# Logical Execution Time in the Automotive Environment
## Introduction and Application

Martin Alfranseder, Stefan Kuntz, Martin Kardos, Ralph Mader;
Continental Automotive GmbH

**Within the next years, EE architecture in automotive systems will change significantly. Domain controller platforms are introduced that combine different kinds of microcontrollers equipped with several cores and replace ECUs that are used nowadays. System topologies change from simple networks to hierarchical networks. Furthermore, features like automated and autonomous driving lead to the fact that functions, which formerly were decoupled from each other, get more and more tightly connected and distributed. In addition, there are still parts of software that are only capable of running on a single core and shall be enabled for multicore based systems. Such functions and complex chains of effects demand stronger timing requirements that are getting more and more difficult to handle. Thus, new methods like Logical Execution Time (LET) are required to ensure all these aspects. This paper briefly describes the concept of LET and presents an overview of different implementations in practical Powertrain applications. Finally, the current status of introducing LET in the AUTOSAR standard is given.**

### Motivation

All the challenges and problems described above can be summarized as follows: The relevant behavior of real-time programs is determined by when input is read and output is written and not when and where programs execute any code [1]. This is the main motivation behind LET which abstracts from physical execution of functions. LET can be an enabler to improve the hardware platform independence of software components. Further, it can lead to better stability in chains of effects and support parallelization of partly parallel code [2, 3].

### The paradigm of LET

LET was introduced in 2000 by Prof. Thomas Henzinger and his group in the Electrical Engineering and Computer Sciences (EECS) department at the University of California, Berkeley [4]. LET is an abstraction from the physical execution time of a real-time program. Hereby, the observable temporal behavior of a task is independent from its physical execution. It is only assumed that physical task execution is fast enough to fit somewhere within the logical start and end points. Thus, a LET must be designed greater or equal than the Worst-Case Execution Time (WCET) of the contained functions. Note, that for dimensioning of LETs, the communication time between different cores or even different controllers shall be included to the WCET analysis in advance.

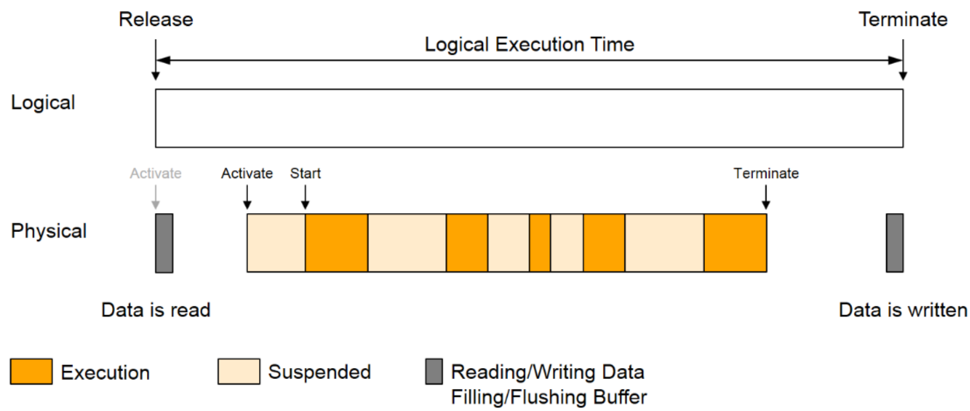


Figure 1: The Logical Execution Time for an instance of a task and its relation to the physical execution on a real-time environment.

Figure 1 shows the idea of the Logical Execution Time. LET does not care about when exactly instructions of a program are executed by a processor. In addition, a LET includes communication time which is necessary to exchange data across the system until the results of calculation get visible to all observers. The only two things that matter are on the one hand, that the input data remains stable from the release point of a LET until the terminate point; and cannot be altered in between. And on the other hand, the results of a calculation are only visible system wide at the corresponding terminate point. This leads to the fact that the functional behavior of the system is not statistical (non-deterministic) anymore, depending on the point in time of the calculation of its functions, but instead gets deterministic and therefore the systems behaves more robust.

**Ways of implementing LET**

There exist different possible solutions to realize LET behavior in real-time systems. An overview of the most known approaches is presented in this section.
For avionic systems, for instance, native OS solutions exist that ensure LET scheduling (activation and termination of tasks) and data exchange (buffering). The concept of LET is not yet established well in automotive systems and therefore not yet supported by AUTOSAR compliant operating systems.

A second method takes advantage of ring buffers. Reading and writing data takes place from/to a global ring buffer. LET behavior is ensured by altering the read and write pointers of such a buffer at the right points in time for read and write operations. Imagine, a LET gets processed. The result of calculation is written into position two of a ring buffer. But read accesses from other tasks are referred to the old value in position one of the buffer as long as the LET is not terminated. After termination of the LET the reference of read accesses changes to position two of the ring buffer. For more details on this approach see [5]. This approach gets close to its limits, if several LETs in a system have different periods and if memory is handled in a very fine-grained manner. Then, complexity for designing the ring buffer and manage the altering of the pointers rises rapidly.
A third alternative is that one can make use of features provided by AUTOSAR OS and implement so called Driver Tasks that manage reading and writing data to local buffers as well as activating physical tasks. Compared to the method that uses ring

buffers, this approach shows its weakness if LETs are getting very short. The overhead of executing Driver Tasks is the limiting factor, here.

**LET implementations at Continental**

Continental decided to make use of the third variant of the above described approaches to implement LET in powertrain systems. Dedicated OS-Task were introduced that ensure LET behavior for functions that are selected to follow this paradigm. Within this section, different types of LET implementation are presented. Note, that each of these implementation examples cover solutions on a multicore microcontroller. Nevertheless, LET can be applied for distributed systems or networks as well.

**End-Of-Period.** This was the very first approach of LET where different assumptions were made before. First, all LETs are time triggered and periodic. The duration of LETs equals their recurrence. Figure 2 shows an example schedule for this approach. For instance, a LET with a period of 10 ms has the duration of 10 ms. Further, all periods and durations of those LET have a common time base, which is 2.5 ms in this example which means that all other periods are a multiple of this period. Based on these assumptions, a periodic OS task called 'End-Of-Period' (EOP) was defined with the period of 2.5 ms that takes over the management of the LETs and the buffering of data. Whenever the EOP task is executed, it checks which LET is released or terminated at this point in time. For released LETs, buffers are filled with the regarding input data, whereas for terminated LETs buffers containing output data are flushed back to the main memory.

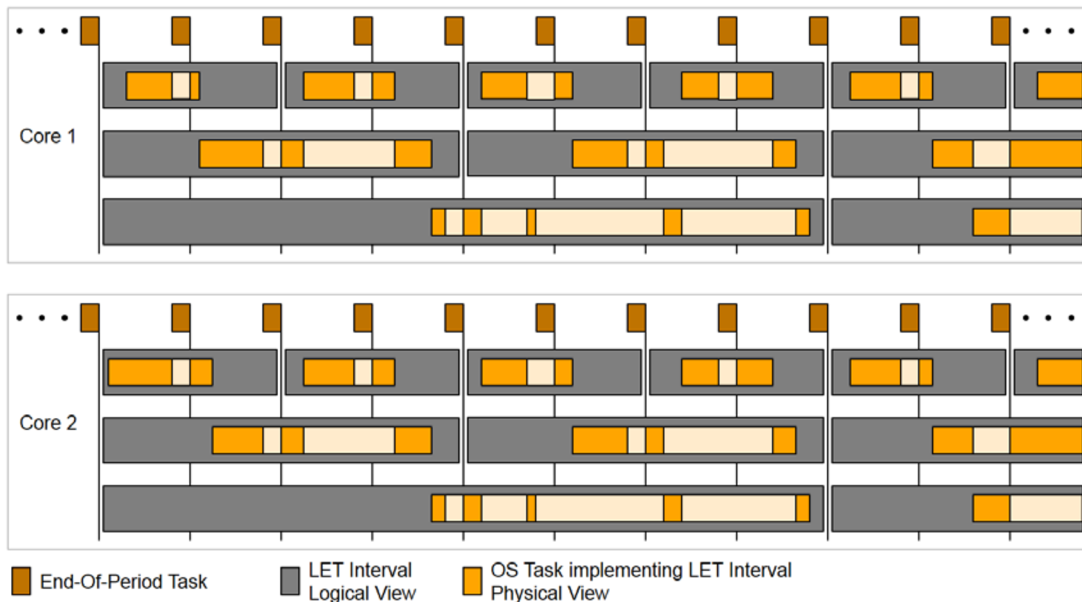


Figure 2: Realization of LET utilizing EOP tasks on two cores.

**TMachine[1].** In comparison to the approach described above, some assumptions can be neglected for this way of LET implementation. LETs still have time triggered periods, but it is not required any longer that their duration equals their period. There even might exist different LETs with same period and different offsets. Additionally,

[1] The concept of the TMachine originates from [6]

the period does not have to be a multiple of a base period any more, it can have any recurrence. Based on the configuration of LETs, a time-table per core is built. This time-table contains all points in time when a LET is released or terminated. Based on the entries in the time-table, interrupt service routines are triggered that activate a driver task. The driver task performs the buffer handling as well as the activation of OS tasks that are related to starting LETs.

In Figure 3 the temporal behavior of LET implemented by means of the TMachine is shown. The TMachine requests an interrupt event at the next point in time that is entered in the time-table. This can either be a release or a terminate of a LET. The TMachine checks which point in time-table is reached now and triggers the corresponding driver task. The driver task has the highest available priority in the system and gets executed immediately. It takes care about the buffer management and the synchronization towards other LETs released or terminated simultaneously, as well as the activation of LETs (implemented by OS tasks).

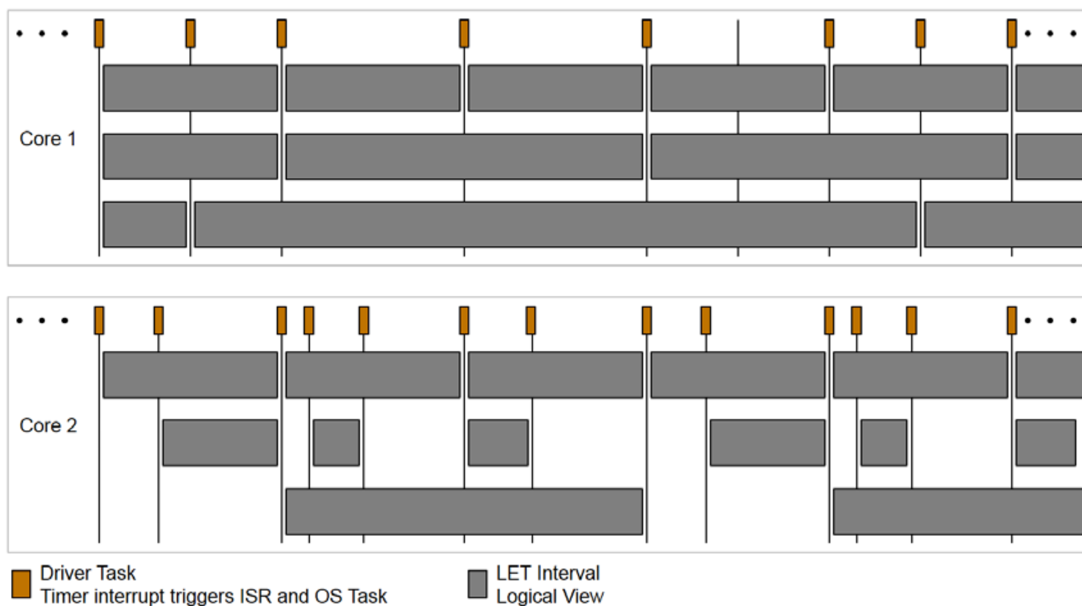


Figure 3: Implementation of LET by means of a TMachine.

This approach allows a much more flexible approach of LET, as many of the constraints of the EOP approach could be neglected. Nevertheless, there still is a drawback in this way of implementing LET, namely that the data buffering of LETs is optimized for LET tasks only. The buffering of non-LET task, interacting with LET tasks, cannot be optimized by this kind of implementation.

**LET with own OS-plugin.** Basically, the timing behavior of this approach is equal to that presented in Figure 4. The main differences are, that the interrupt events are not configured based on a time-table, but rather on a periodic execution scheme extracted out of the task configuration. Furthermore, and this is the most important difference, buffering is handled by a proprietary software integration tool (see PDA layer in Figure 3), so that the data-flow between the LET tasks and the non-LET task can be considered as well.

## Standardization of LET in AUTOSAR

Since early 2017 a concept is worked out in the AUTOSAR development partnership to provide support for Logical Execution Time. It is expected that the AUTOSAR Release 4.4.0, which is supposed to be released end of 2018 time frame, will provide means to specify LETs using the AUTOSAR Specification of Timing Extensions. In a first step the goal is to provide the capabilities to describe models and timing requirements for systems that shall follow the LET paradigm. This includes the specification of LETs for executable entities as well as the effects for communication between these executable entities.

## Summary

In this paper, we presented the motivation and introduction of LET paradigm. Further, we described different feasible implementations and shortly discussed the process of standardization in AUTOSAR.
Our experiences with the different implemented flavors of LET are summarized as follows: The End-Of-Period approach is easy to implement because it utilizes existing standard operating system capabilities without any change. Nevertheless, it is accompanied with strict constraints for designing LETs which are not suitable for every project. The second approach, where LET is fulfilled by a supplier addon to the operating system, gave us much more flexibility for the design of LETs. For instance, those LETs do not need to have periods on a common time base, and their recurrences do not have to be equal to their duration. The main limitation of this implementation approach is, that data exchange between LET and non-LET tasks is not optimized, as buffer handling for both is done by different tools. Finally, LET implemented by using our in-house developed OS plugin allows us to optimize the data flow between LET and non-LET tasks as well while keeping the same level of flexibility as in the previous approach. As a disadvantage of this approach, we have to mention that implementing the OS plugin containing all the flexibility and buffer handling opportunities requires higher effort and careful design.

Concluding, one can say that LET is an instrument that is able to overcome many of problems that arise for modern automotive applications. Determinism provides the key features for distributed real-time functions. It guarantees a more robust behavior of chain of effects by design. Besides these benefits, LET comes with some limitations and disadvantages: A WCET analysis including communication time is required in advance. This limits the theoretical available CPU utilization. Furthermore, additional hardware resources, like additional memory for buffering and CPU utilization for managing the release and termination of LETs, are necessary to apply the LET approach.

## Sources


[1] C. M. Kirsch and A. Sokolova, "The Logical Execution Time Paradigm," in *Advances in Real-Time Systems*, Berlin, Heidelberg, Springer, 2012.

[2] R. Mader, "LET Implementation in Classic AUTOSAR," in *Embedded Multi Core Conference*, Munich, 2018.

[3] R. Ernst, S. Kuntz, S. Quinton and M. Simons, "The Logical Execution Time Paradigm: New Perspectives for Multicore Systems," in *Report from Dagstuhl Seminar 18092*, 2018.

[4] T. Henzinger, B. Horowitz and C. Kirsch, "Giotto: A time-triggered language for

embedded," in *In Proc. International Workshop on Embedded Software (EMSOFT),volume 2211 of LNCS, pages 166–184*, 2001.

[5] P. Caspi, N. Scaiffe, C. Sofronis and S. Tripakis, "Semantics-Preserving Multi-Task Implementation of Synchronous Programs," 2007.

[6] S. Resmerita, K. Butts, P. Derler, A. Naderlinger and W. Pree, "Migration of Legacy Software Towards Correct-by-construction Timing Behavior," in *Proceedings of the 16th Monterey Conference on Foundations of Computer Software: Modeling, Development, and Verification of Adaptive Systems*, Redmond, WA, 2011.

## Authors

**Dr. Martin Alfranseder** joined Continental in 2015 after his PhD Thesis about multicore scheduling and synchronization in embedded multicore real-time systems as Software Project Architect. Since 2017 he is part of the software architecture group at Engine Systems which is part of the Powertrain division at Continental. His main topics are dynamic aspects in software architecture like scheduling or sequencing of functions on electronic control units. Further topics are formal description and automated processing of dynamic constraints and requirements and its adaptation to customer software.
martin.alfranseder@continental-corporation.com

**Stefan Kuntz** leads the Software Architecture team in the business unit Powertrain Engine Systems of Continental Automotive GmbH in Regensburg, Germany. He is an active member of the AUTOSAR Timing Extensions Subgroup, also supports the definition of other AUTOSAR topics, like safety, multi-core, and system definition.
stefan.kuntz@continental-corporation.com

**Dr. Martin Kardos** joined Continental (former SiemensVDO) in 2006 in the position of a Software Platform Architect for Gasoline Systems. In years 2008-2016 he was responsible for the elaboration and rollout of methods for System Architecture Modeling, Variant management and Product Line Engineering within the business unit Engine Systems. Since 2017 he joined the SW Architecture group with the focus on modeling of Software Architectures with emphasis on description and analysis of dynamic aspects.
martin.kardos@continental-corporation.com

**Mr. Ralph Mader** studied Electrical Engineering at the University of Applied Sciences in Regensburg. He worked on the design and selection of micro controllers for the Powertrain area and the efficient use of micro controller resources. Since 2010 in addition he is leading development of the Multi Core software architecture for engine management systems. Mr. Mader represents Continental Automotive GmbH in several research projects in this area.
ralph.mader@continental-corporation.com