CAN driver API - migration from Classical CAN to CAN FD

Uwe Koppe, Johann Tiderko, MicroControl GmbH & Co. KG

The new features of CAN FD also have an impact on the CAN driver layer. Functions have to be provided to switch the bit-rate, handle longer data frames and evaluate the error indicator. In addition, compatibility to the existing API should be observed. The document gives an overview of existing CAN drivers for AUTOSAR[1], can4linux, CANpie FD, CMSIS as well as SocketCAN and their approaches to support CAN FD. It also investigates the complexity, use cases and documentation of the five different CAN drivers.

[1] The word AUTOSAR and the AUTOSAR logo are registered trademarks.

In contrast to other network systems, there has not been a development for a standardized CAN bus application programming interface (API), like e.g. Berkeley sockets for Ethernet socket access. In fact, we can observe that silicon manufacturers or PC interface board manufacturers provide unique hardware abstraction layer (HAL) libraries for supporting their interfaces. The user benefit is of course a fast development of the target application, however, it is rather complex to port this application to another hardware architecture. The discussed APIs in this paper have been selected because of the gained work experience and the freely available documentation. In order to achieve comparison criteria for the CAN APIs the specifications CiA 601-2 [1] (CAN controller interface specification) and CiA 603 [2] (Requirements for network time management) are introduced briefly. These specifications have been developed within the CAN in Automation e.V. by the IG (interest group) CAN FD.

The CiA 601-2 specifies the interface between protocol controller (i.e. CAN FD) and the host controller in order to reduce the effort for adapting the low-level CAN driver. A CAN controller shall support the finite state automaton (FSA) depicted in figure 1. In addition, it is advisable to support the states self-test, receive-only and low-power.

The CAN controller may have either dedicated messages buffers (a buffer that holds only one specific CAN message) or general message buffers, which are organized as single buffers, FIFO queue or priority queue.

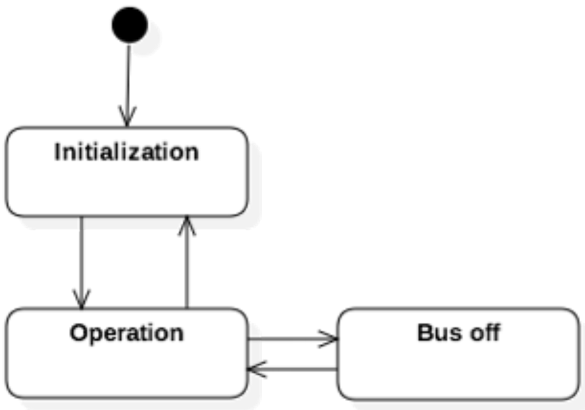

Figure 1: Minimum FSA for CAN

The number of message buffers is not specified, current CAN IPs support up to 128 message buffers. With regard to message reception an acceptance filtering for message ID reception shall be configurable (i.e. filter mask for identifier). For message transmission it is possible to support internal arbitration, a FIFO or a buffer number to determine the transmit sequence. For each transmit buffer it shall be possible to set the FDF bit, respectively BRS bit, individually. A message marker is an optional feature that allows to set a „label“ for a message. The message marker is stored in an event FIFO after transmission which allows the user to determine the transmission order of multiple messages. The CAN status interface shall give information about the current CAN error state as well as transmission and reception state. In contrast to classic CAN the number of parameters and the possible parameter ranges have changed significantly. For nominal bit-timing as well as four data bit-timing it is recommended to support eight parameters in total.

The CiA 603 focuses on AUTOSAR specification which supports network synchronization. Hence, there is a requirement on hardware-based time-stamping for message reception. The resolution of the time-stamp shall be in the range from 1 ns to 1 µs, a time-stamp register should have a width of 32 bit. This allows to measure total time spans in the range from 4,2 seconds (1 ns resolution) to 1,2 hour (1 µs resolution).

With regard to hardware-independent design of an application it should be possible to query the supported capabilities of a CAN hardware. A self-contained event support for message reception and transmission as well as CAN state changes allows an OS-independent driver API. In order to avoid error-prone bit-shifting operations in CAN message structures it is recommended to have functions or macros for configuration and evaluation of CAN messages.

Apart from these technical topics, there are some soft facts which have to be taken into account for comparison. First of all there should be a detailed user specification, both explaining the API and giving examples. Experience in various projects has shown that it should be available as PDF document as well as online (HTML) version. Furthermore, a detailed test specification ideally in conjunction with ready-to-use test cases, are of great advantage at the end of the driver development phase. In addition, code style and static code checking (e.g. splint, MISRA-C) enable the development of robust, compiler-independent code. Last but not least the API license has to be taken into account.

All discussed criteria have been listed in tables 1 and 2, the five APIs have been evaluated according to these. The code example for each API depicts the effort to send a CAN FD standard frame (FBFF format) using an identifier value of 100 and a DLC value of 9 (i.e. 12 byte payload).

AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) is a worldwide development partnership of automotive interested parties founded in 2003. It pursues the objective of creating and establishing an open and standardized software architecture for automotive electronic control units. The AUTOSAR development partnership was formed in July 2003 by BMW, Bosch, Continental, DaimlerChrysler, Siemens VDO and Volkswagen [3]. The current version 4.3 is available for AUTOSAR partners, support for CAN FD was introduced with version 4.2.1.

The AUTORSAR specification for CAN FD [4] is divided into seven documents and defines a hardware abstraction layer covering physical layer, data link layer, transport layer and requirement specification.

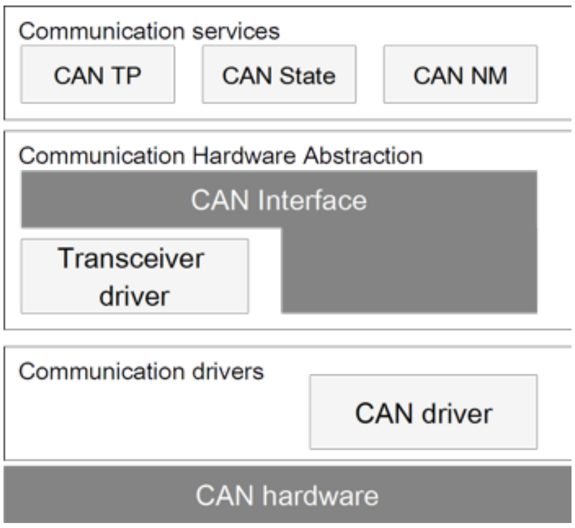

Figure 2: AUTOSAR services for CAN

This leads to a rather complex API with more than 100 functions. The API defines structures and function calls, but does not provide symbol definitions or bit-masks as the other introduced CAN drivers. Since AUTOSAR is used in conjunction with a configuration tool, the software designer is typically not engaged in this complexity. The API does not support the self-test or the receive-only mode of the FSA, also a marker is not provided inside the message structure. Due to the static configuration a capability request function is not required, the same applies to an API for CAN message configuration.

The AUTOSAR code is provided by various vendors as MISRA-C compliant solution under a commercial license. Figure 3 shows an example code for transmission of a CAN FD frame.

```
void WriteCanFdMessage()
{
  Can_PduType frame;
  uint8_t data[12];
  /* setup data */
  data[0] = 0x12;
  ..
  /* two most significant
  ** bits specify the frame
  ** type
  ** ID = 100, FBFF
  */
  frame.id = 0x40000064;
  frame.length = 12;
  frame.sdu = &data[0];
  Can_Write(&canHardware, &frame);
}
```

Figure 3: AUTOSAR example code

Can4linux

The can4linux API is an open source CAN device driver for Linux. The development started in the mid 1990s for the Philips 82C200 CAN controller stand alone chip on a ISA Board AT-CAN-MINI. In 1995, the first version was created to use the CAN bus with Linux for laboratory automation as a project of the Linux Lab Project at FU Berlin [5]. The current version 4.2 is hosted on SourceForge and available under GPL version 2 license. The driver uses the deprecated Linux device interface (/dev/can) for accessing the kernel driver from user space applications. The API supports the concept of message buffers which is rarely documented and lacks dedicated kernel functions for that purpose. The majority of the available CAN drivers support a single FIFO for message reception and transmission each. For message transmission this approach causes the risk of inner priority inversion. A time-stamp is supported with a resolution of 1 µs. Since the user space API is rather simple using 5 common functions (open / close / read / write / ioctl) all CAN status information as well as bit-timing is done via the ioctl() function. For programming of the kernel space driver 35 functions are provided. There is no possibility to configure all required CAN FD bit-timing parameters with the actual version of can4linux. The API lacks functionality for accessing the message structure, test cases and coding style guidelines. Figure 4 an shows example code for transmission of a CAN FD frame.

```
void WriteCanFdMessage()
{
  int fd;
  canmsg_t frame;
  /* open CAN interface  */
  fd = open(„/dev/can0“, O_RDWR);
  frame.flags = MSG_CANFD;
  frame.id = 100;
  frame.length = 12;
  /* setup data          */
  frame.data[0] = 0x12;
  ..
  /* count is number of frames */
  write(fd, &frame, 1);
  close(fd);
}
```

Figure 4: Can4linux example code

CANpie FD

CANpie FD is an open source project and pursues the objective of creating and establishing an open and standardized software API for access to the CAN bus. The current version 3.0 is hosted on Github and available under LGPL version 3 license [8].

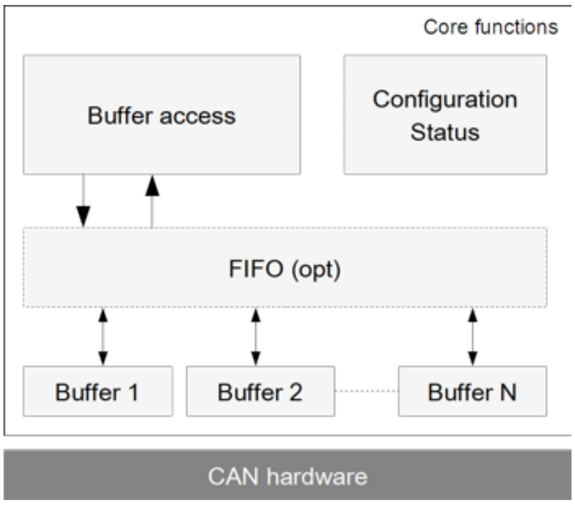

Figure 5: CANpie FD services

The driver uses a C interface for microcontroller access and a C++ interface (using Qt 5) for OS independent access to CAN interface boards.

The API supports the concept of message buffers with a total limit of 255 buffers. It is also possible to connect a FIFO to any available buffer for both transfer directions. Each buffer supports an acceptance mask for message reception. The capabilities of the CAN hardware (CAN FD, number of buffers, maximum bit-rates, etc.) can be queried by a function. Also a message time-stamp is supported with a resolution of 1 ns. The API provides functionality for CAN message configuration and evaluation. The CANpie FD code is MISRA-C compliant, the test specification and test cases are hosted on Github.

Figure 6 shows an example code for transmission of a CAN FD frame.

```
void WriteCanFdMessage()
{
  CpCanMsg_ts frame;
  uint32_t size;
  CpMsgClear(&frame,
             CP_MSG_FORMAT_FBFF);
  CpMsgSetIdentifier(&frame, 100);
  CpMsgSetDlc(&frame, 9);
  /* setup data */
  CpMsgSetData(&frame, 0, 0x12);
  ..
  /* size is number of frames */
  size = 1;
  CpCoreFifoWrite(&canInterface,
                  eCP_BUFFER_2,
                  &frame, &size);
}
```

Figure 6: CANpie FD example code

CMSIS

The CMSIS-Driver (Cortex Microcontroller Software Interface Standard) specification is a software API that describes peripheral driver interfaces for middleware stacks and user applications. The actual version 2.04 is hosted on GitHub and available under Apache 2.0 license [11].

Due to its close relation to the CMSIS core functionality the application area is limited to ARM Cortex microcontrollers. The API supports the concept of message buffers and provides FIFO functionality. The number of supported message buffers is only limited by hardware. Similar to CANpie FD the API provides the possibility to request hardware capabilities. Each buffer supports an acceptance mask. The CMSIS driver documentation [10] refers to a driver validation software, however this is not publicly available.

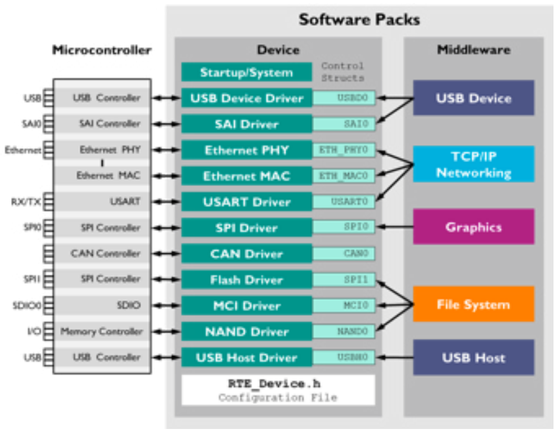

Figure 7: CMSIS driver structure

Although the complete functionality for CAN FD message handling is available, the CMSIS driver API does not provide functionality to configure the data bit-rate.

Figure 8 shows an example code for transmission of a CAN FD frame.

```
void WriteCanFdMessage()
{
  uint8_t  tx_data[12];
  ARM_CAN_MSG_INFO tx_msg_info;

  /* configure buffer number 1 */
  ptrCAN->ObjectConfigure(1U,
             ARM_CAN_OBJ_TX);

  /* clear message info         */
  memset(&tx_msg_info, 0U,
       sizeof(ARM_CAN_MSG_INFO));
  tx_msg_info.id =  \
       ARM_CAN_STANDARD_ID(100U);
  tx_msg_info.edl = 1U;
  /* setup data                 */
  tx_data[0] = 0x12;
  ..
  ptrCAN->MessageSend(tx_obj_idx,
                  &tx_msg_info,
                  tx_data,
                  12U);
}
```

Figure 8: CMSIS driver example code

SocketCAN

SocketCAN is a set of open source CAN drivers and a networking stack contributed by Volkswagen Research to the Linux kernel [12]. Since there is no version information in the SocketCAN header files the application programmer can only use the Linux kernel header version to distinguish between different implementations.

The driver uses the Linux socket interface for accessing the kernel driver from user space applications. The API does not support the concept of message buffers, CAN messages are transferred opening any number of sockets to the kernel network driver. For message transmission this approach causes the risk of inner priority inversion. A time-stamp is not supported, also a message marker is missing. The API uses different structures (and structure elements) for classic CAN and CAN FD. Both, user space application and kernel space driver, distinguish the frame type via the structure size (MTU). This means it is not possible to write (or read) more than one frame.

In order to avoid polling it is possible to install an event handler using the select function in the application. The bit-rate setting is done via the Linux ifconfig command, an additional parameter enables configuration of the data bit-rate. The bit-rate configuration requires root access to the operating system, which limits the field of applications.

Figure 9 shows an example code for transmission of a CAN FD frame.

```
void WriteCanFdMessage()
{
  int s;
  int nbytes;
  struct sockaddr_can addr;
  struct canfd_frame frame;
  struct ifreq ifr;

  const char *ifname = „vcan0";

  if((s = socket(PF_CAN,
               SOCK_RAW,
               CAN_RAW)) < 0)
  {
    perror(„opening failed");
    returns;
  }
  strcpy(ifr.ifr_name, ifname);
  ioctl(s, SIOCGIFINDEX, &ifr);

  addr.can_family = AF_CAN;
  addr.can_ifindex =  \
             ifr.ifr_ifindex;

  printf(„%s at index %d\n",
          ifname,
          ifr.ifr_ifindex);

  if(bind(s,
         (struct sockaddr *)&addr,
          sizeof(addr)) < 0)
  {
    perror(„bind failed");
    return;
  }

  frame.can_id  = 0x123;
  frame.len = 12;
  /* setup data                */
  frame.data[0] = 0x12;


  /* nbytes (MTU) defines classic
  ** frame or CAN FD frame
  */
  nbytes = write(s,
               &frame,
        sizeof(struct can_frame));

  printf(„Wrote %d bytes\n",
          nbytes);

}
```

Figure 9: SocketCAN example code

Table 1: CAN FD API comparison

	AUTOSAR	can4linux	CANpie FD	CMSIS	Socketcan
Version	4.3	4.2	3.0	2.04	-
Device / OS support	Microcontroller	Linux	Microcontroller, Win/Linux/Mac	ARM based microcontroller	Linux
License	Commercial	GPL version 2	LGPL version 3	Apache 2.0	GPL version 2
User documentation	PDF	HTML	PDF / HTML	HTML	Text file
Test documentation	PDF	-	PDF / HTML	o	-
Test cases	+	-	+	o	-
Test code	+	-	+	o	o
Code compliance	MISRA-C	-	MISRA-C	CMSIS	-
API function count	101	5 (35)	21	17	6

Table 2: Technical comparison

	AUTOSAR	can4linux	CANpie FD	CMSIS	Socketcan
FSA: general	+	+	+	+	+
FSA: self test	-	+	+	+	-
FSA: receive-only	-	+	+	+	+
FSA: low-power	+	-	+	-	-
Message buffer	+	+	+ (255 max.)	+	-
Acceptance filtering	o	+(1)	+ (255 max.)	+	+ (SW)
Message marker	-	-	+	o	-
Transmit sequence	FIFO / buffer	FIFO	FIFO / buffer	FIFO / buffer	FIFO
Capability request	-	-	+	+	-
Event support	+	-	+	+	o
CAN status	+	o	+	+	+
CAN message API	-	-	+	-	-
CAN message FDF	+	+	+	+	+
CAN message BRS	+	-	+	+	+
CAN message ESI	+	+	+	+	+
CAN time-stamp	o	1 μs	1 ns .. 1 μs	-	-

Summary

Tables 1 and 2 list all criteria and compare the five selected CAN APIs.

Within the table, the symbol „+“ implies a feature is fully supported, the symbol „-“ means it is not supported. The symbol „o“ denotes a partial support.

Conclusion

From the point of a software application engineer the selected CAN API should be backwards compatible but also allow the complete access to the world of CAN FD. This is especially important for the next years, where classic CAN and CAN FD will exist side-by-side. Higher layer protocols (HLP), for example CANopen, will be used in mature classic CAN applications, new applications might use CANopen FD. The protocol stack should support both HLP version (in order to avoid two firmware versions), hence e.g. a function to setup a bit-rate should provide the required parameters during run-time, not during compile-time. No matter what CAN API you are currently using, the question is: „Does it provide a smooth migration path?“

References

[1] [1] CiA 601, Part 2: CAN controller interface recommendation, CAN in Automation e.V.
[2] CiA 603, CAN Frame time-stamping, CAN in Automation e.V.
[3] https://en.wikipedia.org/wiki/AUTOSAR
[4] AUTOSAR Specification of CAN Driver, Version 4.3.0, www.autosar.org
[5] https://en.wikipedia.org/wiki/Can4linux
[6] https://sourceforge.net/projects/can4linux/
[7] http://www.can-wiki.info/can4linux/man/index.html
[8] https://en.wikipedia.org/wiki/CANpie
[9] https://github.com/canpie/
[10] http://www.keil.com/pack/doc/CMSIS/Driver/html/index.html
[11] https://github.com/ARM-software/CMSIS_5
[12] https://en.wikipedia.org/wiki/SocketCAN
[13] https://www.kernel.org/doc/Documentation/networking/can.txt
[14] Patent DE 10 2004 020 880 A1, Schnittstelle zur Kommunikation zwischen Fahrzeug-Applikationen und Fahrzeug-Bussystemen

Uwe Koppe
MicroControl GmbH & Co. KG
Junkersring 23
DE-53844 Troisdorf
koppe@microcontrol.net
www.microcontrol.net

Johann Tiderko
MicroControl GmbH & Co. KG
Junkersring 23
DE-53844 Troisdorf
johann.tiderko@microcontrol.net
www.microcontrol.net