Safety Architectures on Multicore Processors – Mastering the Time Domain

Thomas Barth
Department of Electrical Engineering
Hochschule Darmstadt – University of Applied Sciences
Darmstadt, Germany
thomas.barth@h-da.de

Prof. Dr.-Ing. Peter Fromm
Department of Electrical Engineering
Hochschule Darmstadt – University of Applied Sciences
Darmstadt, Germany
peter.fromm@h-da.de

***Abstract*— A key architecture for building safe architectures is a strict separation of normal application code (also referred to as QM code) and safety function code, considering a separation not only in the memory and peripheral domain but also in the time domain. Whereas hardware features like memory- or bus-protection units allow a comparable simple protection of the memory domain, the supervision of the timing domain is a lot more complex. Race conditions on multicore system are far more likely and complex as compared to a single core system, as we have a true parallel execution of code and more asynchronous architectural patterns. Most safety standards such as IEC61508 [1] and ISO26262 [2] require:**

- **Alive monitoring**
- **Real-time monitoring**
- **Control flow monitoring**

In this paper we will describe a typical signal flow on a multicore safety system and based on this architecture introduce an innovative second-level monitoring layer, which is supervising the real-time constraints of the safety and functional monitoring functions. We will demonstrate the use of selected hardware features of the Infineon AURIX and TLF watchdog chip together with the SafetyOS PxROS from the company HighTec and show, how they can be used in the context of a safety architecture. Furthermore, we will demonstrate the use of a combined watchdog / smart power module, which does not only support an emergency switch-off but also the control of multiple power domains and defined reboot sequences in case of system errors.

Keywords—Timing, Control Flow, Functional Safety, Safety Architectures, Multicore, Runtime Environment

I. Safety Architecture

A very common design pattern for the implementation of a safety architecture is the use of redundant and independent channels. By monitoring and comparing the input, control and output values of both channels, single errors can be detected and the system can be switched into a safe state, as shown in Figure 1.

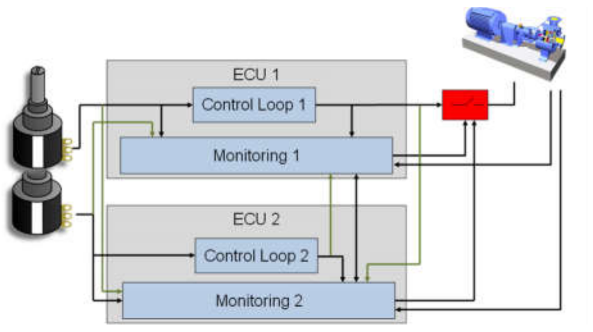


Figure 1 - Dual channel fail safe architecture

Transferring this architecture onto a multicore system by simply replacing the ECU's with a core will not lead to the same level of reliability, as the probability of common cause failures is higher compared to the discrete setup, which is due to shared resources, common power supply and similar [3]. Using a safe operating system providing separation techniques will help, but still the risks caused by a wrong configuration remains.

A possible approach to overcome these weaknesses is the introduction of a multi-layer monitoring architecture [4]. The first layer of monitoring functions will supervise the coherency of the sensor input data, the calculated control variables and the correct transfer to the actors, which still can and should be implemented in a multi-channel structure. As long as the monitoring functions work as intended, the system can be assumed to be in a safe operational state. [5]

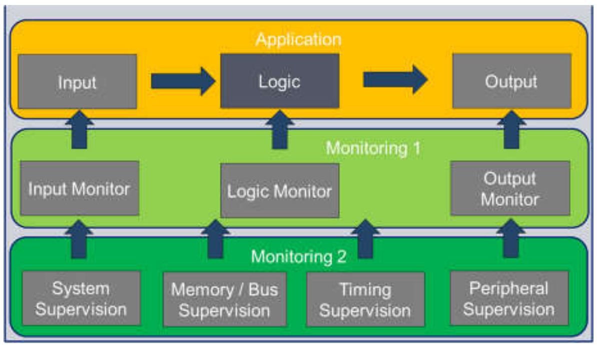


Figure 2 – Multi-layer monitoring architecture

However, what happens if a bug in one of the units shows an impact on the functionality of the monitors? In this case, the system might end up in a dangerous state, as the correct operation of the Input-Logic-Output channel is not supervised anymore. Therefore a second layer of monitoring functions is introduced, which monitors the health state of the two layers below. This health state needs to be actively and periodically reported to an external safety device like a watchdog chip, which in case of a failure will bring the system into a safe state [6].

The external device is required, because in case of a system error the main controller might not be able to reach the safe state by itself, e.g. due to an output task which is acting incorrectly or due to misconfigured or frozen safety ports.

The health-monitoring layer can be divided into four major blocks:

- *System supervision* covers hardware errors reported e.g. by a lockstep core, memory bit flips and similar.
- *Memory and bus supervision* focusses on the separation in the memory domain, by detecting illegal memory accesses reported by the memory protection unit or access violations of shared busses.
- *Timing supervision* ensures that critical software components are executed in predefined intervals. Furthermore, possible violations of the agreed real-time constraints are checked as well as the correct execution order of safety functions. This block will be the focus of this paper.
- Finally yet importantly, the *peripheral supervision* block ensures that all peripheral modules work as expected. Often, access violations can be detected and handled by the core's safety logic using the MPU and bus protection. In addition, the physical operation of the pin can be checked by reading it back or by using external supervision modules.

II. WHY TIMING SUPERVISION?

The supervision of the time domain is a typical requirement in most safety standards in order to detect system malfunctions and to take corrective actions before a system failure might harm humans or the environment.

Alive monitoring checks, if critical functions are executed at all. This is typically done by introducing a watchdog, which needs to be triggered in predefined intervals. In case the watchdog is not triggered, the system will respond with a hardware reset or similar action. This supervision is comparable easy in its implementation; however, the error handling scenarios are limited and usually quite harsh. This technique is often used in fail-safe systems. A failure of the alive monitoring check leads to a transition into a safe state, e.g. by using an external emergency shut-off unit.

Real-time monitoring measures the execution time of safety functions and checks, if the defined timing gates are met. This approach can be used to detect exceeded runtime of a function caused e.g. by a buggy algorithm, resulting in a late update of data required by a following process and subsequent system failure.

Control flow monitoring addresses the correct execution order of code. On the level of single functions this is ensured to a certain extend by using a qualified compiler. In the following code sequence, we can assume that the assignment in line 1 will be executed before the if-statement in line 2, which is followed by the function call in line 4, in case `funcA` returned the value 4.

```
1   a = funcA(); //a is global
2   if (4 == a)
3   {
4       funcB()
5   }
```

Figure 3 - Code snippet control flow

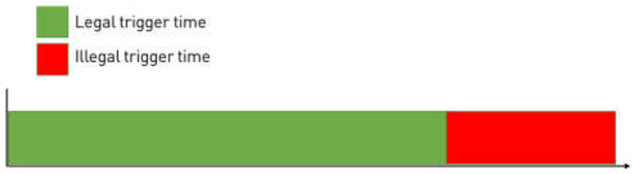
However, what happens, if an interrupt appears between line 1 and 2, modifying the value of the global variable `a`? Probably the behavior is not as expected. The same might happen, if we use a preemptive operating system. Here, a higher priority task may interrupt a lower priority task if it is activated. This might lead to a wrong behavior if not all critical sections are correctly identified and secured. This becomes even more an issue on multicore systems, where the cores execute code completely independent but share memory and other resources.

III. Alive Montoring

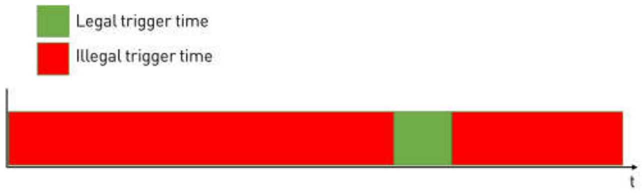
Alive monitoring is the most basic check, which detects if a system is alive or not. Being alive does not mean that a system is operational, it simply means that there is user-code executed and that the system is not locked inside an infinite loop, ISR, Trap or similar.

As alive monitoring aims to check if software is executed, the monitor itself cannot be implemented in software but hardware features have to be utilized, in order to ensure that errors can be detected even if no code is executed. A common hardware feature for alive monitoring are watchdog timers, which need to be triggered in predefined intervals. If a watchdog timer is not triggered as expected, it causes a hardware event, which can be used for error escalation.

Hardware vendors provide different watchdog timers. The most basic is a hardware counter, which automatically counts down a timeout value. If it reaches zero, a hardware event is triggered. Software has to ensure that the counter value is reset before the counter reaches zero; it becomes possible to detect whether software has reached the retriggering sequence within a certain interval. However, it is not possible to check if the watchdog has been triggered multiple times during an interval. Hence, it is only possible to check if user-code is executed within a maximum time but not if it is executed with the correct frequency.



A window watchdogs feature a time-window, in which it expects to be triggered. Only if it is triggered within the window, it is properly reset. If it is triggered outside of the window, it will report an error. Window watchdogs therefore not only allow checking if user-code is executed within a maximum time, but also introduce a minimum time. With window watchdogs, it becomes possible to monitor if software is executed within defined timing constraints.



Functional Watchdogs extend the trigger mechanism by introducing a protocol. Only if the protocol is adhered the watchdog is triggered, otherwise an error is reported. An example for a functional watchdog is the question and answer watchdog, where the watchdog provides a question, which has to be answered by software. In the most basic fashion, there is a limited set of questions and the answers are stored in a constant table. Functional watchdogs add a certain complexity and allow checking not only if the watchdog is triggered but also if basic mechanisms of the system are operational.

On bare metal systems with super loop architecture, the watchdog could be triggered in each iteration of the super loop. However, most systems run an operating system where the alive state is only given if certain tasks are executed periodically. In this case, every periodic task has to be monitored. This can be solved by defining a background or watchdog task, which triggers the watchdog only if all tasks report execution. With this approach, the task triggering the watchdog needs to have a higher cycle time and a lower priority than any of the monitored tasks.

As a major drawback, only cyclic tasks can be monitored using alive monitoring. Event driven tasks do not have a constant start time. More advanced concepts like deadline monitoring or control flow monitoring are required to secure such tasks.

While the alive state of a single core controller can be defined quite easily, the alive state of a multicore controller with multiple independent CPUs might be more complex. However, shared resources and similar can be used for inter-core communication. In this scenario, a watchdog-task on one core could gather information about all the tasks executed, even those on remote cores. Alive monitoring on multicore controllers is manageable but requires a well-designed overall architecture, which considers alive monitoring and inter-core communication.

IV. Realtime Monitoring

Real-time monitoring measures the execution time of a software function and compares it against a given design goal. The following picture shows the most important timing gates of a software function. The release time R of a function determines the earliest point of time a function can start. The start time S is the true time the function will start and the end time E is the true time the function does end. The deadline D is the latest time the function may end. The computation time C is the time, the function is active. As long as the execution of the function is not interrupted, C=E-S.

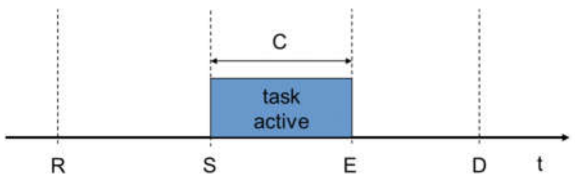


Figure 4 - Timing gates

As long as the conditions R < S and E < D are true for all functions, the system fulfills the real-time requirements.

A very simple and commonly used solution is to measure the idle time of a low priority background task. As long as the background task is executed, the system is not working to capacity - at least if all runnables which should have been called in this cycle have been activated.

To get a more detailed picture, we could also start a measurement at the entry point of the function and stop it at the end, which allows us to get values for the timing gates S and E. Safety operating systems like PXROS-HR [7] provide special services to abort a function if a timing gate is not set.

```
1  abortEventMask = PxExpectAbort(ev, func);
2  if (abortEventMask.events != 0)
3  {
4      //Do some error handling
5  }
```

Figure 5 - Realtime monitoring using abort functions

In line 1, the function `func` will be called and an event is provided, which will terminate the function `func` in case it is activated. A possible configuration would e.g. be to use a 1ms timing event. If the function requires more than 1ms runtime, it will be terminated and error handling can be initiated. Compared to traditional timing measurement, this approach has the advantage that the worst-case execution time of the function is known, allowing accurate timing violation detection. If a timing violation is caught on functional level, aggressive error escalation on system level can be avoided. Another advantage is that we can start the aborting event at the release time R and set it to the maximum cycle time (D-R) to protect all runnables, which will be called during this period.

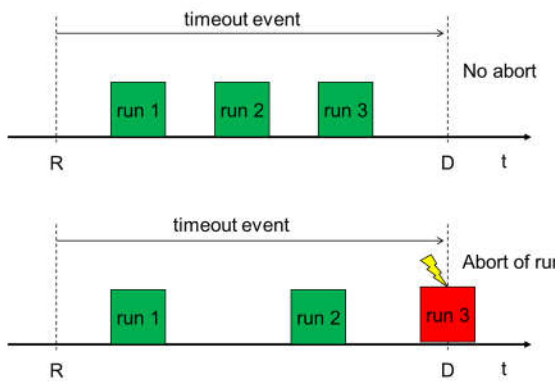


Figure 6 - Using abort event for deadline monitoring

A more data centric approach is to store the age of data together with the data payload. The age metadata is set to 0 whenever a new value is written to the data and incremented in cyclic intervals, e.g. every 1ms. Before using the data, the age and implicitly the call updating the data can be verified.

```
template <class T>
class data{
   uint21_t m_age;
   T m_data;
};
```

The disadvantage of this solution is the comparable high runtime overhead required for data update. This can be limited by increasing the update cycle time, but this decreases the precision of the information.

An alternative approach is to store the absolute time whenever the data is updated. When using the data, the current time needs to be subtracted to get the age. As this will typically only happen a few times during every cycle, the overhead is comparable low and a higher timer resolution can be applied. Furthermore, the absolute time can be used to analyze the control flow to a certain extend.

V. Control Flow Monitoring

In a multicore environment introducing asynchronous, non-blocking messaging mechanisms between the cores, deadline monitoring alone comes to a limit, as shown in the following example.

In order to avoid unintended data corruption on communication channels, only one task in the system is allowed to physically access the communication ports, e.g. CAN. All other tasks who want to use this port send a message containing the data to be transmitted to this service task. The service task will queue and transmit the data through the bus and will return an answer protocol to the requesting task using another message.

For data transmission, the requester task thus requires two runnables: One for sending the message and one, which is activated upon message-receive to process the return data. Let us assume the following valid sequence of operations:

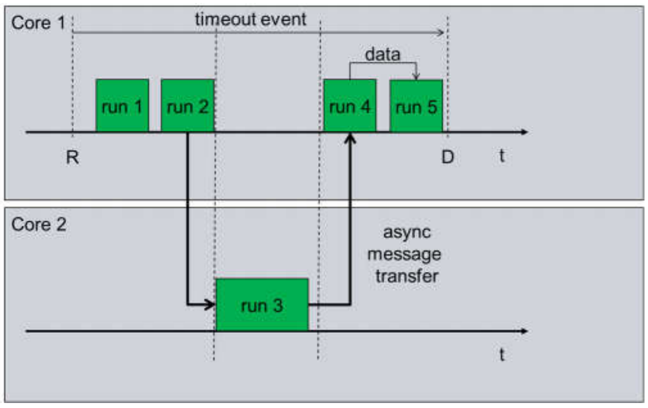


Figure 7 - Asynchronous communication

Runnable `run2()` transfers data to a service task of core 2 using an asynchronous message. The service is executed in runnable `run3()` and the return value is send back using another message, which will activate runnable `run4()` on core 1. The received data is stored in a shared memory and is used by runnable `run5()`. Runnable `run5()` assumes that the data has been updated in this cycle.

What happens if the service runnable `run3()` is delayed? If the delay is short, the timeout event will fire, a correct detection / handling of the error as shown in the picture below is possible.

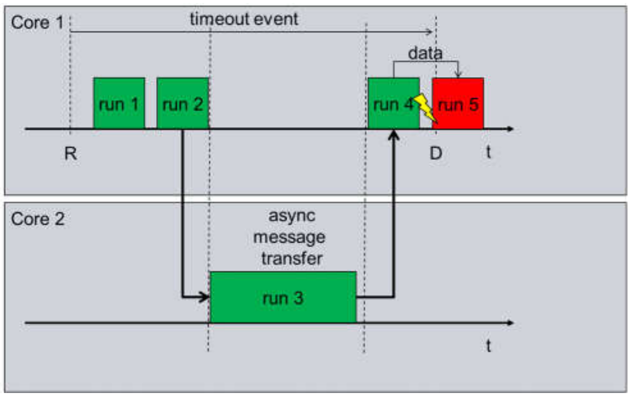


Figure 8 - Asynchronous messaging and deadline monitoring

If the delay increases, we might run in the following situation, where the update of the data in `run4()` happens in the next cycle. Core 2 might detect the deadline violation of `run3()` (at least if the service is executed in a blocking way) but core 1 might not be aware of what has happened, because the timing event supervising the execution of the runnables has been reset at the deadline D. As no code has been executed at that time, the behavior seems ok.

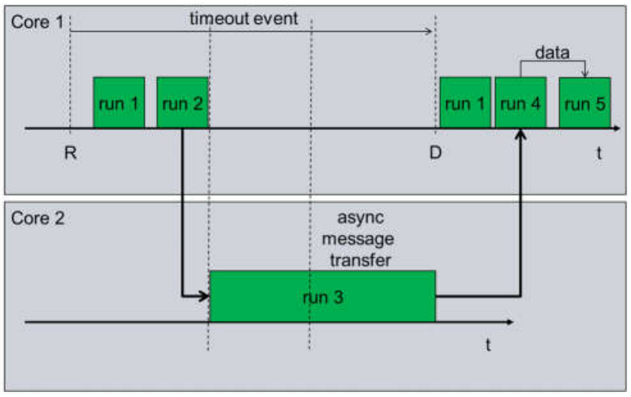


Figure 9 - Undetected error with asynchronous messaging

What can be done? One option would be to escalate a possible deadline violation of core 2 to core 1, but this will result in a complex error handling hierarchy.

The key problem is the following: Whereas the runnables `run1()` and `run2()` are called in a synchronous way, the event driven runnables `run4()` and `run5()` have a rather stochastic start time. Obviously, we have to verify that all runnables are executed within the expected order and timeline to ensure a proper operation of the system.

A comparable easy approach, which is a first step toward control flow monitoring, is to use the update time metadata concept introduced in the previous chapter. By comparing the update time of the message data in `run4()` with the request time of `run2()`, the significant delay would become visible.

Alive and deadline monitoring mainly focus on runtime constraints, whereas control flow monitoring checks if code is executed in a valid sequence. How can we describe a valid sequence? In the example above, obviously the flow 1-2-3-4-5 would be a valid sequence. This however only is true, if all runnables are executed in the same cycle, which is adding another condition. Furthermore, runnable `run1()` is independent from the other runnables and may be executed anytime i.e. also 2-1-3-4-5 would be a valid sequence. This trivial example shows that the description and validation of all rules for a real system will become very complex and time consuming.

A compromise could be to exclusively monitor critical sequences and conditions, which cannot be detected using the simpler and robust alive and real-time monitoring approaches.

An alternative implementation is to use token passing. Tokes can be sent from one runnable to another runnable and ensure the correct order of execution. However, also tokens have limitations if there are multiple valid sequences or if the data-path is reconfigured during runtime.

For a multicore system, a key requirement for control flow monitoring is a synchronous operation of the cores, which is typically hard to be achieved, as the cores might have different boot times. Using a common timer to store the update time of data signals is a possible solution to solve this problem.

VI. Error Escalation and Reaching the Safe State

Occasional violations in real-time or control flow monitoring might be handled locally without negative impact on the safety function. However, frequent timing violations as well as violations of the alive monitoring indicate severe malfunction of the system and need to be escalated

One approach to satisfy those needs is the introduction of a warning counter with threshold. A detected timing violation increases the warning counter, while correct timing decreases the counter. If a predefined threshold is reached, the timing violation is escalated.

The escalation of critical timing violations needs to be handled in hardware, as it cannot be guaranteed that software is executed properly anymore. Microcontrollers used for safety applications such as the Infineon AURIX feature a Safety Management Unit (SMU), which collects hardware error signals and defines the system reaction to an error. All watchdog error events cause so-called SMU alarms. The SMU reaction to an alarm is configurable; it is possible to send an interrupt/NMI request, to stop certain cores or to perform a reset. To implement a safety architecture, the SMU needs to be combined with an external watchdog such as the Infineon TLF35584, which is a multiple output power system supply for safety-relevant applications. In addition to power supply functionalities, it provides functional safety features like voltage monitoring, external watchdogs and error monitoring. A companion chip reduces the probability of common cause failures, as it is equipped with own vital components such as power supply or clock generation. By creating an own time domain on an external chip, the reliability of the watchdog concept is increased. Standards such as ISO26262 require the utilization of an external monitor in order to reach higher safety integrity levels.

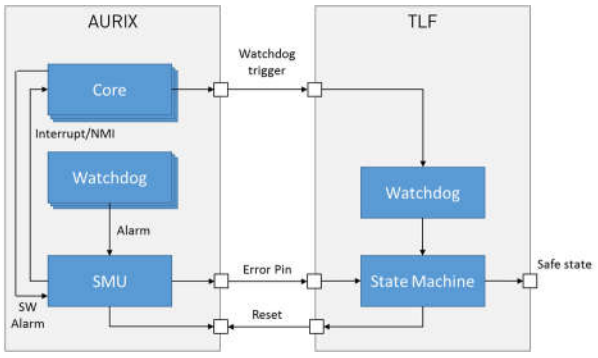


Figure 10 - AURIX Microcontroller with TLF companion chip

With the presented methods, it is possible to detect and escalate timing violations within the scope of the microcontroller. However, as the controller is usually part of a larger system, it has to be ensured that errors detected in the scope of the controller may not lead to unintended behavior of attached modules, such as actuators.

A possible solution is the implementation of a safe power supply unit. A prototype of such a system has been developed in the framework of the publically funded ZIM project “Future Technology Multicore” focusing on providing design patterns and solutions for safe multicore applications. The safe power supply ”SmartPower” is providing supply voltage and boot up, reboot and shutdown sequences for three safety domains: ECU, logic modules and actuators. ”SmartPower” is connected to the safe state pin of the TLF and turns off the system as a last line of defense in case of fatal errors.

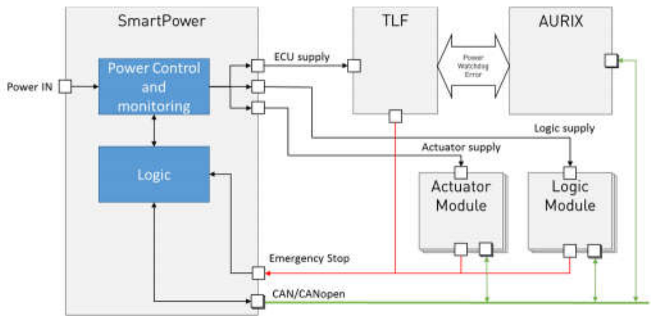


Figure 11 - Complete safety architecture

Furthermore, all power domains are permanently monitored for voltage and current overflows. In our architecture, the module also performs alive monitoring for all CANOpen nodes. The implementation of the logic is based on a Cypress PSoC, where the safety functions are realized in software and programmable hardware. This allows easy adaption of the system to different user requirements.

VII. References

[1] IEC, Teil 3: Anforderungen an die Software/ Funktionale Sicherheit elektrischer/elektronischer/programmierbarer elektronischer Systeme, 61508, VDE, 2001 (3 July 2001).

[2] ISO, Part 6: Product development: software level / Road vehicles - Functional Safety, 26262, Genf, 2011 (2011).

[3] J. Barth, et al., 10 Schritte zum Performance Level, Bosch Rexroth Group, 2014.

[4] Thomas Barth, Peter Fromm, A Monitoring Based Safety Architecture for Multicore Microcontrollers, Nürnberg, 2017.

[5] Thomas Barth, Peter Fromm, Functional Safety on Multicore Microcontrollers for Industrial Applications, Nürnberg, 2016.

[6] Prof. Dr.-Ing. Peter Fromm, Thomas Barth, Mario Cupelli, Sicherheit auf allen Kernen - Entwicklung einer Safety Architektur auf dem AURIX TC27x, Sindelfingen, 2015.

[7] HighTec EDV Systeme GmbH, Tricore Development Platform User Guide v4.6.5.0, Saarbrücken, 2015.