Real-time communication: Part IV - CANopen source code configuration

(Source: Embedded Systems Academy)

Within this article series, Olaf Pfeiffer from Embedded Systems Academy (EmSA) is setting in perspective the timing requirements for different real-time capable communication systems, such as CAN, CANopen, and real-time Ethernet.

In this fourth and last article in the series, the author shows which optimization options are typically available when working with CANopen source code, here, on example of Micro CANopen Plus from EmSA. The Part I, published in the December 2023 issue of the CAN Newsletter magazine, describes how to select the right real-time timeframe for certain applications. The Part II, published in March 2024, shows the different timeframes required by different applications and reviews what this means for the communication system used. In Part III (June 2024), the author examines the real-time capabilities and limitations of CAN and CANopen in greater detail.

Throughout this article, the author will examine specific examples illuminating the process of configuring, optimizing, and fine-tuning a CANopen stack to cater to advanced temporal requirements and possible system constraints. While the focus is on Micro CANopen Plus from EmSA, the explored principles and methods are likely to work with other CANopen source code implementations in a similar way.

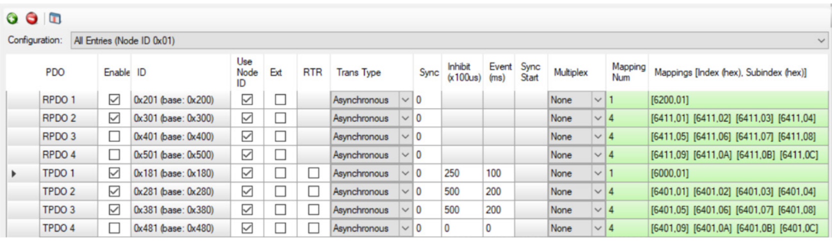
This part, translating the theoretical knowledge into practical implementation, aims to provide comprehensive insights and guidance for experienced CAN system designers and newcomers.

CANopen PDO configurations

The configuration of CANopen PDOs (process data objects) plays a critical role in determining the response time in various applications. Depending on the required response time and the necessity to synchronize signals across multiple nodes, different PDO triggering mechanisms can be applied.

Response time of 100 ms: For applications where the required response time is 100 ms or longer, there are typically two configuration methods that work well (and can also be combined):

- PDO triggering by event time (cyclic transmission): Here, the PDOs are transmitted cyclically at specified time intervals, such as every 50 ms. This periodic transmission ensures consistent response times.
- Change of state (COS) detection with inhibit time: This configuration transmits PDOs based on changes in state, with a minimum time (inhibit time) between transmissions. This inhibit time ensures that a toggling input does not produce back-to-back messages.



Configuration: All Entries (Node ID 0x01)

PDO	Enable	ID	Use Node ID	Ext	RTR	Trans Type	Sync	Inhibit (x100us)	Event (ms)	Sync Start	Multiplex	Mapping Num	Mappings [Index (hex), Subindex (hex)]
RPDO 1	☑	0x201 (base: 0x200)	☑	☐		Asynchronous	0				None	1	[6200,01]
RPDO 2	☑	0x301 (base: 0x300)	☑	☐		Asynchronous	0				None	4	[6411,01] [6411,02] [6411,03] [6411,04]
RPDO 3	☐	0x401 (base: 0x400)	☑	☐		Asynchronous	0				None	4	[6411,05] [6411,06] [6411,07] [6411,08]
RPDO 4	☐	0x501 (base: 0x500)	☑	☐		Asynchronous	0				None	4	[6411,09] [6411,0A] [6411,0B] [6411,0C]
TPDO 1	☑	0x181 (base: 0x180)	☑	☐	☐	Asynchronous	0	250	100		None	1	[6000,01]
TPDO 2	☑	0x281 (base: 0x280)	☑	☐	☐	Asynchronous	0	500	200		None	4	[6401,01] [6401,02] [6401,03] [6401,04]
TPDO 3	☑	0x381 (base: 0x380)	☑	☐	☐	Asynchronous	0	500	200		None	4	[6401,05] [6401,06] [6401,07] [6401,08]
TPDO 4	☐	0x481 (base: 0x480)	☑	☐	☐	Asynchronous	0	0	0		None	4	[6401,09] [6401,0A] [6401,0B] [6401,0C]

Figure 1: PDO configuration with CANopen Architect (Source: EmSA)

Smaller response times or synchronized signals across multiple nodes: For applications requiring smaller response times or where there is a need to synchronize signals across multiple nodes, the Sync mode becomes the preferred method:

- Sync mode: In this configuration, one Sync producer generates a Sync CANopen message at stable, repeating intervals, e.g., every 10 ms. This Sync message serves as a triggering mechanism, used by all devices in the network to apply data synchronously, at the same time.

Advanced Sync usage: When using the CANopen Sync mode, there are two advanced features to take advantage of. First, most Sync consumers allow the configuration of the Sync CANopen message identifier to be used. So, one could configure a system to use multiple Sync trigger messages and select which devices react to which trigger. Secondly, the latest version of CANopen supports the use of Sync with an integrated configurable counter. As an example, this could be configured to count until 4. On the Sync consumer side, one can configure the count value that each consumer listens to, again providing the option of grouping devices to react on specific Syncs on the system.

Choosing the right PDO configuration is vital for achieving the lowest response times. While cyclic transmission and COS (change of state) detection with inhibit time are suitable for more relaxed response time requirements, the Sync mode becomes essential when handling tighter time constraints or needing to synchronize multiple devices. More details on these different trigger mechanisms and how they can be combined is shown in a video by EmSA.

Generic data flow in a CANopen protocol stack

On the lowest hardware level, a CAN controller will be receiving CAN frames. Depending on filters, these might be placed into pre-selected buffers or queues, and an interrupt signal will be generated. The processor handling the CAN controller starts processing the "CAN receive interrupt service" – typically part of a processor-specific driver. A generic driver will now simply pass on the CAN data to another software queue for later processing; Advanced options will be discussed later.

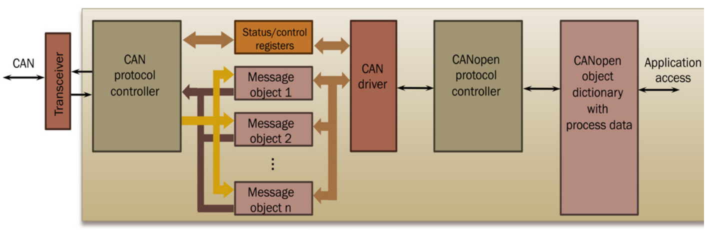


Figure 2: CANopen data flow (Source: EmSA)

At any time, the application program might update some of the process data to be transmitted via CANopen. To keep the CANopen stack alive, there will be a function, such as "ProcessStack()", that needs to be called frequently (for example, simply in a "main while(1) background loop"). When called, this function typically first checks if CANopen messages were received; if so, they are processed. If the data involves updating process data, then there is typically a callback to inform the application that new process data has arrived.

When all received CANopen messages are processed, the function checks if there is anything to transmit. It may detect that outgoing process data was modified by the application, and depending on the configuration of timers and transmission mode, initiate the transmission of a corresponding CANopen message. Such transmissions are typically passed to the driver level, possibly into a transmit queue, and it depends on the driver configuration when exactly this CANopen message will be passed to the CAN controller for transmission.

Basic configuration and control options

Unless the required processing and response time is smaller than 100 ms such a data flow works good enough for most applications. If required response times get smaller, you should start looking into possible optimizations. When reviewing the generic data flow above, possible optimizations include the CAN driver optimization for receive, CAN driver optimization for transmit, and optimization of "ProcessStack()".

CAN driver optimization for receiving: Many default drivers supplied by chip manufacturers (or even CANopen stack providers) might not take full advantage of the specific features of a CAN controller. One of the first possible optimization checks is to ensure that, where possible, hardware receive filtering and hardware receive buffers or queues are utilized, thus eliminating the need for a long (delaying) software receive queue.

CAN driver optimization for transmission: Before reviewing this, consider what is more critical to your application – whether this device can transmit as many CAN frames back-to-back as possible, or whether the transmission should be somewhat throttled to ensure that no single device can produce too long a period of high-priority back-to-back traffic. If it should be throttled, consider implementing a transmit trigger based on a timer, such as being able to transmit one CAN frame every millisecond (EmSA's default driver uses the 1-ms timer interrupt for this).

Considerations for "ProcessStack()": A typical question regarding "ProcessStack()" is how often it should be called and what the worst-case execution time is. Some prefer to call it from a fixed timer interrupt instead of the background loop. There is no generic answer to these questions. In the Micro CANopen Plus implementation, we try to keep the execution time short by not executing all pending CANopen tasks but only the most vital ones. How often it should be called depends heavily on the local device's communication. The Micro CANopen Plus implementation, however, has a slick feature here: with every call, only the most critical pending tasks of the CANopen stack are performed. Producing the heartbeat message is always the least important task. Therefore, by monitoring the device's heartbeat signal's accuracy, you can determine if calls to "ProcessStack()" occur often enough. If there are not enough calls to "ProcessStack()", the heartbeat becomes slower than specified, or it may not be transmitted at all.

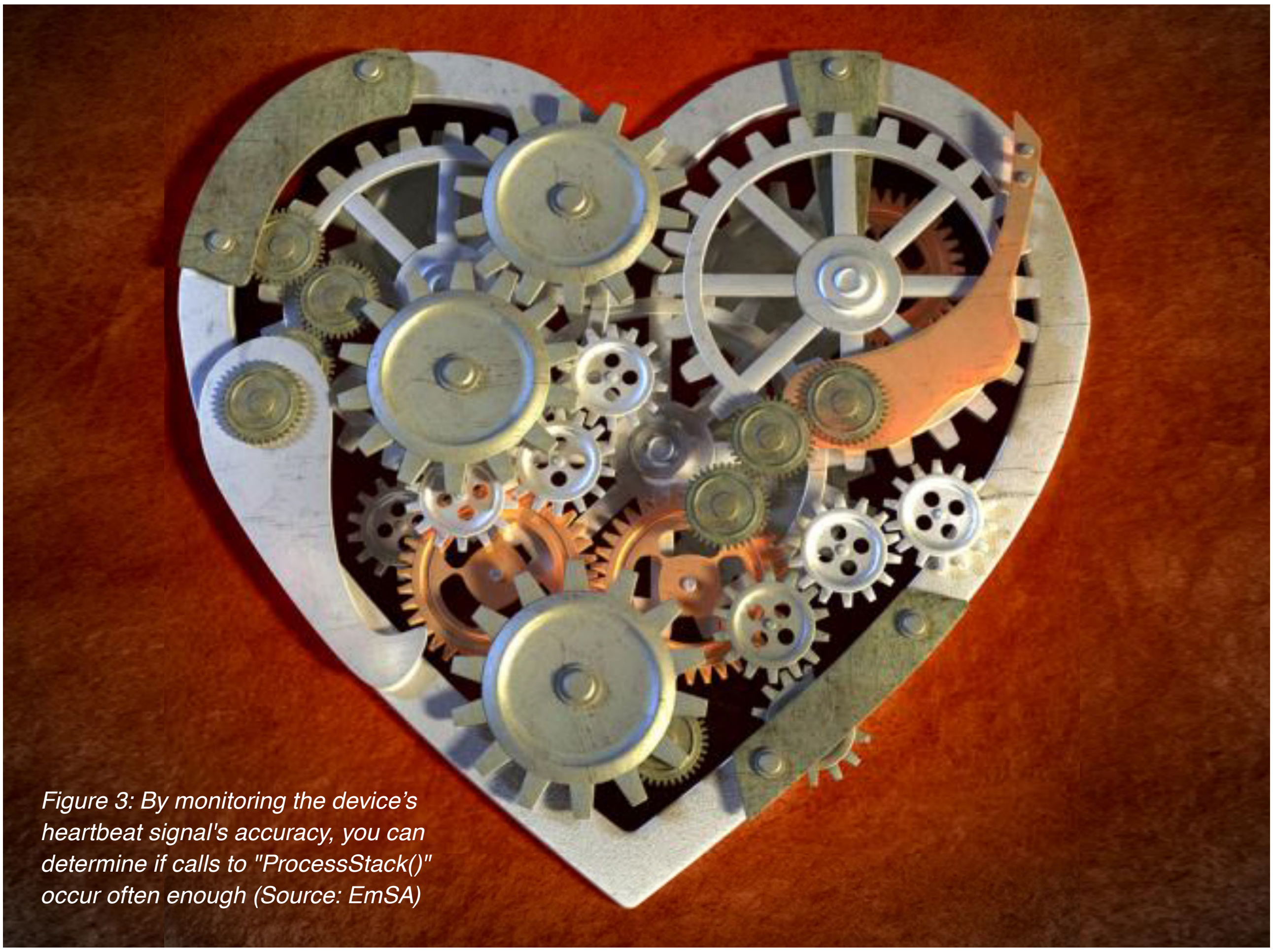


Figure 3: By monitoring the device's heartbeat signal's accuracy, you can determine if calls to "ProcessStack()" occur often enough (Source: EmSA)

Note on return value for "ProcessStack()": Another important factor is the "ProcessStack()" function's return value. It returns TRUE when a pending CANopen task was executed, and FALSE when there is no CANopen task pending. If you want to ensure that all pending CANopen tasks are executed in your code, use:

```
while(ProcessStack())
{
}
```

This will keep re-calling the function until all pending CANopen tasks have been executed.

Direct task trigger: The function "ProcessStack()" serves those who prefer not to go into the details of all the CANopen tasks executed from within. For further optimization, an application can bypass calling this function and directly invoke the dedicated CANopen tasks: "ProcessStackRx()" and "ProcessStackTick()".

- ***Sub-task "ProcessStackRx()":*** This task handles processing a received CANopen message. For an optimized call, it would ideally be initiated directly from the CAN receive interrupt or triggered by some signal set in the CAN receive interrupt.
- ***Sub-task "ProcessStackTick()":*** This task checks if the process data to be transmitted has changed (or was triggered for transmission) and if any actions based on the millisecond-timer need to be taken. The most efficient way to call this is only after process data has changed or the millisecond-timer has incremented.

This approach provides a more refined control over the execution of specific tasks within the CANopen stack, allowing for more precise tuning of performance and responsiveness.

Bringing together CAN driver, CANopen stack, and application

On most 32-bit-based micro-controllers, the enhancements discussed so far are suitable for bringing the total response time down to a range of 10 ms to "a few milliseconds." This can be achieved without requiring optimizations that lead to a fully custom implementation that might be challenging to maintain. These optimizations were confined to leveraging individually triggered CANopen stack processes when needed. In general, this can be taken further. However, making changes at such an intrinsic level in a system can make it much more challenging to maintain or port to a different architecture when necessary. Therefore, the following is more to illustrate "what is theoretically possible," pushing optimizations beyond the point where a system remains easy to test, maintain, and port.

On the lowest hardware level, review if your CAN controller is configured to directly create a CAN receive interrupt with the reception of the Sync message and if you can easily detect the difference to any other CANopen message (e.g. own filter/receive buffer).

The only reasonable CANopen PDO communication mode for further real-time improvements would be the CANopen Sync mode. If that is used and we concentrate on ▷

optimizing it, then the previous other optimizations might become redundant.

Focusing on Sync optimization would require to modify the CAN receive interrupt service routine to directly call the CANopen stack function(s) responsible for Sync handling when a Sync signal is received. In the case of Micro CANopen Plus, this would be the function "HandleSync()". When executing this from within the interrupt service routine, please keep in mind:

- Not to store this Sync in the regular receive queue (we already process it).
- That for both Sync-related transmit and receive data, call back functions to the application will be called – still executing at the interrupt service level.
- When using an RTOS (real-time operating system), a better solution would be to set a trigger signal in the interrupt that Sync was received, subsequently triggering the execution task immediately after the interrupt has completed.

With such a modification, a response time within a millisecond is achievable. If all devices participating in the Sync communication implement the Sync handling with equal optimization, the variation among the devices (e.g. when they each apply their outputs synchronously) can be as low as a few microseconds. Nevertheless, these are extreme values that have been observed to work in test and lab environments. For real-world applications demanding such short response or sync times, careful testing would be required to ensure that these targets can be reached under all realistic circumstances.

Final conclusion

The journey through this four-part series has provided an in-depth exploration of the components that shape embedded real-time communication. From hardware selection to advanced optimization techniques, the underlying theme that resonates is the pivotal role of response times in determining every aspect of system design and configuration.

1. *Hardware selection:* The required response times dictate the hardware capabilities needed, influencing decisions on modules, possibly micro-controllers and other essential components.
2. *Operating system considerations:* Whether working with an RTOS or implementing a more specific, bespoke system, the response times heavily influence how the operating system needs to be configured.
3. *Network technologies:* Depending on the required throughput and speed, different network protocols and technologies must be taken into consideration. As an example, this series looked at the specifics of CANopen and its configurations, illustrating the nuanced choices required to meet different application demands.
4. *Optimization choices:* Perhaps one of the most profound insights is the realization that optimization is not a one-size-fits-all approach. Depending on the required response times, certain optimizations become essential, while others can be bypassed. It's a matter of fine-tuning, understanding what needs to be harnessed, and what can be left untouched without affecting performance.
5. *Strategic ignorance:* Contrary to the instinct to utilize every possible advantage, there are instances where the time frame allows for the deliberate ignoring of certain optimizations. Not every register provided by a network controller needs to be exploited; it's a balance between performance and the demands of the particular application.

Through this series, the author has illuminated the complex interplay of hardware, operating systems, and network technologies, all governed by the essential factor of response times. The insights offered serve as a guide for making strategic choices in system design, highlighting the importance of tailored optimization and thoughtful decision-making. These principles enable you to craft robust and efficient real-time communication systems, suited to your application demands. ◀

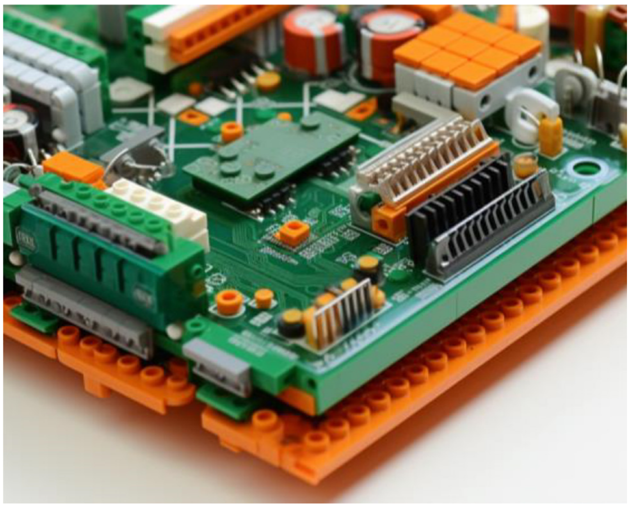


Figure 4: From hardware selection to optimization techniques, the underlying theme that resonates is the pivotal role of response times in determining every aspect of system design and configuration (Source: EmSA)

Author

Olaf Pfeiffer
EmSA (Embedded Systems Academy)
info@esacademy.com
www.esacademy.de

Engineering