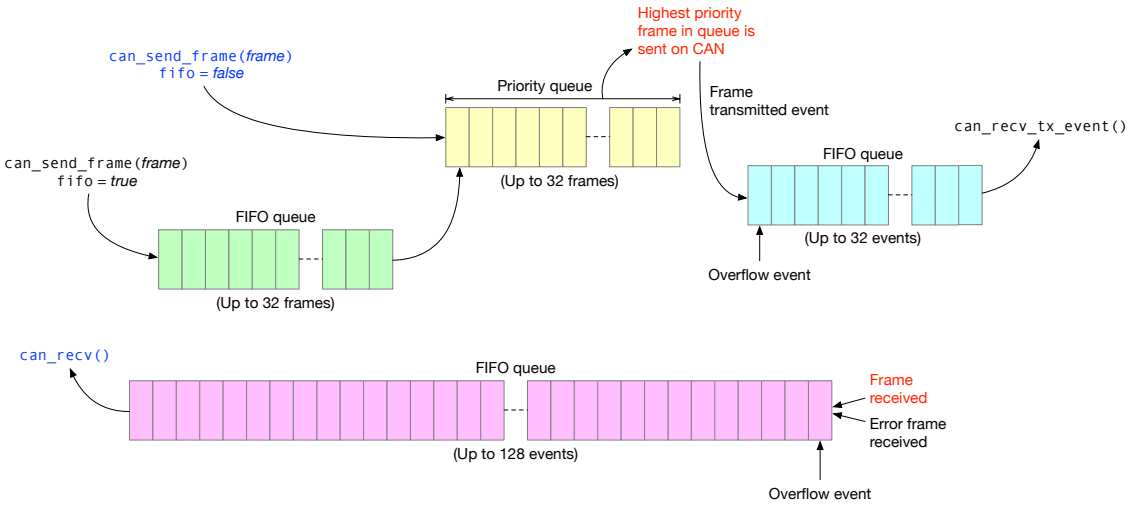
- A generic C CAN API that is intended to compatible with any CAN controller.
- A MCP25xxFD driver intended to work with any microcontroller connected via SPI to a Microchip MCP2517FD or Microchip MCP2518FD or Microchip MCP251863 CAN controller / transceiver chip.
- A binding for the Raspberry Pi Pico C SDK that instantiates the MCP25xxFD driver for a Raspberry Pi Pico with the Canis Labs CANPico board.

The primary goal for the SDK is to provide a safe framework sending and receiving CAN frames where the transmission of CAN frames is **free of priority inversion**.

Priority Inversion is a widespread problem with CAN software that causes timing instability, and it is essential for any robust system that software avoid it by handling transmission priorities properly. For a description of priority inversion with CAN, see the Canis Labs CTO blog page: https://kentindell.github.io/2020/06/29/can-priority-inversion/

1.2 The queueing system

The messaging system of the SDK is illustrated below:



Outgoing frames are normally placed into a priority queue (up to 32 frames) and the highest priority CAN frame in the queue is sent first on the bus. The transmission event is stored in a FIFO queue with a timestamp so that the application can determine how the frames were transmitted. The system is free of priority inversion by use of this priority queue.

Incoming frames are placed into a receive FIFO. As well as frame received events, the FIFO can optionally store error frames, allowing the full traffic on the bus to be determined.

Both the transmit event and receive event FIFOs can have an overflow event placed in them if there is no room for items to be stored in the FIFO. This allows an application to determine if CAN frames have been dropped.

1.3 Interrupts

The system is driven by interrupts (on the MCP25xxFD there is a single level-sensitive interrupt wired to a GPIO pin on the host microcontroller). Four standard interrupt sources are defined:

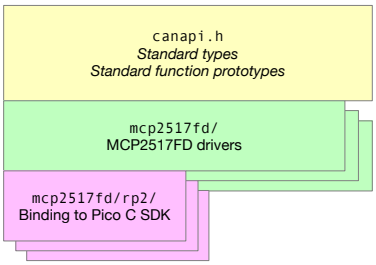
- Frame reception
- Frame transmission
- Error frames
- Mode changes (specifically the transition to Bus Off).

Other controllers may have different interrupts. The binding to a specific host and controller is handled by the SDK structure.

The API defines ISR call-ins and call-backs to allow CAN events to be handled immediately.

1.4 SDK structure

The diagram below illustrates the SDK structure:



The CAN API is made available to the application by including `canapi.h`. The specific device and the specific board are selected with compile-time definitions. For the MCP25xxFD CAN controller on the Raspberry Pi Pico, `MCP25xxFD` and `HOST_CANPICO` are defined.

1.5 Data types

The generic parts of the API begin `can_` and the hardware-specific parts begin with the name of the CAN controller hardware (e.g. `mcp25xxfd_`)

Standard C typedef types are defined for CAN objects. For example, `can_frame_t` defines a standardized way to hold a CAN frame and `can_id_t` defines a standardized way to represent CAN IDs. The application should not directly access the contents of these types but instead use the accessor functions: the contents may change to support future CAN controller hardware (for example, in the future the API will cover CAN-HG frames that can contain very large payloads, such as 276 bytes on a J1939 250kbit/sec bus).

The API structure for a CAN frame contain the attributes of CAN (11- or 29-bit ID, data of 0-8 bytes, remote/data flag, DLC length code) but also includes two further attributes:

- An indication of the filter that matched against a received frame
- A user-defined reference

The filter 'hit' is useful feature for determining the identity of an incoming frame (especially if there is no fixed ID for a frame when the ID contains other parameters, for example with J1939).

The user-defined reference is intended to store an application-specific identity reference for outgoing frames so that transmitted frames. It may be defined as an index into an array or a pointer to a structure (the MicroPython CAN API for CANPico uses this to point to MicroPython `can_make_id` instances that contain a `can_frame_t` structure).

1.6 API and concurrency

The API is designed to support multiple CAN controllers using a `controller` parameter. These functions are *non-reentrant* with respect to the controller: they cannot be called while a call is already taking place for the same controller. In general, do not call these functions from an ISR unless the application has specific measures to prevent concurrent access. In addition, depending on the SPI binding for the target hardware, these functions may not be called while *any* controller is being accessed by another call (for example, if two controllers share the same SPI port then special measures must be added by the application to prevent concurrent access to the SPI port).

The non-reentrant API functions in general are not thread safe. In a multitasking RTOS environment do not call any of the non-reentrant functions concurrently: serialize the calls with RTOS concurrency control mechanisms. For example, define a resource lock for each controller or for each SPI port and then acquire the appropriate lock around each a call to non-reentrant API functions.

2 Basic API usage

2.1 Starting and stopping the CAN controller

The CAN controller is represented by an instance of type `can_controller_t`. The call to initialize a controller is:

```
can_errorcode_t can_setup_controller(can_controller_t *controller,
                                     can_bitrate_t *bitrate,
                                     can_id_filters_t *all_filters,
                                     can_mode_t mode,
                                     uint16_t options)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

The application is responsible to allocating the CAN controller structure before the call and persisting this allocation until the CAN controller is stopped. The application must also bind to the controller a specific hardware interface before making this call. For the CANPico hardware, the call to the function:

```
void mcp25xxfd_spi_bind_canpico(can_interface_t *interface)
```

sets up the details of the SPI pins of the CANPico to the controller-specific `host_interface` member of the `can_controller_t` structure (an example usage is included in `hello_can.c`).

The call will attempt to set up the CAN controller with the requested settings. It will also enable controller interrupts and put the controller into the requested operating mode.

The parameters to the call are defined as follows:

Parameter	Description
`controller`	A pointer to the CAN controller structure allocated by the application. This stores the frame queues and other information.
`bitrate`	A structure defining the desired bit rate (see below).
`all_filters`	A list of CAN ID filters. The constant `CAN_NO_FILTERS` is pre-defined to indicate no ID filters are wanted.
`mode`	The operational mode of the CAN controller (see below).
`options`	Optional behaviour of the CAN controller (see below)

The `bitrate` parameter is a pointer to structure allocated for the purpose of passing bit rate information to this call. The member `profile` defines standardized CAN bit rate profiles as follows:

Profile symbol	Speed (kbit/sec)	Sample point (%)	Notes
`CAN_BITRATE_500K_75`	500	75	
`CAN_BITRATE_250K_75`	250	75	
`CAN_BITRATE_125K_75`	125	75	
`CAN_BITRATE_1M_75`	1000	75	
`CAN_BITRATE_500K_50`	500	50	

Profile symbol	Speed (kbit/sec)	Sample point (%)	Notes
CAN_BITRATE_250K_50	250	50	
CAN_BITRATE_125K_50	125	50	
CAN_BITRATE_1M_50	1000	50	
CAN_BITRATE_2M_50	2000	50	*Non-standard*
CAN_BITRATE_4M_90	4000	90	*Non-standard*
CAN_BITRATE_2_5M_75	2500	75	*Non-standard*
CAN_BITRATE_2M_80	2000	80	*Non-standard*
CAN_BITRATE_500K_875	500	87.5	
CAN_BITRATE_250K_875	250	87.5	
CAN_BITRATE_125K_875	125	87.5	
CAN_BITRATE_1M_875	1000	87.5	
CAN_BITRATE_CUSTOM	Custom bit rate (see below)		

The non-standard bit rates may not be available for all CAN hardware.

If the application sets `profile` to `CAN_BITRATE_CUSTOM` then extra parameters in the `bitrate` parameter structure are used:

Parameter	Description
brp	Baud rate pre-scaler -1. This divides the CAN protocol engine clock by the given value + 1 (so a value of `brp` of 4 is /5). The initial clock rate is hardware-specific (on the Canis Labs CANPico this is 40MHz).
tseg1	CAN TSEG1 -1
tseg2	CAN TSEG2 -1
sjw	CAN SJW -1

Example:

- Setting `brp` of 4 indicates a /5 of the 40MHz clock, giving an 8MHz time quantum clock (125ns per time quantum).
- A `tseg1` value of 10 and a `tseg2` value of 3 gives a total of TSEG1 + TSEG2 of 15. The CAN standard includes 1 for the SYNC_SEG value, giving 16 time quanta per bit, i.e. 2000ns. This is a bit rate of 500kbit/sec.

The `mode` parameter indicates the operation mode as follows:

Mode symbol	Description
CAN_MODE_NORMAL	Normal send and receive of CAN frames
CAN_MODE_LISTEN_ONLY	Controller receives CAN frames but otherwise does not participate in the CAN protocol
CAN_MODE_ACK_ONLY	Controller receives CAN frames and will generate a dominant bit in the ACK field for well-formed frames
CAN_MODE_OFFLINE	Controller does not participate in the CAN protocol

The call to `can_setup_controller()` can be made while the controller is operating, Calling it with `mode` set to `CAN_MODE_OFFLINE` is a way of stopping it communicating any further.

The options parameter is a bit mask with bits set to select specific options. Specific options are defined in the table below, and multiple options can be selected by ORing the masks together.

Option symbol	Description
`CAN_OPTIONS_NONE`	Always defined as 0 so this can be passed as a value when no options are required.
`CANCAN_OPTION_REJECT_REMOTE`	Any remote frames received are automatically discarded and not put into the receive FIFO.
`CAN_OPTION_HARD_RESET`	This is an option for the MCP25xxFD CAN controllers, used to generate an SPI command that does a hard reset on the chip.
`CAN_OPTION_RECV_ERRORS`	CAN error frame events by default are discarded and not put into the receive event FIFO. This option causes them to be stored in the FIFO.
`CAN_OPTION_OPEN_DRAIN`	This is an option for the MCP25xxFD to put the CAN TX controller output pin into open drain mode (it is needed on the CANPico if the CANHack toolkit is used to drive the CAN TX pin on the transceiver via a GPIO port).
`CAN_OPTION_RECORD_TX_EVENTS`	This option causes CAN frame transmit events to be put into the transmit event FIFO.
`CAN_OPTION_REJECT_OVERFLOW`	This option causes receive FIFO overflow events to be discarded and not put into the receive FIFO.

The error codes returned from the `can_setup_controller()` call are given below:

Return value	Description
`CAN_ERC_NO_ERROR`	Always defined as 0 so this can used as succeed / fail Boolean test.
`CAN_ERC_BAD_BITRATE`	The bit rate parameters are not legal.
`CAN_ERC_RANGE`	The number of ID filters supplied exceed the maximum (defined by `CAN_MAX_ID_FILTERS`)
`CAN_ERC_BAD_INIT`	Initialization failed because the CAN controller could not be put into configuration mode or could not be put into an operational mode. This could indicate an SPI connection hardware fault.
`CAN_ERC_BAD_WRITE`	The controller could not be put into TX open drain mode. This could indicate an SPI connection hardware fault.
`CAN_ERC_NO_INTERFACE`	An interface to the controller has not been bound.

The call to stop the controller is defined as:

```
void can_stop_controller(can_controller_t *controller)
```

This function is non-reentrant: do not call it while a call is already taking place.

If this call is made after the controller has been set up then it will disable interrupts from the controller and put the controller into an offline mode. If the controller has not been initialized then this call does nothing.

2.2 Sending CAN frames

The C typedef `can_frame_t` holds CAN frames. They can be created by the application with the following function:

```
void can_make_frame(can_frame_t *frame,
                    bool ide,
                    uint32_t arbitration_id,
                    uint8_t dlc,
                    uint8_t *data,
                    bool remote)
```

The application is responsible for allocating the space for a CAN frame. The `can_make_frame()` call is passed a pointer to this memory and creates a CAN frame from the supplied parameters. These are defined as follows:

Parameter	Description
`frame`	A pointer to the allocated CAN frame structure.
`ide`	Set to `true` if the CAN frame has an extended ID (i.e. 29 bits).
`arbitration_id`	The 11 or 29 bits of the arbitration ID for the frame (29 bits if `ide` is `true` and 11 bits if not).
`dlc`	The DLC of the CAN frame. Only the bottom 4 bits are used.
`data`	The data for the CAN frame is copied into the instance from a pointer to the data source. Only the number of bytes implied by the `dlc` and `remote` parameters are accessed.
`remote`	Set to `true` if the frame is a remote frame. The `data` parameter is ignored.

A frame is queued for transmission using the following call:

```
can_errorcode_t can_send_frame(can_controller_t *controller,
                               can_frame_t *frame,
                               bool fifo)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

The parameter `frame` is a pointer to the frame instance (typically created with the `can_make_frame()` call). The parameter fifo indicates if the frame should be queued into the priority queue (`fifo` is `false`) or into the FIFO transmit queue that feeds into the priority queue.

If two frames in the priority queue have the same ID, then when deciding which to enter into arbitration CAN controllers break the tie arbitrarily and can therefore send frames out of order. This means that CAN frames transmitted as a burst with the same ID (for example, segmented messaging) should go into the FIFO transmit queue to preserve ordering.

Care must in general be taken for frames put into the FIFO transmit queue not to become subject to priority inversion.

Error return codes from the `can_send_frame()` call are given below:

Return value	Description
`CAN_ERC_NO_ERROR`	Frame queued successfully.
`CAN_ERC_BAD_INIT`	The CAN controller has not been set up.
`CAN_ERC_NO_ROOM`	There was no room in the priority or FIFO transmit queue for the frame.

The call `can_is_space()` returns `true` if there will be space for frames to be queued:

```
bool can_is_space(can_controller_t *controller,
                  uint32_t n_frames,
                  bool fifo)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

2.3 Receiving CAN frames

The receive FIFO contains *events*, of which "received CAN frame" is one. An event may also be "seen error frame" or "FIFO overflowed and events were dropped". To receive a CAN frame, it must be extracted from the event.

Receiving an event from the receive FIFO is done with the `can_recv()` call, defined as follows:

```
bool can_recv(can_controller_t *controller,
              can_rx_event_t *event)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

The application is required to allocate the structure of type `can_rx_event_t` for the call to copy an event into. The call returns `true` if the front of the FIFO queue was removed and copied to the structure, and `false` otherwise (reasons for `false` being returned include no events in the queue and the CAN controller not being set up).

To see if the event contains a frame, the call `can_event_is_frame()` is used, defined as:

```
bool can_event_is_frame(can_rx_event_t *event)
```

If the call returns `true` then the CAN frame can be extracted with:

```
can_frame_t *can_event_get_frame(can_rx_event_t *event)
```

This returns a pointer to the CAN frame structure contained within the event. The timestamp of the received CAN frame can be obtained by asking for the timestamp of the event with the `can_event_get_timestamp()` call:

```
uint32_t can_event_get_timestamp(can_rx_event_t *event)
```

The timestamp is a 32-bit integer measuring the time in microseconds for the CAN start-of-frame (SOF) of the received frame.

The contents of a CAN frame can be unpacked with various functions that are passed a pointer to the `can_frame_t` structure:

Function	Description
`can_frame_is_extended`	Returns `true` if the CAN ID of the frame is 29 bits long and false if 11 bits.
`can_frame_is_remote`	Returns `true` if the CAN frame is a remote frame.
`can_frame_get_arbitration_id`	Returns the integer value of the arbitration ID (either 11 or 29 bits).
`can_frame_get_data`	Returns a pointer to the bytes of the data.
`can_frame_get_data_len`	Returns the length of the CAN frame data.
`can_frame_get_dlc`	Returns the DLC value of the CAN frame.
`can_frame_get_id_filter`	Returns the index of the CAN ID filter that caused the frame to be accepted (0 if ID filtering not enabled).

2.4 Vectoring the CAN controller interrupt ISR

The vectoring of an interrupt to the CAN drivers may require application support, depending on the CAN controller and the host microcontroller hardware.

The MCP25xxFD CAN controller requires a level-sensitive interrupt from a GPIO pin on the host microcontroller. The binding of these drivers to the RP2040 sets up the GPIO pin appropriately when the CAN controller is set up, but on the RP2040 the GPIO interrupt vector is shared for all interrupts, and if there are other GPIO interrupts then the application must call the ISRs appropriately. The MCP25xxFD drivers define the following function to be called:

```
void mcp25xxfd_irq_handler(can_controller_t *controller)
```

On the RP2040 the application is responsible for calling this (on other host microcontrollers the binding may include this vectoring) with the parameter set to the bound controller structure.

3 A Hello World Example

3.1 Building the example

The C SDK is distributed with a simple example for the Raspberry Pi Pico with the Canis Labs CANPico board in the following files:

File	Description
`hello_can.c`	Example file
`CMakeLists.txt`	Build file for `cmake` for the Raspberry Pi Pico

The program uses the standard Raspberry Pi Pico C SDK. This is installed by cloning the repository, for example:

```
$ git clone git@github.com:raspberrypi/pico-sdk.git
```

The environment variable `PICO_SDK_PATH` must be set to point to where this is cloned. The example program is built using `cmake`:

```
$ cmake CMakeLists.txt
```

And then make:

```
$ make
```

The file `hello_can.uf2` is a firmware image for the example program.

Support for the Pico W is also provided: see `CMakeLists.txt` for details of how to build for the Pico W.

3.2 Deploying the example

This can be programmed into the Raspberry Pi Pico by pressing the selector button when plugging in the board via USB: it mounts as a removable disk and `hello_can.uf2` is then copied to the disk.

3.3 Running the example

The example sends a 5 byte CAN frame every half second, flashing the LED each time. In addition, it will print out frames it receives (up to a maximum of ten per half second cycle).

On Linux, minicom can be used to display the text on the serial port:

```
$ minicom -b115200 -o -D /dev/ttyACM0
```

(Assuming the serial port to the target is `/dev/ttyACM0`)

The terminal will very likely display something like this at every blink:

```
Frames queued OK=30
Frames queued OK=31
Frames queued OK=32
CAN send error: 4, sent=32
Frames queued OK=32
CAN send error: 4, sent=32
Frames queued OK=32
CAN send error: 4, sent=32
Frames queued OK=32
```

The most common error with CAN is to connect a controller to an empty bus. Here, the frames are being put into the CAN controller, but no frames are being sent out – the frame doesn't get acknowledged because there is nothing else on the bus, so there is an error raised and the frame is retransmitted. The priority queue is limited to 32 frames so eventually overflows, resulting in an API error returned.

If another CAN controller is connected to the CAN bus, then the frames are transmitted normally, and the queue does not overflow. And incoming frames are reported like this:

```
Frames queued OK=163
Frames queued OK=164
Frames queued OK=165
0x100 68656c6c6f (82324274)
0x101 776f726c64 (82324452)
Frames queued OK=166
Frames queued OK=167
Frames queued OK=168
```

The first number is the CAN ID, the second is the payload, and the third is the timestamp.

4 Advanced API use

4.1 CAN ID filtering

The CAN protocol engine of a CAN controller receives every frame but in most applications the host software does not need to see all these frames. Instead, ID acceptance filters are used to filter out unwanted frames.

The symbol `CAN_MAX_ID_FILTERS` is defined as the number of ID filters available to the application (on the MCP25xxFD there are 32 ID filters). An ID filter consists of two words that correspond to the arbitration ID and are applied in hardware to a CAN ID (to either an extended 29-bit ID or a standard 11-bit ID).

The mask contains "must match" and "don't care" bits for the corresponding arbitration ID bits, and the match contains the value in the case of "must match" in the mask. An ID filter is represented by the type `can_id_filter_t` and there are several creation functions:

Function	Description
`can_make_id_filter`	Create a filter for a given CAN ID
`can_make_id_filter_all`	Create a filter that allows all CAN IDs through
`can_make_id_filter_masked`	Create a filter from a mask/match pair
`can_make_id_filter_disabled`	Create a non-enabled filter

ID filters are passed to the `can_setup_controller()` function via a list of filters described by the typedef `can_id_filters_t`:

```
typedef struct {
    can_id_filter_t *filter_list;
    uint8_t n_filters;
} can_id_filters_t;
```

The array can be discarded once the call is made: the call it will copy the information into the CAN controller, which will persist until the next call. The placement in the list of a given filter is important: when a CAN frame is received, the index of the ID filter that led to it being accepted is stored in the CAN frame. This filter index is returned for a given frame with:

```
uint8_t can_frame_get_id_filter(can_frame_t *frame)
```

The filter index be used for quickly identifying an incoming frame.

An ID filter can be created to accept a specified CAN ID:

```
void can_make_id_filter(can_id_filter_t *filter, can_id_t canid)
```

A CAN ID can be obtained by the `can_make_id()` call:

```
can_id_t can_make_id(bool extended, uint32_t arbitration_id)
```

An ID filter can also be created from a specific mask and match value:

```
void can_make_id_filter_masked(can_id_filter_t *filter,
                               bool ide,
                               uint32_t arbitration_id_match,
                               uint32_t arbitration_id_mask)
```

The parameter `ide` indicates if the filter is for an extended (29-bit) or standard (11-bit) CAN ID. In a J1939 system this would typically be used to set an ID filter for a wanted PGN but mask out the other fields (such as priority).

4.2 Frame transmission FIFO

Sometimes the application needs to know when a frame has been transmitted on the bus (for example, to measure frame latencies). The API has support for this with a FIFO that stores frame transmission events.

The storing of frame transmit events is optional functionality and only takes place if the CAN controller setup included the option `CAN_OPTION_RECORD_TX_EVENTS`. A frame transmission event is contained within the typedef structure `can_tx_event_t` and is returned by the receive call:

```
bool can_recv_tx_event(can_controller_t *controller,
                       can_tx_event_t *event)
```

This function is non-reentrant: do not call it while a call is already taking place for the same controller.

When the function returns `true` then the event structure has been filled in (and returns `false` if the FIFO is empty). The number of events in the FIFO can also be requested:

```
uint32_t can_recv_tx_events_pending(can_controller_t *controller)
```

This function is non-reentrant: do not call it while a call is already taking place for the same controller.

The events stored in the FIFO are one of two types: *frame* and *overflow*. The timestamp of any type of event is returned with:

```
uint32_t can_tx_event_get_timestamp(can_tx_event_t *event)
```

The following call indicates if the event represents the transmission of a frame.

```
bool can_tx_event_is_frame(can_tx_event_t *event)
```

Frame transmission events contain a user reference of type `can_uref_t` that is associated with a CAN frame by the application and can later be used by the application to associate the transmitted frame with the source of the frame within the application. The user reference is associated with a frame with the following call:

```
void can_frame_set_uref(can_frame_t *frame, void *ref)
```

The reference is a generic type that can be stored in a pointer-sized type (typically 32-bits). This could be an index into an array of frame control blocks, or it could be a pointer to some other structure. The MicroPython firmware implementing the CAN API uses

this reference to refer to the `CANFrame` instance that represents the CAN frame. The user reference for the transmitted frame can be obtained with:

```
can_uref_t can_tx_event_get_uref(can_tx_event_t *event)
```

4.3 ISR call-backs

To process events very quickly, the API includes the following call-backs for events, called from interrupt level:

- CAN frame transmitted
- CAN frame received
- CAN error seen

Because the call-backs are made at ISR level they should execute quickly. On the RP2040 with XIP (eXecute-In-Place) flash memory the call-back code should placed in RAM to avoid CPU execution priority inversion[1].

The CAN frame transmitted call-back is defined as follows:

```
void can_isr_callback_frame_tx(can_uref_t uref,
                               uint32_t timestamp)
```

The call is made from the CAN controller receive handler, which may be vectored through a single handler, depending on the target hardware (the MCP25xxFD drivers have a single interrupt handler). The parameters include the user reference for the transmitted frame and the timestamp of the transmission (defined as the SOF for the CAN frame on the MCP25xxFD – other CAN controller hardware may have a different defined timestamp point).

A 'weak' call-back handler (that does nothing) is included in the driver code so the application need do nothing more than define this function. The call-back is made even if the `CAN_OPTION_RECORD_TX_EVENTS` is not selected and even if there is no space in the transmit FIFO.

The CAN frame received call-back is defined as follows:

```
void can_isr_callback_frame_rx(can_frame_t *frame,
                               uint32_t timestamp)
```

A 'weak' call-back handler (that does nothing) is included in the driver code so the application need do nothing more than define this function. The call-back is made even if there is no space in the receive FIFO. **The lifetime of the storage for `frame` is limited to the duration of the call-back**: the handler must copy out the relevant information and store it somewhere safely.

The CAN error call-back is defined as follows:

```
void can_isr_callback_error(can_error_t error,
                            uint32_t timestamp)
```

[1] See the blog post by Dr. Ken Tindell: https://kentindell.github.io/2021/03/05/pico-priority-inversion/

The call is made with an indication of the error seen and a timestamp for the error. A 'weak' call-back handler (that does nothing) is included in the driver code so the application need do nothing more than define this function. The call-back is made even if there is no space in the receive FIFO to store the error and even if the option `CAN_OPTION_RECV_ERRORS` is not selected. The error is encoded in type `can_error_t` and there are several standard functions to extract the details of the error. These are covered later.

4.4 FIFO overflow events

When an event is to be added to a FIFO there may be no room for it. In this case, an overflow event is placed in the FIFO that records the start of the overflow and keeps track of how many other events have been lost (this can be when building a CAN logging tool).

An overflow event is identified by the function calls `can_event_is_overflow()` and `can_tx_event_is_overflow()` (for the receive and transmit FIFOs respectively).

The receive overflow event is obtained with:

```
can_rx_overflow_event_t *can_event_get_overflow(can_rx_event_t *event)
```

The number of error frames dropped (if errors are being logged into the receive FIFO) is given by:

```
uint32_t can_rx_overflow_get_error_cnt(can_rx_overflow_event_t *over-
flow)
```

The number of CAN frames dropped is given by:

```
uint32_t can_rx_overflow_get_frame_cnt(can_rx_overflow_event_t *over-
flow)
```

For transmit overflow events, the number of frame transmit events dropped is given by:

```
uint32_t can_tx_event_get_overflow_cnt(can_tx_event_t *event)
```

4.5 Error events

CAN error events are optionally stored in the receive event queue (if the option `CAN_OPTION_RECV_ERRORS` is selected). If the event is an error event (i.e. `can_event_is_error()` returns true) then the error is returned from the call:

```
can_error_t *can_event_get_error(can_rx_event_t *event)
```

An error is of type `can_error_t` and there are standard functions to test for the type of error:

Function	Description
`can_error_is_crc`	CAN CRC error
`can_error_is_stuff`	CAN stuff error
`can_error_is_form`	CAN form error
`can_error_is_ack`	CAN acknowledge error
`can_error_is_bit1`	CAN bit error (sent a 1 but received a 0)
`can_error_is_bit0`	CAN bit error (sent a 0 but received a 1)

Specific CAN controller hardware may not be able to report the two types of bit error, in which case the driver will report both types as a sent 1/received 0 error.

4.6 Controller status

The CAN protocol defines the status of a CAN controller: whether it is operating normally or if it is in Error Passive or Bus Off states. There are two error counters that define these states: the Transmit Error Counter (TEC) and the Receive Error Counter (REC). The controller status is defined by the typedef `can_status_t` and obtained by:

```
can_status_t can_get_status(can_controller_t *controller)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

There are standard functions defined to extract TEC and REC and the controller status:

Function	Description
`can_status_is_bus_off`	Returns true if the controller is Bus Off
`can_status_is_error_passive`	Returns true if the controller is Error Passive
`can_status_is_error_warn`	Returns true if the controller is in a Warn state
`can_status_get_tec`	Return the TEC
`can_status_get_rec`	Return the REC
`can_status_request_recover`	Request the controller recover from Bus Off

NB: the MCP25xxFD CAN controller will automatically recover from Bus Off.

4.7 To and from byte representations of events

The CAN API includes functions to convert objects to and from bytes. This is particularly useful when tunnelling CAN frames from another network. The following function creates a CAN frame from a block of 19 bytes:

```
void can_make_frame_from_bytes(can_frame_t *frame, uint8_t *src)
```

The bytes describing the frame have the following format:

Byte	Bits	Description
0	0	If 1 then a CAN frame is remote
	7:1	*Reserved* (must be set to 0)
1	3:0	DLC
	7:4	*Reserved* (must be set to 0)
2	7:0	*Reserved* (must be set to 0)
3:6	31:0	Application tag
7:10	31:30	*Reserved* (must be set to 0)
	29	If 1 then extended 29-bit ID else 11-bit ID
	28:0	Arbitration ID (29 bits if extended, 11 otherwise)
11:18	7:0	Data bytes

The 32-bit words for the application tag and ID are in big-endian format (i.e. lower byte address is upper bits). The application tag is placed in the user reference for the frame.

The following function creates a block of bytes from a receive event:

```
uint32_t can_recv_as_bytes(can_controller_t *controller,
                           uint8_t *dest,
                           size_t n_bytes)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

The function returns 0 if the receive FIFO is empty or 19. The parameter `dest` points to where the bytes should be written. The parameter `n_bytes` is the size of the block of bytes; it must be at least 19 bytes in size for a FIFO receive operation to take place.

The format of the bytes describing the event is as follows:

Byte	Bits	Description
0	1:0	Event type: 0: *Reserved* 1: CAN frame 2: Overflow 3: CAN error frame
	7:2	*Reserved* (will be set to 0)
1:4	31:0	Timestamp (in microseconds)

If the event is a CAN frame, then bytes 5 to 18 are defined as:

Byte	Bits	Description
5	3:0	DLC
	7:4	*Reserved* (will be set to 0)
6	7:0	ID filter index
7:10	31:30	*Reserved* (will be set to 0)
	29	If 1 then extended 29-bit ID else 11-bit ID
	28:0	Arbitration ID (29 bits if extended, 11 otherwise)
11:18	7:0	Data bytes

If the event is a CAN error frame, then bytes 5 to 18 are defined as:

Byte	Bits	Description
5:6		*Reserved*
7:10	31:22	*Reserved*
	21	CRC error
	20	Stuff error
	19	Form error
	18	Ack error
	17	Bit error (sent 1 received 0)
	16	Bit error (sent 0 received 1)
	15:0	*Reserved*
11:18		*Reserved*

If the event is an overflow event, then bytes 5 to 18 are defined as:

Byte	Bits	Description
5:6		*Reserved*
7:10	31:0	Dropped frames
11:14	31:0	Dropped errors
15:18		*Reserved*

The following function creates a block of bytes from a transmit event:

```
uint32_t can_recv_tx_event_as_bytes(can_controller_t *controller,
                                    uint8_t *dest,
                                    size_t n_bytes);
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

The parameter `n_bytes` is the size of the buffer pointed to by `dest`, and the function returns the number of bytes filled. The format for the bytes is as follows:

Byte	Bits	Description	
0	1:0	Event type: 0: Transmission of frame 1: *Reserved* 2: Overflow 3: *Reserved*	
	7:2	*Reserved* (will be set to 0)	
1:4	31:0	Application tag	Overflow count
5:8	31:0	Timestamp (in microseconds)	

The 32-bit words for the application, overflow count and ID are in big-endian format (i.e. lower byte address is upper bits). Bytes 1:4 are the application tag if the event is a frame transmission, and the number of frame transmission events dropped if the event is an overflow.

The application tag is derived from the user reference using an application-defined call-back function:

```
uint32_t can_isr_callback_uref(can_uref_t uref)
```

A 'weak' call-back handler (that returns 0) is included in the driver code so the application need do nothing more than define this function. This function is called with interrupts disabled so should run quickly (on the RP2040 this means being located in RAM rather than XIP flash).

4.8 Time

The CAN controller runs a 32-bit free-running timer counting microseconds. The value of this can be obtained with:

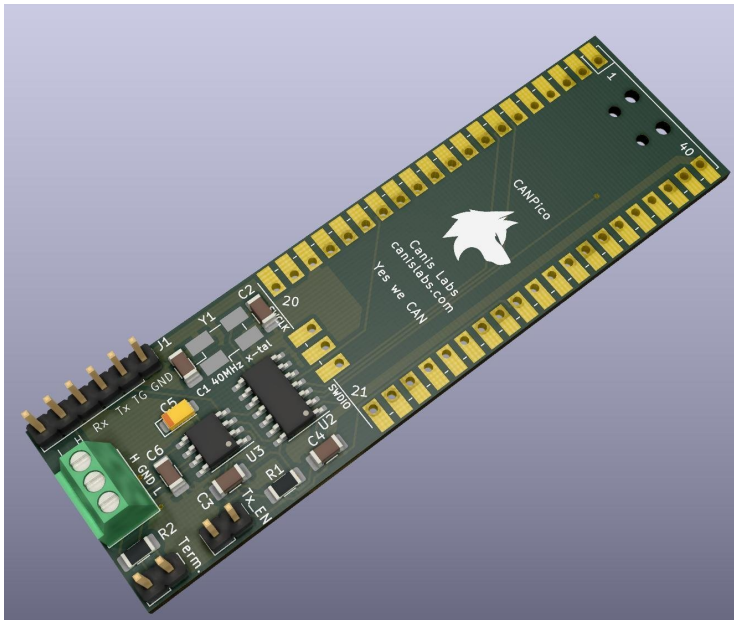
```
uint32_t can_get_time(can_controller_t *controller)
```

This function is non-reentrant: do not call it while a call is already taking place to the same controller.

The time returned will always lag the current time because the time taken to obtain the call may be significant (particularly for an external CAN controller connected via SPI).

5 CAN Pico

The CANPico is a 'sock' board designed to connect a Raspberry Pi Pico board to a CAN bus. A Pico board is soldered down on to the upper area of the board and the lower area of the board contains the CAN connectors.



CAN bus is a protocol used in many applications, from trucks to buses to aircraft to agricultural equipment to medical devices to construction equipment and even spacecraft but its most common use is in cars.

WARNING. Connecting the CANPico directly to a vehicle CAN bus comes with risk and should not be undertaken without understanding this risk. Grounding a vehicle chassis through the USB port of a Pico connected to a laptop, PC or USB hub powered from the mains may cause permanent damage to the vehicle's electronics and/or the Raspberry Pi Pico and devices it is connected to. The CAN transceiver will tolerate a ground potential difference ("ground offset") of between -2V/+7V. Connecting pin 2 of the screw terminal to the target system's ground will establish a common ground reference. The CAN bus must be properly terminated (with 120Ω resistors at either end). If the CAN bus is already terminated at both ends then the termination jumper on the CANPico should not be closed. In addition, causing disruption to a vehicle's CAN bus traffic may be interpreted by the vehicle fault management systems as a hardware fault and cause the vehicle to permanently shut down CAN communications with consequent loss of functionality.

6 History

6.1 Issue 01 2022-08-01

- First issue

6.2 Issue 02 2023-03-06

- Changed API to permit multiple CAN controllers of the same type (by adding a `controller` parameter)
- Added the `mcp25xxfd_spi_bind_canpico()` API call to bind a specific SPI port to an MCP25xxFD CAN controller.
- Added error codes `CAN_ERC_BAD_WRITE` (for failure to set the MCP25xxFD into TX open drain mode) and `CAN_ERC_NO_INTERFACE` (to indicate that the host interface has not been bound before attempting to set up the CAN controller).
- Renamed 2517FD to 25xxFD (and `mcp2517fd`.. to `mcp25xxfd`..) to reflect the support for the MCP25xxFD family of Microchip CAN devices (i.e. MCP2517FD, MCP2518FD, MCP251863).