Thread prioritization for an embedded CANopen master stack with web interface

D. M. Armenis and J. S. Smith
Department of Electrical Engineering and Electronics
The University Of Liverpool

As Tele-robotics continuously receives industrial attention, an embedded CANopen Master Stack, accessible via the Internet, is proposed. A configurable real-time operating system permits multi-threaded development whilst PLD technology ensures system evolution. The impact thread prioritization has on system performance is investigated, whilst initial design considerations are also presented.

Introduction

Multi-threaded, real-time, control processes introduce synchronization and timing issues, unknown to many embedded applications [1,2]. The scope of the presented work is both to identify the reasons for prioritizing control over safety threads and to determine their minimum request period during normal operation of an embedded CANopen Stack with Web Interface. The results of this study provide a clear indication of the schedulability of the proposed system. The determinism of the stack can also be improved, a required property for bridging real-time Ethernet [3] with CAN. Initially, the Stack was composed of two transceiver threads, three communication threads and two databases implemented in eCos [4,5]. Three monitoring threads, for guarding the functionality of the two required Motor CANopen (Drive) Slave Nodes and the optional CANopen Encoder Slave Node [6,7,8], were added to the second version. In its current version three control threads and two user interface threads, were added as shown in figure 1. The developed system has been applied to an Unmanned Underwater Vehicle (UUV) [9].

Timing Issues

Extension of the system to accommodate more slave devices, raise some very interesting synchronization questions. Suppose that a network is composed of slave nodes incapable of serving more than two interrupts levels (e.g. slave nodes based on PIC18F453 devices). In the two mandatory slave (Drive) nodes, the higher priority interrupt is used as a pulse generator. It is responsible for distributing a pulse, with fixed width, to the stepper motors commanding them to advance. The lower priority interrupt is used to transmit the EMCY PDO during an emergency. In the Encoder Node, the last slave node in the system, the higher priority interrupt is used as the SYNCH producer for the network. The lower priority interrupt is used for the EMCY event as in the Drive Nodes. As a result, both time depended protocols and CAN bus receivers are left unsupported. While the CAN bus registers can be monitored by a polling procedure, the mandatory Error Control Services can only be realised by the CANopen Master under the Node Guarding Protocol. As expected, the Node Guarding event can be attained by a timed, interrupt-based operation. Nevertheless, the delay introduced by the pooling procedure, at the slave nodes, will manifest certain jitter effects during control directives, which deem this approach inappropriate for the proposed system. Consequently, the Node Guarding Protocol was accommodated in the stack, implemented under the one-thread-per-slave philosophy, with a lower priority than the control threads. Explicitly in this case, the designer is facing periodic, hard, real time schedulability issues as a consequence of delays introduced by the control threads duration throughout normal manoeuvring. It is therefore vital to identify the relationship between the *Guard Slave Thread Priority (GSTP)* and *Life Time Value (LTV)* [8] in an attempted to decouple the system control effort from the Node Guarding Protocol.

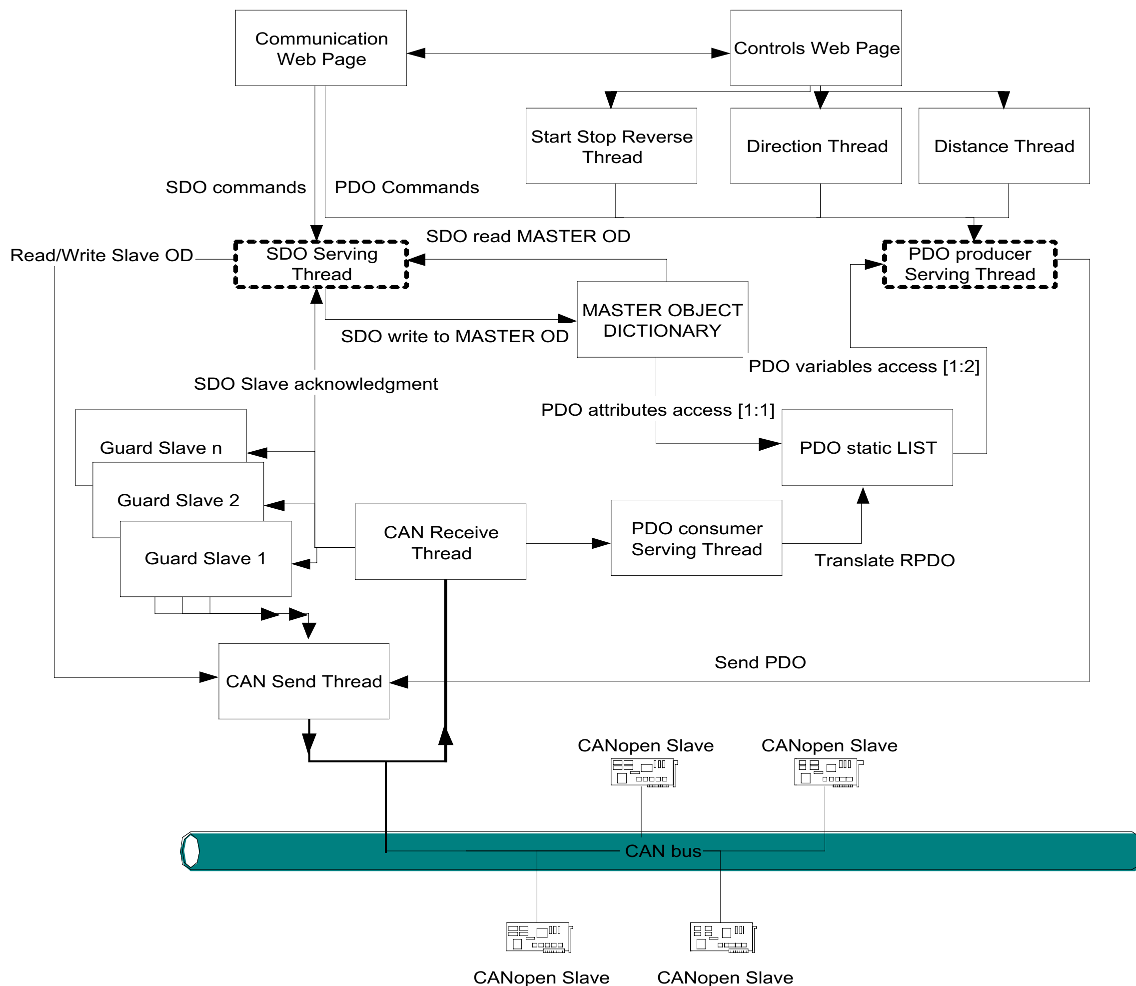


Figure 1: Run-Time Threads

Leung [10] states that a decrease in the *GSTP* (priority) can be achieved by increasing the *LTV* (deadline) under the notion of the deadline monotonic scheduling scheme. As such, a considerable increase in the Node Guarding Time (period) would have been able to accommodate the maximum system delay. Then again, a defective node must be identified instantly, in an attempt to contain the problem and ensure the system's graceful degradation, this is provided by a small Node Guarding period. In the current stage of the UUV development, where there is no Deliberative Module, the user can send low level commands to the robot on an irregular basis making the whole system sporadic. Assuming a fixed time interval where at least one command will be issued, the system becomes periodic. Therefore the proposed work differentiates slightly from the Node Guarding Protocol by permitting the Node Guarding Remote Transmit Requests to occur at random intervals during the Node Life Time. This optimizes the Node Guarding time to the best functional time interval. In the following sections, two guarantees are proposed that ensures at least one Remote Transmit Request before the Node Life Time expires. Both approaches try to minimize the *LTV* until the system reaches the CPU's maximum utilization limit.

The remainder of the paper is organized in the following manner. Synchronization Techniques for the Node Guarding Threads are identified in the next section. This is followed by the Priority Assignment Calculations section where the schedulability of the system is tested. The methodology for the proposed tests is then presented. Subsequently, in the Discussion section, the worst case results are considered. The paper concludes by highlighting the importance of this work to the field of UUVs.

Synchronization Techniques

Assuming one or more slave nodes, the developer might face the dilemma of organizing the Node Guarding Threads in one of four different ways as described in table 1.

Table 1: Synchronization Methods

Synchronization Methods	Same *GSTP* for all slaves	Same *LTV* for all slaves
I		
II		_
III	_	
IV	_	_

I. In this method, the *GSTP* can be decreased by increasing the *LTV* individually for each node. Neither the priority nor the *LTV* is the same for any slave as shown in figure 2. Assuming that the constant C_{guard}, is the time necessary for a guard slave thread to be completed. If no higher priority thread is active this is subject only to code length and the processor speed,. It is also the same for all guard slave threads. Let *n* be the number of all mandatory Node Guarding Threads during the application. The minimum period for the highest priority thread is $T^{\min} = n \cdot C_{\text{guard}}$ so that there is enough time for every thread to be completed before the first thread is overdue. Let *D* be an unexpected system delay (i.e. a higher priority thread becomes active) such that $C_{guard} < D < T^{\min}$. Then for every thread *p* with:

$$n \cdot C_{guard} > p \cdot C_{guard} + D \qquad (1)$$

where $p \in [1..n]$, the minimum *LTV* can be equal to the minimum period T^{min}. Alternatively, the minimum *LTV* is:

$$LTV_p^{\min} = D + (n + p - 1) \cdot C_{guard} \qquad (2)$$

This equation is subject to the time the delay occurred. As the delay approaches the lowest priority threads, the minimum *LTV* increases for all the system except for those threads that satisfy relationship (1).

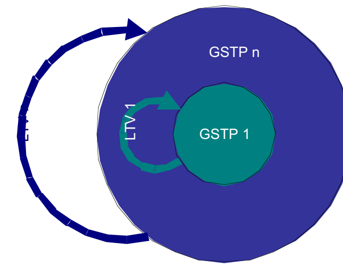


Figure 2: Sync method I

II. In this scenario the nodes all have the same LTV_{min} but a different *GSTP*. This case is illustrated in figure 3. The difference from method I is that all threads must satisfy equation (3), to ensure normal operation during a delay.

$$LTV^{\min} = D + (2n - 1) \cdot C_{guard} \qquad (3)$$

Although (3) makes the system independent of the time the delay occurs, it introduces a uniform approach for the *LTV* which might not be desirable in some implementations.

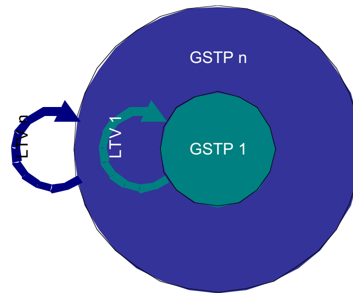


Figure 3: Sync method II

III. In the third method all the threads have the same priority. Again the minimum *LTV* is $LTV_{\min} = n \cdot C_{guard}$ but many threads do accommodate larger values as shown in figure 4. In an event of a system delay only the threads that have $LTV < C_{guard} \cdot a + D$ will fail regardless of the time that the delay occurred.

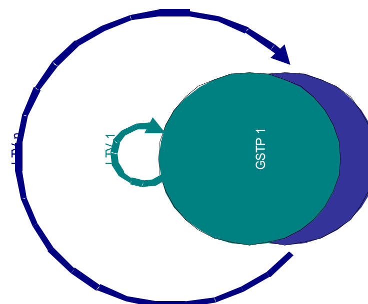


Figure 4: Sync method III

IV. In the last method, all the threads are sharing both the same priority and the same *LTV* as illustrated in figure 5. If the delay satisfies the relationship $n \cdot C_{guard} < D$, all threads will fail but if $n \cdot C_{guard} \geq D$ all threads will succeed.

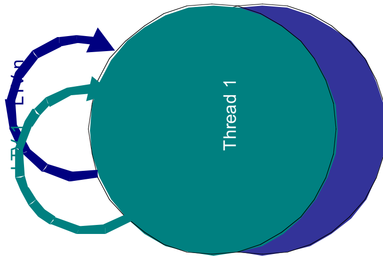
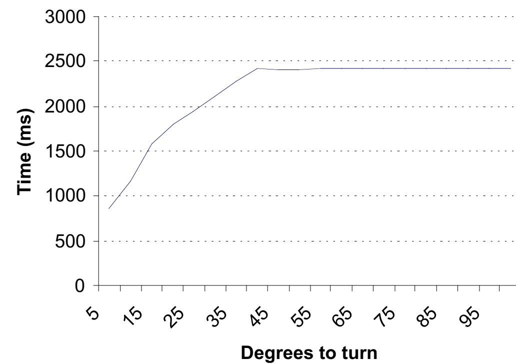


Figure 5: Sync method IV

Priority Assignment Calculations

Timing trials of the control application determined the direction deadline on the UUV system. This is the maximum time of processor usage from the higher priority threads. It was found that 4350 ms was necessary for the UUV robot to execute a turn of 45 degrees from rest. This is the maximum permitted rotation angle for the vehicle due to its dynamics constraints. The two graphs shown in figure 6 illustrate the introduced delays. The worst case scenario is covered under the assumption that the CAN bus is inaccessible to the Guard Slaves Threads for the duration of the turn. In any other case the delay is reduced considerably but jitter effects may appear to the Drive Nodes. The timing of the deadline is mainly related to the acceleration and deceleration coefficients of the motor drives. Figure 6 depicts the calculated time duration for turning the vehicle with the current settings of the acceleration and deceleration coefficients. There are two main characteristics in such a system that must be accounted for in an accurate schedulability analysis.

1. There are only two groups of threads running during normal operation, the control threads and the Guard Slave Threads. They are not related and their access to the CAN bus is buffered under a FIFO policy, therefore they are asynchronous.
2. They are both sporadic in the sense that their execution interval is related to the user's unpredictable input but they will not be executed more than once during their deadline.



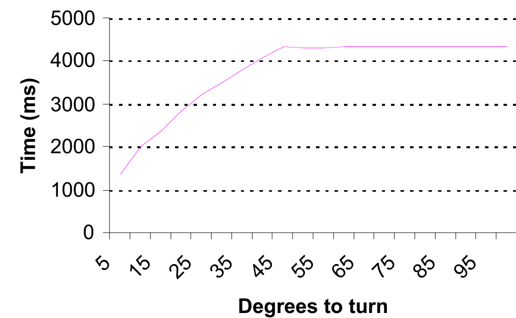


Figure 6: Turn duration travelling at full speed (top) and from rest (bottom)

Methodology

To obtain the necessary values for the tests, a hardware timer, normally the one used to drive the real-time clock, was used. Generally, this timer can be read with a resolution typically in the range of a few _s. For each measurement, the operation was repeated a number of times. Time stamps were obtained at the beginning and at the end of each thread and they were analyzed, generating average (mean), maximum and minimum values. The sample variance (a measure of how close most samples are to the mean) was also calculated.

According to [11,12] the schedulability of a system can be measured by simple procedures, based upon the concept of *critical instants* [1]. Namely, they represent the times when all processes are released simultaneously. For a system, this is the time of the worst-case processor demand. If all threads can meet their deadlines at a critical instant then the system is said to be schedulable. For the system to be

schedulable any of the following equations must hold.

$$\frac{I_{guard}^{t_0}}{t_0} + \frac{C_{guard}}{t_0} \leq 1 \quad (4)$$

where $t_0 = C_{dir} + C_{guard}$

$$\frac{I_{guard}^{t_1}}{t_1} + \frac{C_{guard}}{t_1} \leq 1 \quad (5)$$

where $t_1 = I_{guard}^{t_0} + C_{guard}$

With $I_{guard}^{x} = \left\lceil \frac{x}{T_{dir}} \right\rceil \cdot C_{dir}$ (6)

and $x \in [t_0, t_1]$

Where *x* is the space of critical instances, the times when the threads are being released.

Given that the Direction Thread must have a higher priority than the Guard Slave Threads, if the later is not to interfere with the control procedure, the deadline of the Direction thread must be less than the deadline of the Guard Slave Threads. The following properties were therefore assigned to the study:

For the Direction Thread:

Direction Thread Computation Time
(C_{dir}) = deadline (D_{dir}) < period(T_{dir})
C_{dir} = 4350ms,
$C_{distance}$= 50ms,
$T_{dir} = (C_{dir} + C_{distance})$

By incrementing the period of the Direction Thread beyond its deadline, by a time equal to the Distance Thread computation time, distance commands can be issued between turns.

For the Guard Slave Thread:

Guard Thread Computation Time
(C_{guard}) < deadline (*LTV*) = period
C_{guard} = 40ms

Substituting the above values into equation (4) and (5) proves that the system satisfies the equality for one Guard Slave Thread, therefore it is schedulable with its current priorities assignment. If more slaves are added to the system then for n Guard Slave Threads we have:

$$I_{guard}^{x} = \left\lceil \frac{x}{T_{sys}} \right\rceil \cdot C_{dir}$$

where $T_{sys} = (C_{dir} + C_{distance} + (n-1) \cdot C_{guard})$

Combining the above equations with equations (4) and (5), the schedulability of the system is proved once more.

Discussion

With reference to the aforementioned synchronization techniques, the advantages of having the same priority and different life time values can be clarified. When the Guard Slave Threads have different priorities, the system is prone to the duration of the delay as well as to the time the delay occurred. On the other hand when the same priorities are assumed the system performance depends solely on the duration of the interrupt.

In real-time operating systems, such as eCos [5], it is possible to calculate the maximum delay of the system but it is impossible to determine the time of the occurrence due to user unpredictability. Additionally, devices of different functionality often have different requirements for their life circle, thus each thread can be tailor-made to map the requirements of the relative slave node. Following the worst case delay of the UUV system and assuming full exploitation of the CAN bus with 127 slave nodes, the calculated LTV_{min} is given by:

$$LTV_{\min} = 14.47\,\text{sec} \quad \text{for n} = 127 \quad (7)$$

In any case this is the theoretically maximum delay. If the acceleration and declaration coefficients were increased then the *LTV* will be reduced due to the quicker response from the slave. Similarly if they were decreased, the improved idle time for the direction thread could result in a lower bus load, therefore permitting the Guard Slave Threads to run even sooner.

Conclusion

The work presented was motivated by research in the autonomous UUV field. In such systems the early diagnosis of faulty nodes can result in quick recovery procedures thus avoiding damaging the vehicle or losing it into the sea. Several threads with different priorities are common to this kind of embedded application so precise timing is always a concern. Consequently a customized CANopen Mater Stack was developed and implemented by means of both PLD technology and a real-time, multi-threading operating system. Industrial CANopen Master Stacks [13], can benefit from the functionality introduced by this approach. Among other developmental variations, the Node Guarding capabilities were realized as individual threads, sharing the same priority but with distinctive tailor-made Life Times as is dictated by the second synchronization paradigm. It is believed that in the near future similar applications will appear in the emerging field of Tele-embodiment [14,15].

References

1. Liu, J. and Lee, E. (2003): "Timed multitasking for real-time embedded software", IEEE Control Systems Magazine, 23:1.
2. Wittenmark B., J. Nilsson and M. T rngren. (1995) "Timing Problems in Real-Time Control Systems". Proc. Of the American Control Conference. US.
3. Schneider S. *Making Ethernet work real-time*. Sensors Magazine, 2000.
4. Dallaway, J., Garnett, N., Larmour, J., Lunn, A., Thomas, G., Veer, B., (2004), "*RedBoot User's Guide*", Red Hat Inc.
5. Garnett, N., Larmour, J., Lunn, A., Thomas, G., Veer, B., (2003), "*eCos Reference Manual*", Red Hat Inc.
6. CAN in Automation e.V., "CANopen-Device Profile for Encoders", CiA Draft Standard Proposal 406, Version 2.0.
7. CAN in Automation e.V., "CANopen-Device Profile for Drives and Motion Control", CiA Draft Standard Proposal DSP-402, Version 1.1.
8. CAN in Automation e.V., "CANopen-Application Layer and Communication Profile", CiA Draft Standard 301, Version 4.01.
9. Evans J.C., Smith J.S., Martin P., and Wong Y.S. "Beach And Near-Shore Crawling UUV for Oceanographic Measurements" IEEE Int. Conf. Oceans '99 Seattle, USA. pp 1300-6
10. Leung, J.Y.T., and J. Whitehead. (1982). "*On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks.*" Perf. Eval. (Netherlands), 2, pp. 237-250.
11. Audsley, N.C., Burns, A., Richardson, M.F., and Wellings, A.J., (1991) "*Hard RealTime Scheduling: The Deadline Monotonic Approach*." In Proc. of the Eighth Workshop on Real-Time Operating Systems and Software, pages 133--138, Atlanta, GA, USA.
12. Audsley, N.C. (1990). "*Deadline Monotonic Scheduling*." YCS 146, Dept. of Comp. Sci., Univ. of York.
13. Etschberger K., Schlegel C. *CANopen-based distributed intelligent automation systems*, Proceeding of the 8th International CAN Conference, Las Vegas, USA, 2002.
14. Goldberg K., Siegwart R., *Beyond Webcams: An Introduction to Online Robots*, MIT Press, 2002.
15. Goldberg K., *The Robot in the Garden: Telerobotics and Telepistemology in the Age of the Internet*, MIT Press, 2000

Dimitris Armenis
Department of Electrical Engineering and Electronics.
The University of Liverpool
Brownlow Hill
Liverpool, L69 3GJ, UK
Tel: 44 151 794 4602
Fax: 44 151 794 4540
E.Mail: D.Armenis@liverpool.ac.uk

Jeremy S. Smith
Department of Electrical Engineering and Electronics.
The University of Liverpool
Brownlow Hill
Liverpool, L69 3GJ, UK
Tel: 44 151 794 4514
Fax: 44 151 794 4540
E.Mail: J.S.Smith@liverpool.ac.uk