Design of High-Performance CAN Driver Architecture for Embedded Linux

Sakari Junnila, Risto Pajula, Mickey Shroff, Teemu Siuruainen, Marek Kwitek, Pasi Tuominen,
Wapice Ltd.

Use of Linux in embedded systems has become vastly popular. On hardware platforms, the ARM processor cores have a strong foothold. To address the needs of Linux-based embedded automation systems, Wapice has implemented custom high-performance CAN driver architecture. The Wapice Custom Can Driver (WCCD) is targeted for embedded Linux and optimized for ARM-based platforms. In this paper, we present our findings made during the development process and methods we used to optimize the driver performance. Performance measurements, comparing the effects of optimizations, are also presented. We discuss how CAN message buffering algorithms affect the bus performance, show how Linux kernel version affects the interrupt latencies, and present challenges related to SPI-based CAN transceiver chip usage. To evaluate our design, we present the performance of Wapice Custom CAN driver (CPU load, CANmsg/s) and compare it against SocketCAN and LinCAN implementations. We also show results of a brief study on porting WCCD to other processor platform. Based on the results, we conclude that WCCD is an extremely high-performance embedded Linux CAN driver which can match or outperform the compared existing implementations.

1. Introduction

Linux has taken a strong foothold in embedded systems during the last decade. The emergence of real-time Linux variants and patches have made Linux attractive for embedded automation systems [1]. The real-time challenges also effect driver development [2], including the CAN-bus driver, one of the key communication interfaces in embedded automation.

The CAN-bus and the short frame lengths of the protocol create demanding requirements for the implementation of the CAN-bus node. The minimum CAN frame length is 47 bits [3]. To achieve reliable operation, each node should be able to handle frame reception within 47 us. It is the shortest CAN frame duration when the bus speed is 1Mbit/s. This means that each bus node should be able to handle up to 21277 received CAN frames per second. When this requirement is combined with modern operating system architecture such as Linux and ARM core processor platform, unique challenges are set to the application and the device driver architecture.

1.1. Related work

The most commonly used Linux device driver is SocketCAN, which is included in Linux mainline kernel [4]. SocketCAN is built on standard Linux networking infrastructure, making it easy to use for developers familiar with TCP/IP. As Linux networking infrastructure is designed for high volume data transmission optimizing throughput, this implementation approach does not yield optimal results in low-latency, low-throughput embedded CAN systems.

LinCAN, a project started at Czech Technical University of Prague, has been designed with emphasis on strict real-time properties and reliability [5]. It is a versatile

Sakari Junnila, Risto Pajula,
Mickey Shroff, Teemu Siuruainen,
Marek Kwitek and Pasi Tuominen
Wapice Ltd
Yliopistonranta 5, 65200 Vaasa
Tel. +358 6 319 4000
Fax +358 6 319 4001
{firstname.lastname}@wapice.com
www.wapice.com

driver which has also been ported to various platforms and is available as open source code. It is implemented as a Linux character device driver. A downside with the LinCAN vs. SocketCAN has been the lack of a common API. As a solution, a SocketCAN compatible VCA (Virtual CAN API) has been proposed and implemented [5]. LinCAN is not included in Linux mainline kernel.

LinCAN was developed to address the RT-performance drawbacks of SocketCAN, and as such has similar goals to our work. The developers of LinCAN have compared their driver performance to SocketCAN in [5], which is closely related to the work presented in this paper.

1.2. Background and goals

Our work was inspired by the development of a CAN-over-Ethernet gateway device [6], a configurable platform for real-time monitoring applications. It used a proprietary CAN-based protocol with high CAN-bus frame rate of over 8000 messages per second. The gateway device has an Atmel AT91 SAM9260 microcontroller with ARM 926EJ-S core running at 180 MHz. CAN-bus interface is implemented with Microchip MCP2515 CAN controller. The microcontroller communicates with the CAN controller through Serial Peripheral Interface (SPI) bus.

The gateway device used SocketCAN as its original CAN driver implementation and publicly available MCP2515 device driver. The driver was later merged into the 2.6.33 kernel. The work progressed in three phases. At first, the platform and device driver optimizations were applied to the SocketCAN driver. Even with full platform optimizations the performance requirements of the application were not met. In the second phase, the LinCAN driver was ported to the gateway device. Performance was compared, but also LinCAN couldn’t match the performance requirements. This led to the design of the Wapice Custom CAN Driver (WCCD). It uses the same platform and CAN chip device driver optimizations but specifically addresses the problems in the CAN message buffering. It also optimizes the interface between the kernel space and the user space application. The following design constrains were defined for the WCCD.

- The driver architecture should have high performance and it should handle full theoretical CAN message rate.
- It should consume as little as possible of the CPU time.
- It should minimize the messages lost due to HW buffer overflow situations.

The following aspect in the driver design was not considered important:

- Ability to read CAN messages from multiple user space threads.

We studied the previously mentioned CAN driver alternatives and present an optimized driver implementation, measure and evaluate its performance, and discuss the drawbacks with our chosen optimizations. The following Section 2 presents common optimization methods that can be applied. Section 3 presents our driver architecture and the applied optimizations. The performance of the driver was compared against the SocketCAN and the LinCAN drivers. These results are presented in Section 4. This is followed by Conclusions and ending with discussion based on the results in Section 6.

2. Optimization methods

We have identified five critical implementation areas: Interrupts, direct memory access (DMA), inlining, use of locks and branch hints.

2.1. Interrupts

Interrupts can cause a significant amount of CPU load to the system if the interrupt rate is high. Each interrupt results in at least one context save and reload procedure. In traditional Linux systems, the interrupt service routine (ISR) schedules the occurring interrupts inside the interrupt task context. Normally, the ISR is also divided to top and bottom halves. The bottom half is signaled from the interrupt thread context.

In a preempt-RT Linux, interrupts are normally run in their own thread context. The benefit of threaded interrupts is that they can

be prioritized. In addition, the preempt-RT Linux provides a mechanism to run ISR's in the interrupt context.

Due to the amount of context switches needed in the Linux architecture, the frequency of interrupts should be minimized and possible code paths should be made as short as possible for optimum performance. They introduce latency to the system.

2.2. DMA

Most modern CPU's allow memory transfer operations to be offloaded from the CPU to DMA engine, which can be used to move data between memory locations and peripheral devices. Normally an interrupt is generated when a DMA transfer has completed. Using the DMA engine, an application can perform other operations while memory content is transferred.

Use of DMA requires platform dependent code to allow DMA buffers to remain coherent throughout transactions. Platform itself defines where DMA buffering can be used.

2.3. Inlining

Generally, it is good to have functions to separate functionalities and to reduce the amount of program code. However, every function call causes several register writes on target CPU. This is not an issue on functions which are called only a few times, but for functions that are called several thousands of times a second, the amount of time spent accumulates. When a function is inlined, the function call is replaced with the content of your function. It will increase the size of your final binary, but it can speed it up moderately.

GCC, a popular Linux compiler, only takes the inline keyword as a hint, which may not be taken into account. Functions can be forced to be inlined regardless of the optimization level by setting a special attribute. The attribute is defined after the function name like this:

```
inline int YourFunction(void)
__attribute__((always_inline));
```

2.4. Use of locks

In all multi-threaded systems some kind of mutual exclusion mechanism is needed to ensure validity of the shared data. Operating systems usually provide mutexes, semaphores, spinlocks and critical sections for this purpose. Usage of these mechanisms can be relatively CPU time expensive. Alternatively an atomic operation support can be used to implement simple synchronization between threads and interrupts. Linux OS provides a platform dependent atomic API that is available for many CPU platforms. This API can be used to implement efficient algorithms and to synchronize data between threads.

The ARMv5 CPU core architecture used on our target processor does not provide CPU instruction set support for the implementation of the atomic API. Instead, the implementation uses short critical sections where interrupt are blocked.

2.5. Branch hints

If the CPU execution jumps to a branch that the CPU branch prediction logic has not been able to predict, the CPU instruction pipeline has to be flushed. Flushing can cause a delay of several clock cycles. It is advisable at the source code level to provide hints for the compiler when there is strong probability of certain condition with a compiler keyword. GCC provides mechanism for static source code level branch prediction hints. This allows the code to be organized so that in the more likely code path the execution is linear. Linux kernel provides preprocessor macros likely() and unlikely() to control the branch prediction.

3. Linux CAN device driver implementation

In general, a device driver is a software component which provides access to a HW resource (device). From the main operation viewpoint a low-level communication peripheral driver such as CAN device driver has three main tasks: receive, transmit and media/bus control. In addition, the device driver has operating system dependent functionalities. Implementation of receive and transmit operations define the practical driver performance: latency and throughput. For our driver design, the main goal was to

obtain high-performance buffer read and write operations.

To obtain high performance, we made the following design decisions, for which reasoning is given later:

- Implement driver as character device.
- Interrupt service routine (ISR) should never have to wait while adding CAN frames to the buffer.
  - o ISR operation can never be interrupted by the character device reading the buffer. No operation is done while ISR is using the buffer.
  - o Character device's buffer operation can be interrupted at any moment by the CAN ISR.
- Multiple CAN frames can be read at once.
- One-way communication in reception; ISR adds CAN frames and character device reads them from the buffer.
- Received CAN frames are overwritten by new frames if no more free space is available in the buffer.

The character device driver implementation principle is presented in Figure 1. IOCTL is used to set read timeout, bitrate and other configuration options. Timer counter interrupt is used to schedule buffer reads. The architecture of WCCD is shown in Figure 2.

3.1. Platform resources

The MCP2515 has two CAN message reception buffers with rollover support. Thus 2-stage primitive HW FIFO is possible [7]. After the reception of the first message (First buffer is full), the MCP2515 will generate a CPU interrupt. In the worst case situation, this gives 2*47 us theoretical minimum for the CPU to service the interrupt and read the message through SPI before an HW buffer overflow can occur.

The time to handle a CAN message reception can be divided to following phases; ISR latency, the SPI transaction time and the software overhead in the interrupt service routine.

3.2. Performed platform optimizations

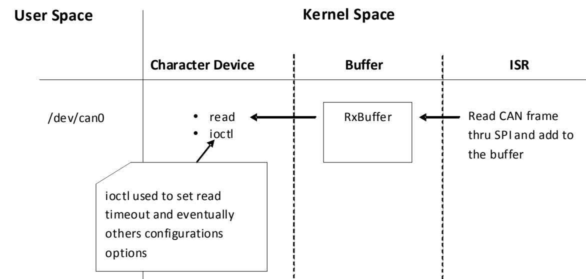

Figure 1. Receive buffer implementation.

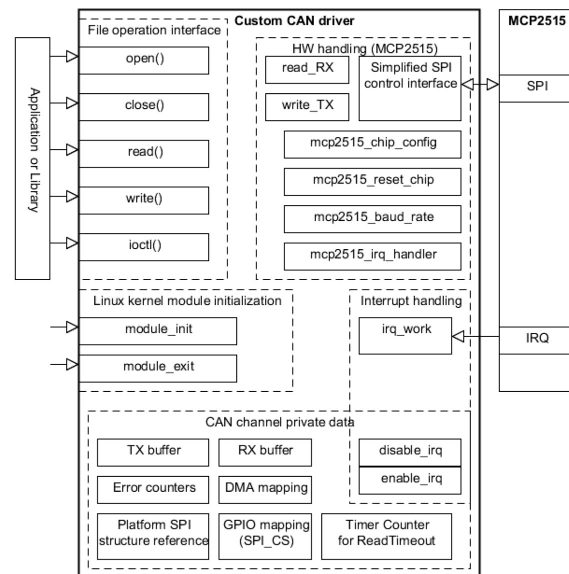

Figure 2. WCCD architecture on ARM platform using an external SPI CAN-controller (MPC2515).

The interrupt latency in the system has been optimized by using Preempt-RT patched kernel. The Preempt-RT patch moves normal ISR routines to tasks, which provides better priorization between interrupts and overall interrupt latency. The IRQF_NODELAY flag provided by the Preempt-RT framework was used to prioritize the MCP2515 CAN ISR and thus the CAN message handling over all the other functionalities in the system. Different kernel versions were tested and the effects of the kernel versions to the ISR latency are displayed in Table 1.

The interrupt latency has been measured by initializing a CPU peripheral timer interrupt to

Table 1. Kernel version effect on ISR latency.

	Min (us)	Avg (us)	Max (us)	Max - USB (us)
2.6.29.6-rt24	8,2	13	108	131
2.6.31.12-rt20	9,2	13,9	115,6	125
2.6.29.6	5,7	10,4	77	6727
2.6.31.12	7,6	11,4	88,1	6572

be performed in the overflow situation. The

free running counter of the timer is sampled in the ISR and the interrupt latency is calculated from the value of the free running counter. Over 2000 interrupts were sampled to get the results for the interrupt latencies. During the test, 10 user space threads were running with 100 % CPU utilization. The Max-USB results demonstrate the interrupt latency during the insertion of USB memory stick and mounting of the FAT file system. From the results we can conclude that the non-preempt-RT systems can provide better average latency but they fail to perform deterministically under some operating situations. Also, the maximum measured latency indicates that it is impossible to guarantee no loss of CAN frames, because the maximum interrupt latency should be less than two CAN frame transmission times (MCP2515 has two reception buffers).

3.2.1. PIO optimizations

In the hardware platform used, the MCP2515 interrupt line is connected to the Parallel Input/Output (PIO) controller of the Atmel AT91SAM9260 microcontroller. The PIO controller can be configured to give interrupts from the PIO pin status changes. In the AT91 Linux port, the PIO controller interrupts are handled in the different code path than other interrupts. The PIO interrupt handling was optimized by replacing the loop through the PIO_ISR status flags with instructions that resolve the first bit set in the word.

3.2.2. MCP2515 interrupt handling optimizations

The interrupt service routine of the MCP2515 was optimized heavily. Normal micro optimization and static branch prediction techniques were used in this critical code. Also function inlining was used heavily. Logic for the ISR was implemented to disable further interrupt requests from the MCP2515 while executing the ISR. At the end of the ISR routine the PIO pin, normally used as the interrupt pin, is polled in the digital input mode to check if the MCP2515 is still trying to interrupt the processor. With PIO polling, a possible second interrupt can be serviced without unnecessary ISR generation. Depending on the CAN message length, message transmission timing and interrupt latencies; this method can significantly reduce the interrupts from MCP2515.

3.2.3. DMA optimizations

The data transfer to and from the MCP2515 is implemented using the SPI-bus. The AT91SAM9260 supports DMA transfer from the SPI peripheral device receive and transmit buffers directly to the memory of the CPU. The original MCP2515 driver utilized the kernel's DMA buffer allocation dma_alloc_coherent()–API and the spi_message_init() interface. In the target platform, this results in a DMA buffer mapping for each SPI transaction. According to our measurements, a more efficient way is to use the streaming DMA mapping API provided by the kernel. In this implementation, the kernel provides API for the DMA buffer synchronization: the dma_sync_single_for_cpu() –function is called to ensure the data in the buffer can be used by the CPU. Before data in the transfer buffer is started again, the dma_sync_single_for_device() –function must be called to give the ownership of the buffer back to DMA engine. This also ensures the data has been flushed from the cache to the memory before the transfer is started. According to our measurements, the DMA buffer handling optimization saved 16 us from each receive message interrupt.

3.2.4. SPI handling optimizations

The MCP2515 has an SPI interface with maximum SPI clock frequency of 10 MHz [7]. For the receive messages, the most efficient way to service the MCP2515 interrupt requires two SPI transfers. First, the CANITF register is read. The second transfer reads the message buffer content. An SPI transfer requires 3+14 = 17 bytes of data to be transferred through the SPI interface. With 10 Mhz clock, the theoretical minimum for SPI traffic is 13.6 us. The AT91 Linux kernel SPI implementation for the MCP2515 usage was optimized by inlining the code manually in the MCP2515 ISR. Additionally, the SPI implementation was modified to not to generate interrupts; instead the SPI peripheral is polled for the transfer completion from the status flag of SPI peripheral. The current implementation is MCP2515 specific. It would be possible to create a generic AT91 SPI driver which

would provide generic purpose SPI functionality with the same benefits for the short SPI transactions assuming that completion polling is shorter than the interrupt latency.

3.3. Receive buffer implementation

CAN driver receive buffers are usually implemented as ring buffers. They require lock mechanisms during read/write operations: new data cannot be added to the buffer while it is being read. This can increase receive latency.

The receive buffer for the CAN message reception was implemented using a single writer single reader non-blocking algorithm utilizing the Atomic API. This approach minimizes the usage and length of the critical sections and the priority inversion with user space process and the ISR.

3.4. Transmit buffer implementation

The CAN frame write operation in WCCD is implemented with asynchronous I/O. The write operation copies the CAN message data to the internal circular buffers. Driver handles internally the pointers to the head and tail of the buffer.

As a general note, sending as many messages as possible at once reduces the amount of switches between user-space and kernel, and improves the performance.

3.5. Driver usage

At driver API level, read & write is performed using a 20-byte data structure representing CAN frame data to ensure minimum CPU load. If less than full CAN frame is to be sent, the data size field is set to indicate the amount of data and unused data fields are left blank. Only one thread can read from the device file at any given time.

The device read function blocks until a specific timeout occurs. This enables the driver to receive multiple frames and return them to user space with one read() system call. The timeout can be set by the user with IOCTL–function.

3.6. Port to PowerPC

As the receive buffer implementation didn't have platform dependencies, we ported it to a PowerPC based platform previously using SocketCAN. The major difference between the platforms was the internal CAN peripheral vs. the SPI CAN controller used in the ARM platform.

4. Results

The following sections present our performance measurement results. Majority of the measurements vere performed on the ARM core platform. For comparison, we ported part of the driver stack to PowerPC, and repeated CPU load and receive latency measurements on it.

4.1. ARM platform

The ARM implementation results were obtained with Atmel AT91SAM9260-based hardware running custom Linux kernel version 2.6.29.6-rt24. SocketCAN comparisons were made with version dated 21.8.2009 and LinCAN measurements with version 0.3.4 (March-2009). These were the version available at the start of this work. The WCCD measurements were made with WCCD version 2.2.0.

Figure 3 shows CPU load when receiving CAN frames at different frame rates. Figure 4 shows similar CPU load when sending echo messages to another device. As can be seen from both the figures, WCCD load on CPU is significantly lower. To review the sources of receive latency, Figure 5 shows the CAN ISR execution time for reception.

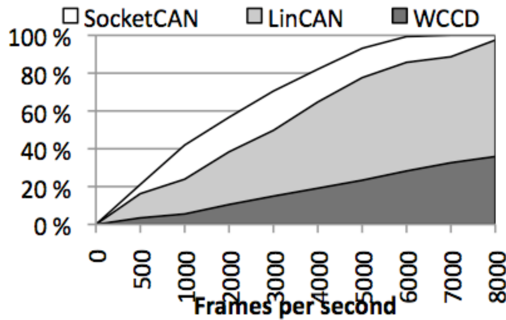

Figure 3. CPU load on receive.

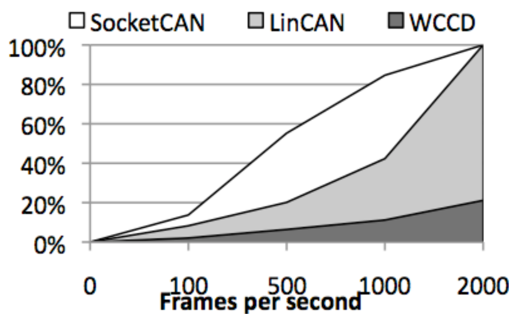

Figure 4. CPU load on echo.

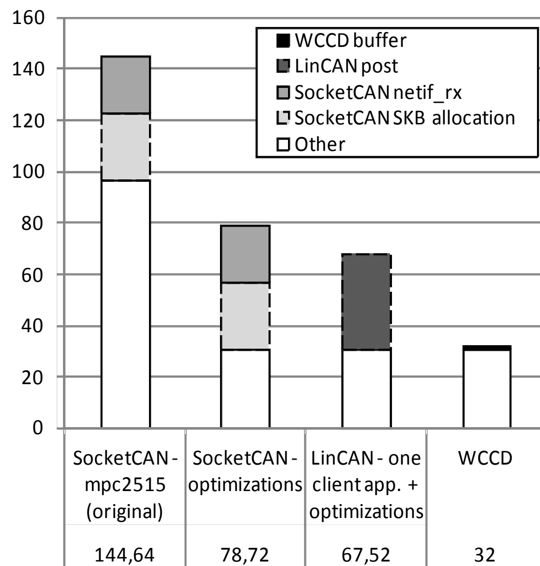

Figure 5. CAN receive ISR execution time (us). Other includes mcp2515, SPI, DMA and AT91 code.

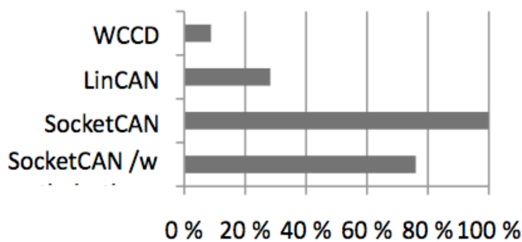

Figure 6. Measured CPU load when transmitting 1000 frames/s.

Figure 6 shows CPU load on transmit. Again, WCCD load is significantly lower. We also measured CAN message transmit latencies for WCCD, which are listed in Table 2. The measurements were performed at 1 Mbit and 250 kbit bus modes. First column shows time from userspace application write call to data on bus. Second column shows the same measurement for driver only. The third column lists the time for the driver when three frames are written to the bus.

4.2. PowerPC platform

For comparison, we ported our CAN driver receive stack to a Freescale MPC5200B PowerPC-based platform. This was done to verify that the receive buffer optimizations were not platform dependent. Linux kernel version for the device was 2.6.23.1 with somewhat older SocketCAN revision 454 (3.8.2007). The IPB-bus used by MSCAN peripheral was set to 66 MHz.

Figure 7 is similar to ARM measurement Figure 5, and compares CAN receive ISR execution time with normal SocketCAN and with WCCD receive buffer implementation. Figure 8 shows the CPU load with different receive rates at 1 Mbps bus speed.

5. Conclusions

Our measurements show that the proposed implementation has significantly lower CPU load both on transmit and receive operations. In addition, receive interrupt execution time is notably shorter and transmit latencies are very short. The measurements have been partly verified on another processor platform with similar results. We conclude that WCCD is highly optimal CAN driver for applications requiring low-latency and low CPU-load. The implemented optimizations propose some

Table 2. Measured CAN-message write times (us). Write-time is the time taken for the write-call execution. Start-delay is the time from start of write call to beginning of bus activity. Total-time is the time from start of write command to end of bus transmission.

	1Mbit			250 kbit		
	1 frame		3 frames	1 frame		3 frames
	userspace	*driver*	*driver*	*userspace*	*driver*	*diver*
Write-time	74	50	50	70	50	50
Start-delay	52	48	48	55	51	50
Total-time	94	99	335	220	240	850
1st frame			100			250
2nd frame			125			300
3rd frame			110			300

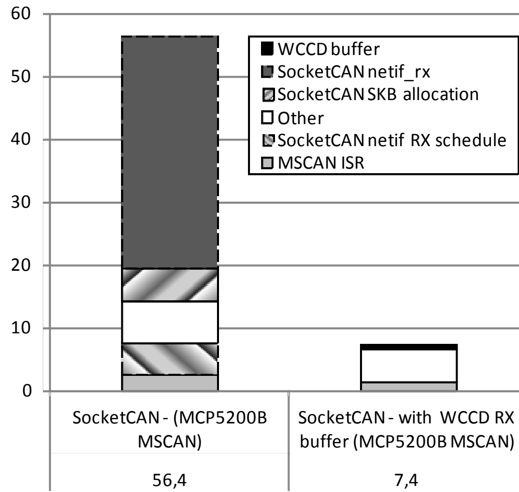

Figure 7. CAN receive ISR execution time (us). Tasks listed in execution order (earliest at bottom).

limitations to the driver use.

6. Discussion

The WCCD driver architecture performs significantly better than the standard solutions in high bus load environments, especially on CPU platforms with limited resources. For the ARM-based gateway device implementation, we have been able to achieve nearly lossless CAN message reception that was impossible with SocketCAN or LinCAN drivers. We have also tested the driver's performance with extremely high bus loads and have been able to receive over 20000 CAN messages per second. To achieve this, we had to make significant amount of optimizations also to the platform and low-level driver code outside of the CAN driver framework. But as the evaluation of the executed performance measurements show, the usage of the WCCD driver architecture provides clear benefits over alternative CAN driver implementations.

As the usage of the Linux for the demanding CAN-bus systems increases, it seems justifiable to create a custom CAN driver architecture and optimize the platform code for a specific purpose to achieve reliable operation.

The biggest downside for the usage of the WCCD is the lack of a standard interface. Standard software cannot be used with it without porting it to the WCCD driver API. However, embedded systems applications are often proprietary and developed to some specific purpose.

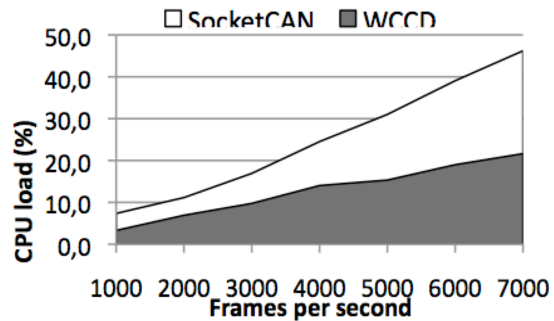

Figure 8. CPU load on CAN receive, PowerPC platform.

As mentioned before, the reading of data from two or more threads or processes is not currently supported. Adding multithreaded (and multiprocess) read, would roughly double the time it currently takes to write to the receive buffer. However, writing data from multiple threads to the device driver is supported.

The platform optimizations made for AT91 Linux port have not been sent to the Linux community as the changes are highly board specific and would require further work to make them more generic. The SPI optimizations made to support the usage from hard interrupt context would probably benefit many projects. However, this would require major work in the whole SPI subsystem of the kernel.

References

[1] Mendoza, P., Vila, J., Ripoli, I., Terrasa, S., Perez, P., "Developing CAN based networks on RT-Linux," in *Proc.8th IEEE Int Conf on Emerging Technologies and Factory Automation,* Antibes-Juan les Pins, France, *Oct 15-18, 2001, pp. 161-167.*

[2] Kiszka, j, "The Real-Time Driver Model and First Applications," in *Proc. 7th Real-Time Linux Workshop*, Lille, France, 2005.

[3] Zuberi, K.M., Shin, K.G., "Non-Preemptive Scheduling of Messages on Controller Area Network for Real-Time Control Applications," in *Proc. Real-Time Technology and Applications Symposium,* Chicago, USA, May 15-17, 1995, pp. 240-249.

[4] "The SocketCAN project", http://developer.berlios.de/projects/socketcan/

[5] Sojka, M., Pisa, P., Petera, M., Spinka, O. And Hanzalek, Z., "A Comparison of Linux CAN Drivers and their Applications," in *Proc. Int. Symposium on Industrial Embedded Systems (SIES)*, Trento, Italy, July 7-9, 2010, pp. 18-27.

[6] Wapice Ltd., "*WRM247 technical data*" http://www.wapice.com/wapice_cms/files/WRM_247_rgb_full.pdf, 2011, 2 pp.

[7] Microchip Technology Inc., "*Microchip MCP2515 Datasheet*", DS22187F, 2010, 88 pp.