

# Návrh subsystému CAN FD sběrnice pro space grade real-time exekutivu RTEMS

**Linux Days 2025**

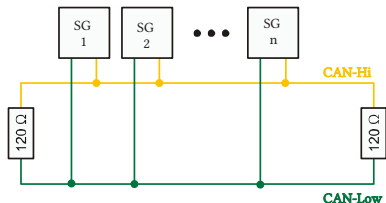
5. 10. 2025

Michal Lenc & Pavel Píša

# **1 Komunikační sběrnice CAN**

Controller Area Network (CAN, CAN bus) je sériová datová sběrnice původně vyvinutá v 80. letech firmou Bosch.

- silně využívaná v automobilovém průmyslu jako hlavní komunikační protokol mezi senzory a výpočetními jednotky v automobilech
- fyzická vrstva jsou dva diferenční dráty, *CAN\_H* a *CAN\_L*, uzemnění a volitelné napájení
- multi-master protocol bez centrálního uzlu



Na sběrnici jsou dva definované stavy:

- recesivní (logická jednička)
- dominantní (logická nula)

Dominantní stav přetlačí recesivní, pokud dva uzly vysílají na sběrnici ve stejný čas různé stavy. Tato charakteristika je důležitá pro řízení přístupu k médium.

- stochastický přístup k médium (bez centrálního rozhodujícího uzlu)
- kolize jsou řešeny až když nastanou
- Carrier-Sense Multiple Access with Collision Resolution (CSMA/CR)
- **kolize vyřešeny na základě priority zpráv během arbitrační fáze přenosu**

Tento přístup k médium umožňuje využití sběrnice v real-time aplikacích, kde bývá vyžadován deterministický přenos kritických zpráv mezi zařízeními.

CAN zpráva se skládá ze dvou částí:

- arbitrační
- datová

Řízení přístupu k médiu řešeno během arbitrační fáze.

- každá zpráva má svůj identifikátor, který odpovídá její prioritě
- během arbitrace jsou identifikátory porovnávány bit po bitu
- dominantní stav přetlačí recesivní -> pokud zařízení vysílající recesivní bit na sběrnici vidí dominantní, z arbitrace ustoupí
- zpráva s vyšší prioritou (nižším ID) vyhraje arbitraci
- po vyhrané arbitraci uzel posílá data, ostatní zařízení se pokusí získat místo při další arbitraci

## Příklad

Máme dvě zařízení používající 3 bitový identifikátor. Zařízení A chce poslat zprávu s identifikátorem  $0x1$  ( $0b001$ ), zařízení B s  $0x2$  ( $0b010$ ). Při porovnání nejvyššího bitu se nic nestane, obě zařízení vysílají dominantní stav (logickou nulu). U druhého bitu (0 pro A, 1 pro B) ale dominantní stav z A přetlačí recesivní z B a tím zpráva zařízení B prohraje arbitraci.

	SOF	2	1	0	...	DATA
A	0	0	0	1	...	...
B	0	0	1	×	×	×

Synchronizační požadavky během arbitrační fáze znamenají omezení rychlosti na 1 Mbit/s při vzdálenosti přibližně 50 metrů.

Tato omezení částečně překonává novější standard CAN FD (flexible data rate).

- arbitrace probíhá standardní rychlostí
- po skončení arbitrační fáze dojde ke změně bitratu a samotná data jsou přenášena vyšší rychlostí
- data na 50 metrech teoreticky až 8 Mbit/s (automotive standard 500 Kbit/s a 2 Mbit/s)
- zvýšen datový payload z 8 bajtů na 64 bajtů
- **není zpětná kompatibilita, tedy nelze použít CAN FD zprávu, pokud na sběrnici operují zařízení, které umí pouze standardní CAN**

V posledních letech se pomalu začíná používat CAN XL standard umožňující přenést až 2048 bajtů v jedné zprávě.

Pro zjednodušení přístupu k hardwaru a plnému využití vlastností sběrnice operační systémy nabízí implementaci CAN/CAN FD subsystémů.

Menší systémy zaměřené na real-time jako NuttX nebo Zephyr mají vlastní character device drivery, GNU/Linux poskytuje na SocketCAN.

- využívá socket API, podobné rozhraní jako TCP/IP
- pomalu a v omezené míře přebírá i do NuttXu a Zephyru
- koncept socketů sedí k CAN síti

Implementace SocketCANu má ale své nevýhody.

- navržena v době standardního CANu, flagy pro CAN a CAN FD rozděleny
- socket API může být pro real-time systémy zbytečně složité
- pro RTEMS by znamenalo nutnost povolené TCP/IP implementace, nevhodné pro menší systémy bez networkingu nebo s omezenou pamětí



## 2 RTEMs

- původně Real-Time Executive for Missile Systems
  - <https://www.rtems.org/>
- vývoj začal koncem 80. letech na žádost US Army Missile Command
- kód vlastnila vláda USA -> public domain, díky čemuž se RTEMS stal rychle populární
- dnes prakticky plnohodotný real-time operační systém
- POSIX kompatibilní, zároveň vlastní rozhraní pro kernel
- networking stack z FreeBSD, IPv4/IPv6 TCP/IP, DNS server



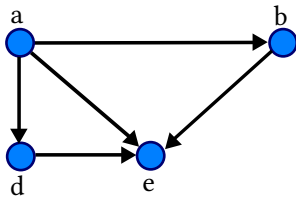
- ARM, x86, PowerPC, RISC-V, i386
- BSP s podporou SMP
- schedulery optimalizované pro více jader
- shared memory design
- single address space system
- kernel v C, aplikace i C++, Java, Lua, Rust

- původní stav bez obecného CAN/CAN FD stacku
- aplikace či BSP musely implementovat řešení specifická pro cílový target
- to zásadně komplikuje práci s drivery napříč platformami, zároveň chybí některé užitečné funkce
- užitečné funkcionality:
  - blokující/neblokující zápis a čtení
  - TX/RX polling
  - filtrace zpráv do prioritních front
  - error reporting
  - echo zprávy v případě úspěšného odeslání
  - konfigurace přes jednotné API

**Cílem bylo navrhnout API založené na modelu POSIX character device driveru**

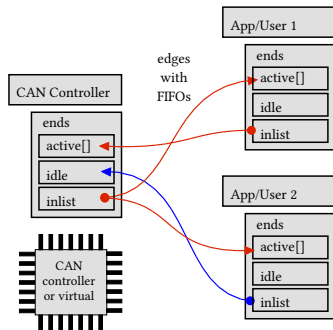
## **3 Design CAN FD subsystemu**

- založeno na LinCAN infrastruktuře
  - loadable modul do Linux kernelu vyvinutý na CVUT začátkem tisíciletí
  - později se od něj v Linuxu upustilo ve prospěch SocketCANu
- POSIX character device driver rozhraní
  - CAN ovladač registrován jako device do */dev* složky (např. */dev/can0*, */dev/can1*, ...)
  - standardní IO funkce *open*, *write*, *read*, *ioctl*
- využití FIFO front organizovaných do orientovaných hran mezi ovladačem a aplikací



Rozhraní sloužící k předávání CAN zpráv mezi kontrolerem a aplikací.

- fronty rozděleny podle směru a priority
- *inlist* jsou vstupy fronty -> zápis
- *idle* jsou prázdné výstupy -> čtení
- *active* jsou aktivní výstupy -> čtení
- stejné typy hran (a tedy funkcí) využity pro oba směry
- aktivní výstupy poskytují tři priority
- filtrace zpráv podle ID nebo flagů



- dvě fronty RX a TX automaticky vytvořeny při otevření driveru
- fronty pro CAN only ovladače alokují pouze 8 bajtů pro data
- fronty jsou pro různé otevřené instance na sobě nezávislé

Aplikace používající CAN mohou mít různé požadavky na přístup ke kontroleru.

- filtrace přijatých zpráv podle flagů
  - rozdělení CAN a CAN FD zpráv
  - fronta pro echo potvrzující odeslání
  - fronta pro error zprávy
- filtrace odesílaných zpráv podle priority
  - některé zprávy může aplikace chtít odeslat prioritně
  - rozdělení zpráv do front podle rozsahu ID (problém s *priority inversion*)
- `_mask` slouží k vyřazení zpráv s daným identifikátor či flagem

```
struct can_filter {  
    uint32_t id;  
    uint32_t id_mask;  
    uint32_t flags;  
    uint32_t flags_mask;  
};
```

```
ssize_t ioctl( fd, RTEMS_CAN_CREATE_QUEUE, &queue );
ssize_t ioctl( fd, RTEMS_CAN_DISCARD_QUEUES, type );

struct rtems_can_queue_param {
    uint8_t direction;
    uint8_t priority;
    uint8_t dlen_max;
    uint8_t buffer_size;
    struct rtems_can_filter filter;
};
```

- mazat lze pouze všechny RX, všechny TX nebo úplně všechny fronty
- infrastruktura zajistí odeslání všech zpráv před zničením front
- fronty se automaticky dealokují při zavolání *close*



```
struct can_frame_header {
    uint64_t timestamp;
    uint32_t can_id;
    uint16_t flags;
    uint16_t dlen;
};

struct can_frame {
    struct can_frame_header header;
    uint8_t data[CAN_FRAME_MAX_DLEN];
};
```

```
ssize_t write( int fd, struct can_frame *frame, size_t count );
```

Pro neblokující přístup lze použít TX polling čekající na alespoň jedno místo ve frontě.

```
ssize_t ioctl( fd, RTEMS_CAN_POLL_TX_READY, &timeout );
```

Aplikace se může chtít ujistit, že zprávy opravdu byly poslány.

```
ssize_t ioctl( fd, RTEMS_CAN_WAIT_TX_DONE, &timeout );
```

Omezení: *RTEMS\_CAN\_WAIT\_TX\_DONE* čeká na odeslání všech zpráv ze všech front.

Lze číst pouze jednu zprávu během jednoho volání *read*.

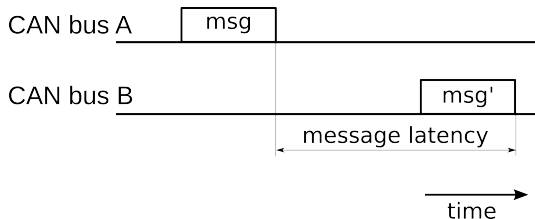
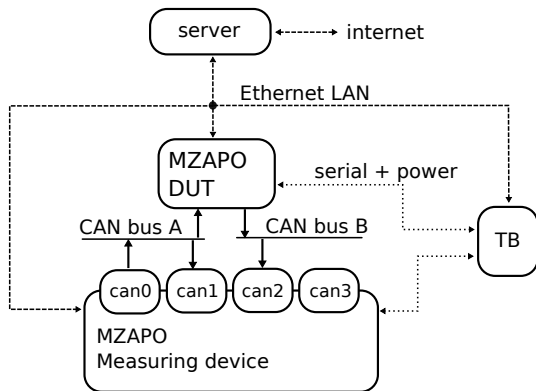
- pro *count* menší než délka zprávy *read* nevyplní celou strukturu, ale jen daný počet bajtů
- *count* musí být větší než délka hlavičky -> vždy se vyplní hlavička s délkou zprávy
- zpráva zůstane ve frontě, dokud není přečtena celá jedním *read*

```
ssize_t read( int fd, struct can_frame *frame, size_t count );
```

Pro neblokující přístup lze použít RX polling čekající na alespoň jednu zprávu ve frontě.

```
ssize_t ioctl( fd, RTEMS_CAN_POLL_RX_AVAIL, &timeout );
```

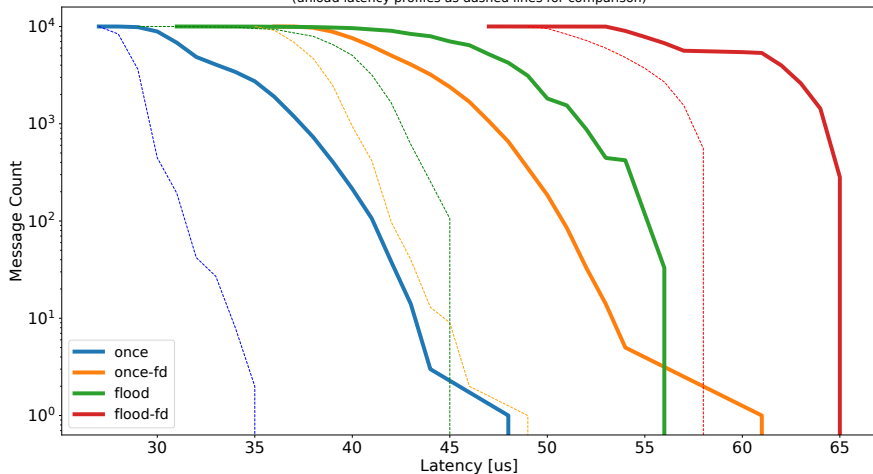
- CAN LaTester infrastruktura původně vyvinutá pro testování CAN latencí v Linux jádře
- *DUT* funguje jako CAN gateway: dostane zprávu ze sběrnice A a okamžitě ji odešle na B



- měřící zařízení generuje zprávy a poskytuje přesné měření
- CVUT MicroZed APO s otevřeným CTU CAN FD řadičem

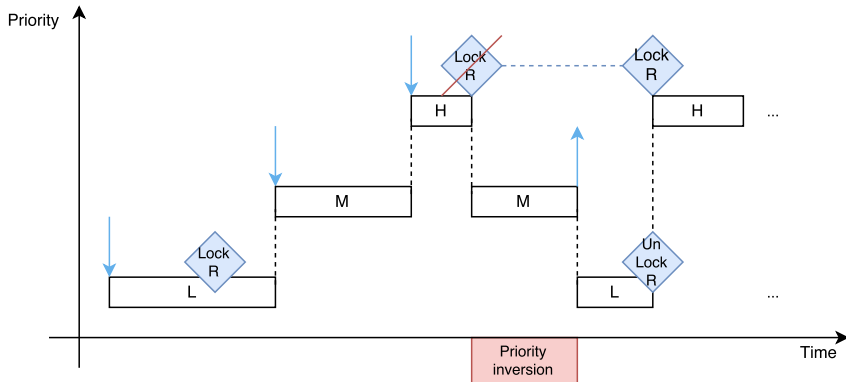
## RTEMS CAN Stack Read to Write Latency Cumulative Histogram with Networking

(unload latency profiles as dashed lines for comparison)

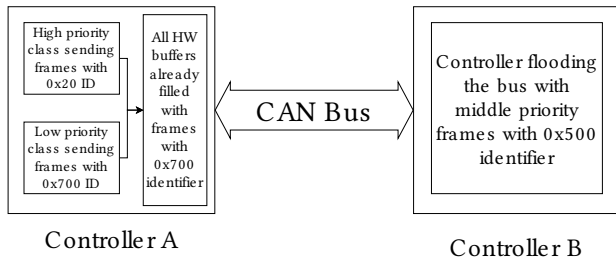


## 4 Problém inverze priorit

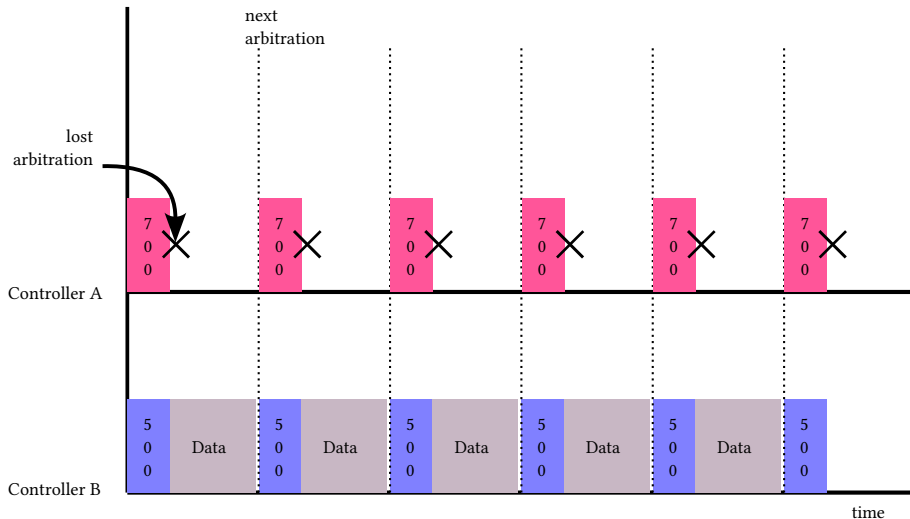
- task s nízkou prioritou, pozastavený taskem se střední prioritou, drží kritickou sekci potřebnou pro task s vysokou prioritou
- task s vysokou prioritou je tak efektivně blokován taskem se střední prioritou



- vzniká během arbitrační fáze
- místo tasku máme CAN zprávy, kritické sekce jsou HW buffery kontroléru
- jedno zařízení zahlcuje sběrnici zprávami se střední prioritou
- druhé se snaží poslat mix zpráv s nízkou a vysokou prioritou
- pokud jsou HW buffery zaplněné zprávami s nízkou prioritou, vysoká priorita se nikdy nedostane k arbitraci







- běžné je využití prioritních tříd/front (*priority classes*), běžně FIFO front
- CAN zprávy jsou přiřazeny do těchto tříd na základě rozsahu jejich identifikátorů (priorit)

## Příklad

Zprávy s ID `0x0` až `0x200` jdou do fronty s nejvyšší prioritou, zprávy `0x201` až `0x500` do nižší priority atd.

- díky tomu se vysokoprioritní zprávy dostávají do hardwaru ve své vlastní frontě
- to řeší zdržení zpráv na úrovni softwarových bufferů
- fronty je potřeba namapovat na hardwarové buffery, což způsobuje problémy

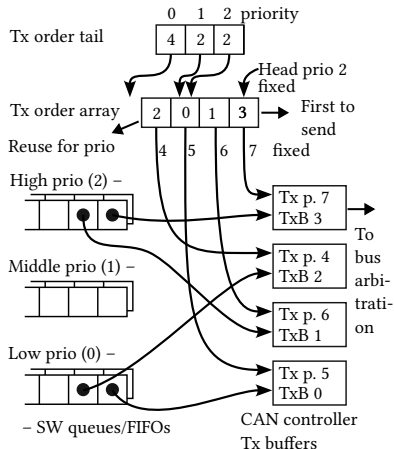
- implementace CAN řadičů v mikrokontrolerech obvykle umožňují posílání zpráv na základě jejich identifikátoru nebo podle pořadí určeného indexem bufferu
- řeší vysílání na základě identifikátoru, tedy priority, problém?
  - aplikace a protokoly mohou vyžadovat zachování pořadí odeslání v rámci prioritní fronty i při použití jiných identifikátorů
  - posílání na základě ID tedy nepoužitelné
- zařízení často limitují počet HW bufferů na jeden pro každou třídu
- to výrazně limituje poskytnutý hardwarový potenciál

**Zprávy z nejvyšší prioritní třídy musí odejít jako první a pořadí zpráv v rámci třídy musí být zachováno**

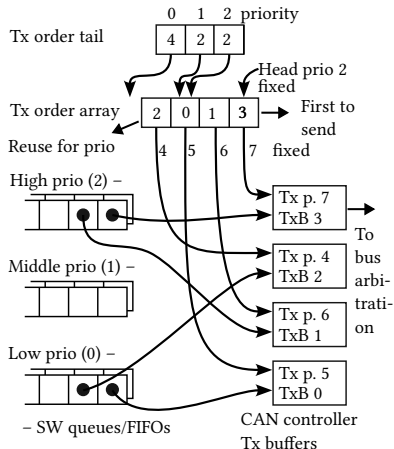
## **5 Dynamická redistribuce hardware bufferů na prioritní třídy**

- design, který rozšiřuje separátní FIFO fronty pro každou prioritní třídu
- přidána dynamická redistribuce HW bufferů na tyto třídy
  - fronty tedy již nejsou pevně přiřazeny k určitým bufferům
- algoritmus zajišťuje, aby HW buffery dostaly správné pořadí v němž mají přistupovat ke sběrnici a tedy k arbitraci
  - toto pořadí je určeno na základě prioritní třídy a pořadí zpráv v rámci této třídy

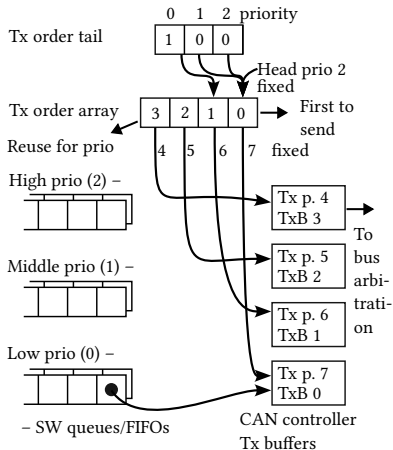
**Pokud je do bufferu vložena nová CAN zpráva, tak buffer musí být ve vysílací sekvenci po všech zprávách ze stejné či vyšší prioritní třídy, ale před všemi zprávami z nižší prioritní třídy**



- *TX order array* ukazuje na HW buffery v pořadí, v jakém by měly jít na arbitráž
- *TX order tail* ukazuje na ocásek dané prioritní třídy - místo, kam by měla přijít nová zpráva z této třídy
- hlavička nejvyšší priority je fixní, hlavičky dalších priorit jsou stejné jako ocásek předchozích priorit
- pokud je vložena nová zpráva, všechny zprávy z nižší priority posunuty doleva

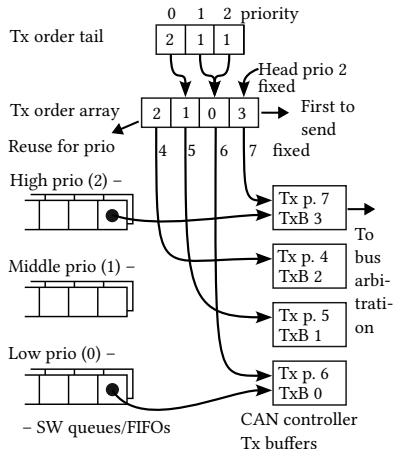


- pokud není volný žádný HW buffer a chceme vložit zprávu vyšší priority, než jaká je v bufferech, musí dojít k abortu
- **nejpozději vložená zpráva z nejnižší priority je odebrána z HW bufferu a držena v SW frontě pro pozdější zpracování**
- nová zpráva vložena do volného HW bufferu a změněno požadí, aby odpovídalo pravidlům

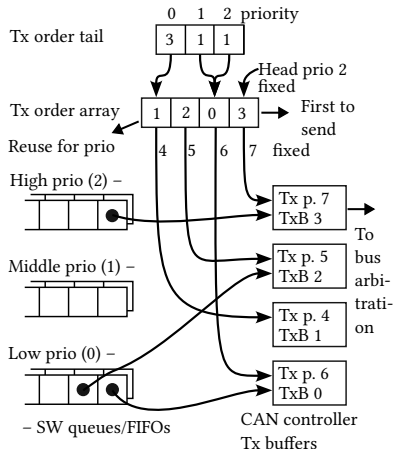


- máme zprávu s nízkou prioritou přiřazenou do HW bufferu 0
- *TX order tail* pro nízkou prioritu přesunut na pozici 1
- buffer 0 je první k poslání s nejvyšší prioritou 7

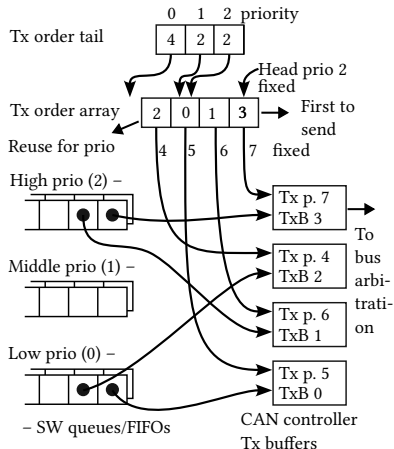




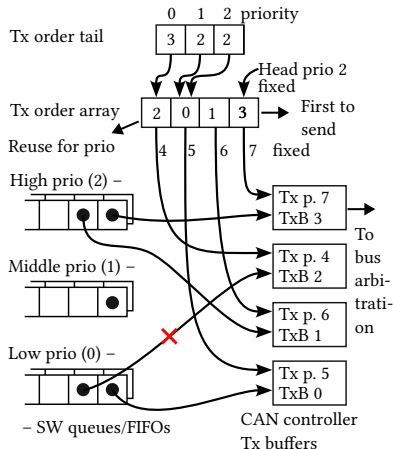
- aplikace zapíše zprávu s vysokou prioritou, ta je přiřazena například do bufferu 3
- tuto zprávu chceme do arbitrace poslat před nízkoprioritní zprávou
- *TX order tail* pro vysokou a střední prioritu posunut na 1, pro nízkou na 2
- buffer 3 teď má nejvyšší prioritu 7, buffer 0 nižší 6
- řadič se tak bude snažit vysílat buffer 3



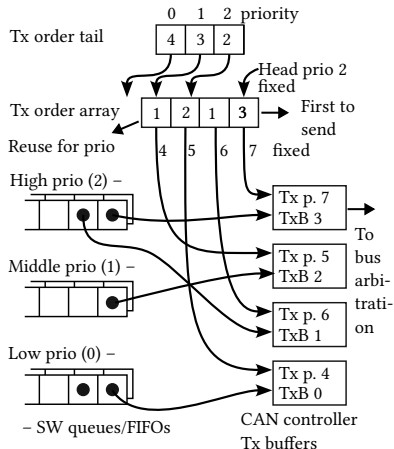
- dojde k zapsání další low priority zprávy, je přiřazená do bufferu 2
- buffer musí být zařazen za všechny vysoko a středně prioritní a za všechny dříve vložené zprávy s nízkou prioritou
- *TX order tail* pro vysoké a střední priority zůstane, po nízkou se posune na 3
- priority bufferů 2 a 1 se prohodí
- současné pořadí odchodu na sběrnici:
  - buffer 3 s vysoko prioritní zprávou
  - buffer 0 s první nízkou prioritní
  - buffer 2 s druhou nízkou prioritní



- zpráva s vysokou prioritou vložena do bufferu 1
- buffer musí být zařazen za předchozí vysoko prioritní zprávu a před všechny střední a nízké priority
- *TX order tail* pro vysokou se posune na 2, pro nižší priority se posune doleva (nízká už se nevejde)
- současné pořadí odchodu na sběrnici:
  - buffer 3 s první vysoko prioritní
  - buffer 1 s druhou vysoko prioritní
  - buffer 0 s první nízkou prioritní
  - buffer 2 s druhou nízkou prioritní



- aplikace zapíše do fronty zprávu se střední prioritou
- všechny HW buffery jsou plné, musí dojít k abortu poslední zprávy s nejnižší prioritou
- proveden abort na buffer 2 - poslední vložená nízko prioritní zpráva
- *TX order tail* pro nízko prioritu posunut na pozici 3
- odebraná zpráva s nízko prioritou ponechána v SW bufferu, bude vyřešena později

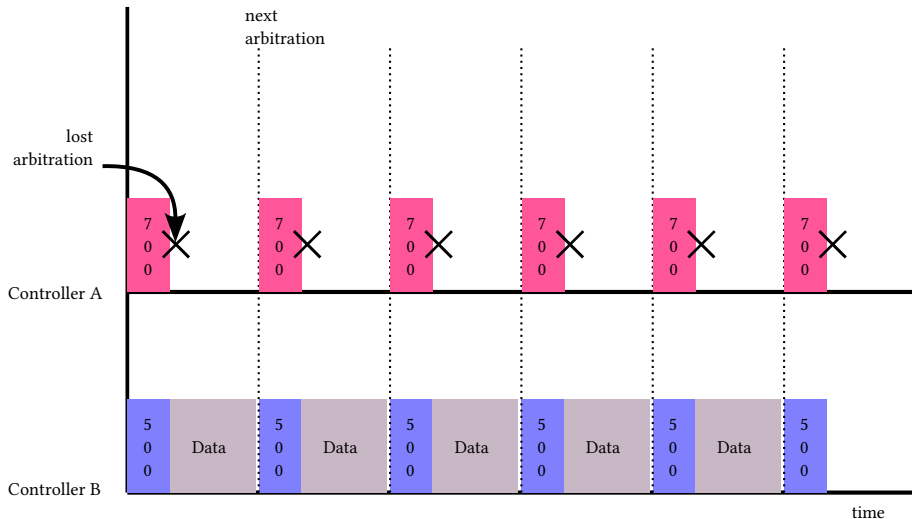


- zpráva se střední prioritou je vložena do již volného bufferu 2
- tento buffer se musí zařadit mezi vysoko a nízko prioritní buffery
- *TX order tail* pro střední prioritu posunut na 3, pro nízko na 4
- buffer 2 dostane prioritu 5, buffer 0 (drží zprávu s nízkou prioritou) dostane 4
- současné pořadí odchodu na sběrnici:
  - buffer 3 s první vysoko prioritní
  - buffer 1 s druhou vysoko prioritní
  - buffer 2 se středně prioritní
  - buffer 0 s první nízko prioritní

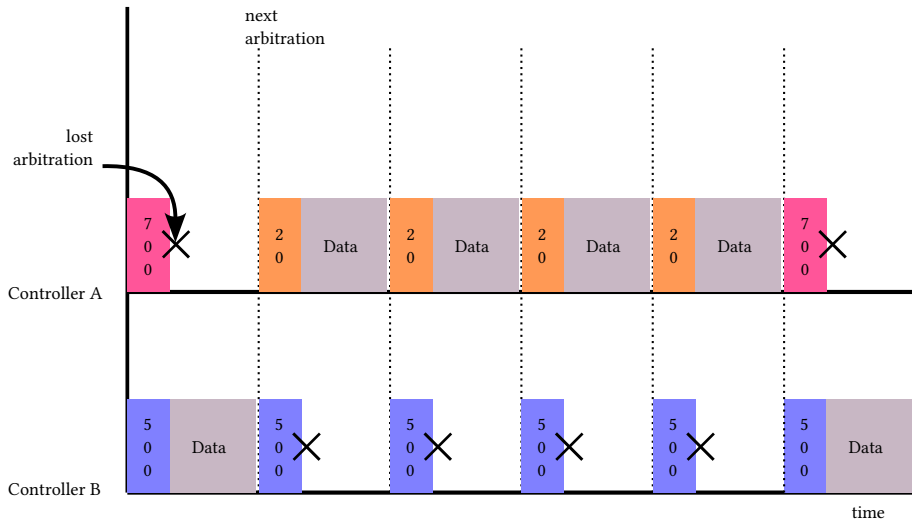
- na vyšší vrstvu (obecné CAN/CAN FD rozhraní)
  - podpora více prioritních tříd a filtrace zpráv do nich
  - možnost přeplánovat abortnuté zprávy na pozdější zpracování (slot by měl zůstat alokovaný, dokud není zpráva odeslána na sběrnici)
- na řadič
  - možnost určit pořadí, v jakém mají HW buffery odcházet na sběrnici (ideálně bez deaktivace řadiče)
  - možnost provedení abortu na HW buffer

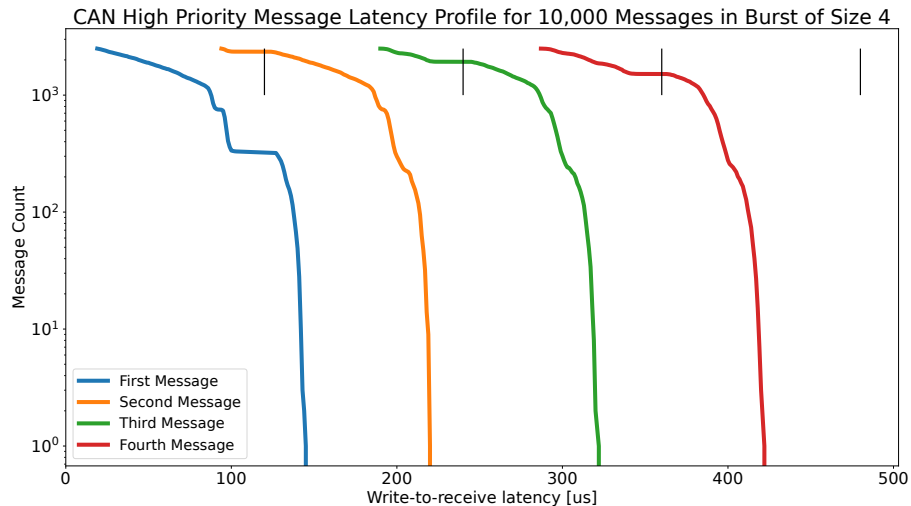
- testování na kitu MicroZed APO s Xilinx Zynq-7000 a CTU CAN FD řadiči
  - CTU CAN FD IP řadiče připravené pro abort TX bufferu a změny priority za běhu
  - přesný timestamping s rozlišením 10 ns
- dva CAN FD řadiče na desce
- každý řadič se čtyřmi hardwarovými buffery
- řadič A zahlcuje sběrnici zprávami s ID `0x500` -> střední priorita
- řadič B vysílá zprávy z dvou aplikací
  - jedna posílá ID `0x700` -> nízká priorita
  - druhá posílá ID `0x20` -> vysoká priorita
- vysoká priorita posílaná v burstu po 4 zprávách, zpoždění mezi bursty dostatečně velké, aby se všechny 4 zprávy stihly poslat

**Simulace plně zahlcené sběrnice jedním řadičem, zatímco druhý se snaží přistupovat na sběrnici a dochází v něm k potenciální inverzi priorit.**









## 6 Otázky

- CVUT CAN projekty: <https://canbus.pages.fel.cvut.cz/>
- CAN zapojení CC BY-SA z [https://commons.wikimedia.org/wiki/File:CAN-Bus\\_Elektrische\\_Zweidrahtleitung.svg](https://commons.wikimedia.org/wiki/File:CAN-Bus_Elektrische_Zweidrahtleitung.svg)
- Píša, Pavel. Linux/RT-Linux CAN Driver (LinCAN). <https://cmp.felk.cvut.cz/~pisa/can/doc/lincandoc-0.3.pdf>
- Federico Reghenzani, Giuseppe Massari, and William Fornaciari. 2019. The Real-Time Linux Kernel: A Survey on PREEMPT RT. ACM Comput. Surv. 52, 1, Article 18 (January 2020), 36 pages. <https://doi.org/10.1145/3297714>
- CAN Bus Latency Test Automation for Continuous Testing and Evaluation. Matěj Vasilevski, 2022: <https://dspace.cvut.cz/handle/10467/101450>
- CAN FD Support for Space Grade Real-Time RTEMS Executive; Michal Lenc; 2024: <https://dspace.cvut.cz/handle/10467/114749>