Specifics and Vulnerabilities of the TimingControl in Cyber-Physical Systems

Iliya Georgiev, Ivo Georgiev

***Abstract* — Cyber-physical systems integrate powerful computing (real-time embedded system, operating system, applications, and Internet networking) and physical environment (advanced manufacturing cells, medical platforms, energetics aggregates, social and educational control). The reliable functionality depends extremely on the correct timing. Wrong timing because of buried malfunction or external tampering could be critical. The paper is some analysis of the vulnerable timing parameters that influence the precise processing. Expert estimation of the criticality of different timing parameters is given to support fault-tolerant design considering possible failures.**

***Index Terms*—cyber-physical systems, real-time, timing control, vulnerabilities, Internet of Things**

I. Introduction

Accurate timing is one of the important requirements in cyber-physical systems, especially for real-time modes. Such timing constrains vary from soft real-time (functional deadlines are flexible and, in most cases, not fatal) to hard real-time (exact deadlines must be successful). Embedded systems control timed information flow to/from several sensors and actuators. All configuration works in local area network (LAN) and is open to the world by Internet.

Timing control and possible failures are the motivation of the present analysis. The most sensitive timing parameters are discussed with the possible critical results. Timing control functions are distributed in almost all components of the system and the vulnerabilities in significant degree depend on the real-time modes stability and reliability. Corrupting timing values can result in functional compromising and even in catastrophic collapse of the environment and injuries of the service personnel.

Timing parameters (execution time, deadlines, periods, clocks, scheduling quantum, timeout time, etc.) are just variables distributed in different places of the programs. There is no universal reliable way to protect such variables. For example, cryptography does not help because changing the encrypted value will generate the same functional corruption.

In the literature there are a lot of publications that consider security of the cyber-physical systems. For the interested readers we would recommend some review publications [2, 3, 4]. Our research in not an additional work in that area but addresses timing failures that could generate degrading delay and corruption of the whole system functionality. The paper is partly an extension of a conference paper [1].

Iliya Georgiev is with IEEE, Denver section, Colorado, USA (e-mail: ilgeorg@ieee.org).

Ivo Georgiev is with Metro State University of Denver, P.O. Box 173362, Colorado, USA, (e-mail: igeorgi1@msudenver.edu).

The authors declare no conflict of interest.

The manuscript is organized in nine sections. Section II presents some architectural features of ARM architecture that support real-time functionality. Next sections explain the addressed timing in real-time processing. Timing control vulnerabilities are discussed in section III. Multithreading specifics are in section IV. Section V emphasizes on network and Internet of Thing (IoT) common timing. Section VI characterizes the real-time operating system (RTOS). Real- time high-level programming considering trusted Java and C language libraries are in section VII. Section VIII provides brief discussion of some development approaches and expert view about the timing parameter's criticality. Last section is the conclusion.

II. Architectural Support

Because of the internally driven timing, understanding the architecture of the embedded systems is important to create a real-time functionality. Cyber-physical systems receive and generate analog and digital information from/to the environment. On the other side they follow complicated networking protocols and perform Internet messaging and Web services. Such functional diversity needs precise timing on all levels. Computational power, timing and security are the main design challenges that can be achieved by contemporary instruction level architecture. ARM architecture [5] is prevailing for most microcontrollers that are under production with a diversity of peripheral devices oriented to the environment needs. ARM family of compatible microcontrollers has a common programmer's model: 32 or 64-bit ALU, registers, status words, common memory map (read-only memory for programs and constants, main memory, single cycle read/write memory for the peripheral devices).

For the interested readers we recommend the book of professor J. Valvano [6], which gives excellent view of the microcontroller's structure and C and assembly languages programming. The ARM evolution provides additional functionality: new instructions and open-source libraries for secured and timed signal processing and networking [7].

ARM instruction level architecture adds important features:

a. Flexible addressing modes for digital signal processing that accelerate the conversion of the analog signal from time domain to frequency domain by Fourier transformation. Original Fourier transformation works in the space of the complex numbers. Other Fourier-derivative transformations work in the spaces of real or integer numbers.

b. Arithmetic operations that perform saturation of the operands. Analog signals are digitalized by sampling

https://doi.org/10.47978/TUS.2021.71.01.004

(amplitude values are taken for every time interval) and quantization (the amplitude values are presented as binary digits between lowest and highest value). Digitalization is characterized by a range (the distance between the lowest value and the highest value), precision (the number of bits to present the amplitude value) and resolution (the smallest value between two digits). The real measured amplitude could be greater than the highest value or less than the lowest value. In such cases saturation process is needed – for every amplitude outside the range the highest or correspondingly the lowest digital value is taken (the signal is saturated). Saturating instructions give significant gain in performance.

c. Atomic combined instructions that are register-memory arithmetic/logical instructions (remember historical CISC architecture!). They are load/store combined with addition, subtraction, exclusive-OR, AND instructions, i.e. two execute phases in one.

d. Synchronization atomic instructions to increase the fault-tolerance access to shared resources. There are two types of instructions: a. instructions for atomic access to the synchronization primitives (locks); b. instructions for synchronized processing of some data set by different independent threads that can run in the same processor or in different cores.

e. Cryptographic instructions for encryption/decryption procedures based both on symmetric cryptography and cryptographic hashing.

f. Memory protection instructions that define different non-overlapping regions of the memory space assigning some accessibility and permission flags. The regions are two types: secure or non-secure. Memory protection registers define the type of the regions, assigned control functions, base and limit addresses. Multiple regions can share the same attributes. The memory could be a normal memory (general-purpose instructions use it) and device memory (input/output direct memory access operations use it). Every normal region can be assigned different attributes: cacheable or non-cacheable (write-through or write-back policy), sharable or non- sharable, executable or never-executable. Device memory attributes control the input/output stream: gathering or non- gathering merging in common transaction; reordering or non- reordering, early or non-early buffering.

g. Exception model that supports different types of unusual (or faulty) situation in the processor. Exception processing makes the running thread to stop, and a hardware supported handler occupies the processor. Reset exception is caused by power-down. Hard-fault exception has highest priority and cannot be masked. Supervisor call exception is activated by a special instruction that activates the OS kernel or some supervisor. System timer SysTick exception is generated by the timer itself or by the software. Interrupt exception is a signal from a peripheral device or generated by a thread or handler [8].

The whole system functionality is interrupt driven. Every interrupt type is supported by an interrupt handler whose starting address (vector) is provided in a vector table. The interrupts have priorities, and all of them can be masked but the failure ones. Timing precision in the intensive interrupt processing is the dominating requirement to guarantee the worst-case execution time (WCET).

ARM real-time microcontrollers are integrated with a suit of peripheral devices that cover almost all input/output interfaces and local area networking. The bus hierarchy splits the fetch/execute processor cycle from the read/write input/output stream, which is synchronized by SysTick. Integrated peripheral devices consist of synchronized or not synchronized interfaces (could be also serial or parallel); others perform analog-to-digital/digital-to-analog conversion and networking. The streams are controlled by separate timers and are connected to the sensors and actuators via programmable ports that can be switched between digital or analog signals.

Networking of cyber-physical systems depends on timing. The local-area networks (LAN) are time-sensitive. Connection to the Internet and Web support is called Internet of Things (IoT) and must follow common timing.

III. Vulnerabilities of the Timing Control

Timing control in cyber-physical systems needs to be adjusted to the real-time requirements. All subsystems have separate timing controls that are prone to malicious modification of the internal settings like tweaking the time to corrupt the system. Detailed analysis is needed to implement common timing.

The timers that control the operations to the sensors and actuators are synchronized by the main clock. Clock generators are programmable devices that can change the parameters of the clock sequence and this is one of the most breakable processes.

Main clock frequency can be changed by the programmer by simple change of the parameters in the control registers of the clock generator to increase/decrease the execution speed. Increasing the clock frequency helps to meet tight timing bounds, but the power consumption is high (so is the emitted heat) and the microcontroller becomes sensitive to interference and internal signal races. Slower execution gives better power efficiency and increases the reliability of the microelectronics. Secure clock frequency is directly connected to the timing control of the real-time IoT systems.

The next formula is from [6] to illustrate the relationships in the software work. Equation (1) shows with some simplification that the power is linearly proportional to the main clock frequency.

$$Power = k * F_{main\ clock}, \tag{1}$$

The coefficient k generalizes some technology specifics. With the assumption that one instruction is synchronized with two clocks, the software work dependence on the speed/power ratio is presented in (2).

$$\begin{aligned} Programs\ \text{Workload} &= Number\ of\ executed\ instructions * \\ &* \tfrac{1}{2} F_{main\ clock} = \\ &= Number\ of\ executed\ instructions * \tfrac{1}{2}\ Power/k \end{aligned} \tag{2}$$

Main clock generator has different signal sources: highest-stable piezo or thermostable multi-oscillator.

Several registers provide bit-by-bit control to switch on/off the power and to select the frequency divider or multiplier selecting up to thirty-two frequencies.

Timers (8 to 32 in a system) are devices to synchronize almost all operations of the sensors and actuators*:* periodic interrupt requests, analog or digital signal sequence from the sensors, serial or parallel outputs.

Timers are implemented by simple procedures based on count-down counters that are decremented by the clock or multiples of the clock. Enable/disable flag activates the timer, after that the initial value of the counter is loaded. At the beginning the timer is disabled to store the initial value. After initialization, the timer is enabled. Counting down has several modes. When the counter reaches number zero, it is initialized again. Such procedure is overly sensitive because wrong initial value or wrong mode can destroy the whole input/output operation and further the thread or the current handler.

IV. Timing in Multithreading

Standardized view of computing consists of two sets: a set of processes (plus threads and handlers); a set of resources (CPU time, memory, file structure, interrupt parameters, access rights, semaphores, etc.) that are manipulated in timed multithreaded scenario. Multithreading dynamics in real-time systems depends strongly on exact timing parameters that are risky for the whole functionality. Main sensitive parameters are execution time, possible deadline, and periods. Every thread receives some CPU time (time quantum or slice) to run based on the priority and aging policy. Some of the threads are periodical, they work after some periodical signal from a timer of from a environment.

Execution time can be explicitly declared as a parameter, but most developers use it implicitly to keep it flexible. Some applications (so the RTOS) make occasional check of the running execution time and compare it with a stored parameter.

Thread's deadlines are analyzed during every time quantum in which the thread runs. The deadlines could be changed during the scheduling or interrupt processing. Deadlines are checked on different stages. RTOS makes estimation during every scheduling session whether the deadline could be met (only for hard real-time threads). The Run-Time-Environment (RTE) manipulates the deadlines of the user threads. Thread executable code uses prediction algorithms for possible deadline violation.

Periods of the periodical threads are stored parameters. The timer attached to check the periods must be adjusted with the astronomical time. Estimated execution time, period, and deadline times can be organized in groups. They may be fixed or flexible and must be protected by RTOS, RTE or by the thread or the handler. Multi-core implementation needs deeper protection of that group [9].

Timing control becomes difficult when threads and even some block statements are synchronized to access shared data. Isolation (mutual exclusion) of the shared data (resources) in all cases is based on simple variables – locks (different lock names are used in programming development environments - semaphore, mutex). Locks keep binary state (free/busy) and can protect some resource (data locks) or can block cooperative execution of some protected method, which is called monitor. Synchronization methods (functions) are offered in almost all application programming interfaces under different names, but their internal implementation consists of two base functionalities.

The function *wait()* exchanges a register that keeps the busy state with a lock value in the memory; if the loaded lock is available (free), the critical section can be executed; in case of busy state, they are two scenarios: a. busy waiting – the lock is spinning and the thread loops; b. non-busy waiting – the thread terminates and goes to a blocked state. Spinning locks are mostly used in short critical sections in the kernel of the operating systems. Non-busy waiting scenario is accepted for most user threads.

The function *signal()* – sometimes it is called notify() - releases the lock again by exchange a register value (now free) with the lock variable that was busy during the execution of the critical section.

Lock variables are retrieved and checked very often, and they must be stored in the main memory preferably in some protected sectors.

Each synchronizing function must be atomic, but the register-memory swap occupies two or three instructions. Interrupt mechanism allows interrupt after every instruction, which needs special instructions for such exchange. Some microcontrollers include atomic swap instruction (for example SWP in ARM) or load/store using hardware protected memory sections. Synchronization methods *wait()* and *signal()* should be used in pairs and not swapped. Recommended approach is not to cache them in the instruction cache.

In digital signal processing and other important functions, the running model is the single program-multiple data, where several independent threads make the same processing over different pieces of big data. The threads can terminate their current work at different moments, but they must wait for the others. Here some *barrier instruction(s)* synchronize mutual processing. Barrier could be some event like end of memory operations in a thread or some count (so called barrier counter) that is loaded with the number of the threads. Every execution of the instruction decrements the counter and if the counter is not zero, the thread waits for a zero counter. If the architecture supports protected memory sectors, more advanced barrier instructions are efficient with different signals to wait - end of loads or stores, completion of memory accesses.

Timing sensitivity increases in distributed cyber-physical configuration, where sensors and actuators are grouped (even swamped) in separate LAN. The timing parameters should be controlled periodically following common astronomical time. Writing precise critical sections is a challenge because the proved algorithms must be mixed with some time control. The designers usually try to follow well proved standard synchronization problems (Producer-Consumer, Dining-

Philosophers, Readers-Writer, Cigarette-Smoker) and considering time makes those problems more complicated.

Timing together with deadlock-free programming is another development problem. Deadlock is when some thread receives a resource and needs another that is used by other thread, which on its side needs the resource of the first thread. Conditions are no preemption and only the thread can release the occupied resource. Deadlock-free programming cannot be supported by RTOS, because prevention

and avoidance need search on huge allocation graphs during every scheduling cycle.

The RTOS and the applications can try to detect the deadlocks and to save the system functionality ([10] shows some method). The implementation drawbacks are no timing control consideration, difficult to debug, slowing the execution time, risky.

Serialization of the thread's execution is the most effective way to write deadlock-free real-time programs if the WCET allows it. A thread is not supposed to request a resource that is currently used by some other thread. In this case the thread must release all held resources and request after some delay everything needed.

V. Networking and Internet of Things

Timing control puts additional trends in the diversity of networking hardware, protocols, and software layers. Let us consider a manufacturing cell that has a group of robots, machinery, and transport devices. The robots usually are connected in a synchronized network with a restricted length, most popular is the CAN technology [11]. The robot network and the other devices are connected in lengthy Ethernet network to the microcontroller of the embedded IoT system. The IoT-based system is also an Internet host and performs all Internet layers and protocols. It supports application specific protocols for remote procedure call and message exchange. Timing design must follow well-proved sequence from the manufacturing cell environment through the control processing and further by Web services to the cloud. The timing values of all stages must be calculated, and some maximum latencies to be practically proven. On all nodes some bandwidth control must upgrade the conventional networking. The LANs are now time-sensitive, and the Ethernet layer is time-reserving [12].

Timing control synchronizes the timing behavior in real-time communication by standardized declaring of bandwidth and time intervals.

Internet protocols provide authentication and key management (by public key encryption), confidentiality (by secret key cryptography), message integrity (by cryptography hashing) services. IoT requires timing that the standard Internet layers do not provide. Real-time version of the Internet Protocol (IP) layer tries to meet such timing setting some parameters in the IP headers that give priority to the packets. The given priority and improved congestion control are to provide quality of service, which guarantees "best effort" (the question is how "best"). For tight WCET the best effort policy is not enough, and predictable buffering can help the time estimation.

In cloud-based cyber-physical systems, the sensors and actuators are remote to the microcontroller and are structured in clusters with autonomic control. Time control is organized separately in the clusters and the main system and is driven by the common astronomical time.

VI. RTOS Specifics

RTOS have similar functions as conventional operating systems: multiprocessing and multithreading, memory management, scheduling, synchronization. Additionally, it can support hard and soft real-time multithreading that follows additional requirements [13, 14]: a. process request for service of the external events at strictly defined timing latency; b. predictable time to respond to the interrupts; c. reliable continuous processing after some failure; d. user reconfigurable configuration based on the application specifics and the environment; e. support of real-time constraints that are dictated by the implemented control functions or external devices; f. dynamical change of the priorities according to the deadlines.

On the other side, control functions of RTOS have been reduced to ensure that the critical application runs in predictable computing container. The correct error-free functionality of the cyber-physical system is full responsebility of the designer. RTOS kernel supports only the memory management, multithreaded scheduling and partly synchronization.

Input/output streams are driven by the applications but not by RTOS, which usually manipulates only the system timer and the hardware failures handlers. Timing control is distributed in the application's stack that is important to track the execution time and the deadlines. Obviously, the development does not rely on the protection of the operating system and this could be one of the weakest parts of the design.

The application does not need to call the operating system to process the exceptions and interrupts. Change of the priorities and masks is by privileged operations that can be done by the RTOS or special interrupt handler supervisor. Switch context to interrupt handlers is by vector address and is mixed hardware-software procedure.

Protection of thread's interference is only partly supported by the RTOS. The application threads can execute all instructions including privileged ones. Threads can directly access protected or non-protected memory areas according to the desired configuration. Memory protection is minimal.

Thread scheduling is a RTOS procedure, and it is based on priorities. The scheduling algorithm can be configured according to the specifics of the system. Soft real-time threads are selected for execution by classical round robin priority scheduling with aging of the priorities. Hard real-time scheduling algorithm is the rate-monotoning for periodical threads and is based on the short-first principle (the priority becomes higher for shorter periods). Earliest deadline first scheduling recalculates the deadlines during every scheduling session and runs the thread with the nearest deadline. The latter has unpredictable high overhead and can influence the timing control.

Timing controls in RTOS are time quantum, timeout values and astronomical time.

Time quantum for execution is calculated according to the priorities and average CPU burst that can be given or dynamically predicted. During the scheduling session the quantum and the priorities are dynamically changed following the accepted aging scenario to keep a relative fairness.

Timeout values are risky selection and sometimes underestimated during the global timing control. Information exchanged with the peripheral devices can be time-outed if the operation is suspiciously long. Timeout periods are selected after long practical estimation in real load of the environment. Wrong timeout values can dangerously violate the timing of the system.

Astronomical time must be followed by all subsystems. The RTOS, the threads and all networking nodes must regulate their procedures according to the international

atomic time. The cyber-physical system especially connected to the cloud must follow a protocol to synchronize the astronomical time. The protocol collects information from the neighbors' stations and sets a correction value. Fault-tolerant sessions should prevent the wrong value distributing. RTOS do not provide reliable functions to track the execution time. The designers can specify application-level tracing of the execution time at some degree – some examples are given in [15]. But such approaches are not efficient and difficult to implement and debug.

VII. Specifics in Real-Time High-Level Programming

The functionality of cyber-physical systems is incorporated in a complex software that is organized in different vertical and horizontal levels. The complete software stack consists of general-purpose applications for IoT, specific applications driving the whole cyber-physical functionality, run-time environments (RTE) of the high-level language used, RTOS with a kernel and supporting system programs, libraries, drivers, and interrupt handlers to control the peripheral devices.

Application suite is written in different programming languages that can support on one side Web technologies and, on the other side, can control variety of sensors and actuators in dynamically changing environment.

Hierarchical implementation for example uses Java on the highest IoT level, C language and assembly language for the lowest level (hardware-dependent libraries, drivers, and interrupt handlers). Most of the immensely deployed open-source Android-based IoT-based systems provide such development hierarchy: Java libraries of classes, Java RTE, C language libraries, Linux kernel and drivers, assembly-language libraries, interrupt handlers (general purpose and hardware specific).

Considering correct timing behavior of the system, the designers are supposed to understand very deeply how the high-level programs are translated and executed and to try to isolate the timing control from the general processing. Isolation is a traditional approach to increase the computing security. Classical isolated technologies are memory protection (hardware and software implemented), access control (OS controls the resources access from different processes), firewalls (networking layers level and application driven).

For real-time trusted programming we can briefly explain efficient combination of low-level secure libraries (usually in C language) and Java language. Such technologies gain significant support in combination with the soft real-time operating system Android, which is considered the "operating system of IoT".

Secure library technologies are tightly connected to the processor functions. For ARM microcontrollers radical isolation offers the popular ThrustZone [16, 17], which is efficient because of the architectural support. The main computing components (hardware, data, software) are virtualized into two development containers: trusted and non- trusted. Trusted as well as non-trusted computing (also called secure and non-secure worlds) have separate hardware, data, and software; both can run on the same processor in different time quanta. Trusted resources are protected, and non-trusted software cannot use them but must call for a service from a secure (trusted) monitor. Trusted software could be controlled by a supervising library or even tiny OS module with a separate kernel. Non-trusted container may consist of conventional OS (for example Android) and applications.

For both worlds ARM hardware maintains several states that define what is the processor mode (user, supervisor, system, interrupt). A bit in the secure configuration register declares the world (secure and non-secure). The state gives levels of privileged access to the resources. The privileges levels are four. The highest level is only for the trusted monitor. The other levels are separated for both worlds: lowest level for the applications, next levels for the separate operating system (or supervisor in the trusted world). Memory space is divided between trusted and non-trusted computing. The trusted memory regions cannot be accessed by the non-trusted programs. The secure monitor is the key firmware (TrustZone library) in the trusted functionality. The ARM architecture has a separate instruction to call the monitor – Secure Monitor Call (SMC). Executing SMC activates switch context, which is hardware supported to save the registers and the programming counter. Secure monitor performs important functions: a. supervised power management; b. secure bootstrap of the kernels of the operating systems; c. SMC handling that fully separates both worlds; d. common management of the system; e. control of some exceptions. Both worlds exchange shared data (in registers or in the memory) only under control of the secure monitor.

Java secure programming needs deep understanding of the multithreading organization. Java application, which is invoked for execution, is converted into a process with one basic thread. The process could be considered as a container of the execution code and the process resources. The basic thread runs the execution code and shares all the resources. Additionally, it has its own execution resources (programming counter, stack, program status words or registers). The basic thread can generate new user threads (subclasses of the *Thread* class). Every user thread has separate programming counter, stack and state information and shares all the resources of the process. Designers can declare ten priorities of the user threads but the RTE does not guarantee its exact execution.

The designers must understand the duality of the scheduling mechanism. On one side the Java RTE follows a state diagram for scheduling only of the user threads independently from the OS. The Java state diagram supports the following states: a. *New* - just created user thread; *Ready* - the thread is activated by a *start()* method and is in a pool or a queue of all ready user threads; *Runnable* - the thread is just activated by a *run()* method or returns from a blocked state after successful I/O operation or releasing a lock; *Blocked* - the threads is blocked: either by *wait()* or *sleep()* methods, or by activated I/O operation, or it is time-outed; *Dead* - *run()* method terminates.

On the other side, operating system follows different scheduling state diagram for all active threads. Java *Runnable* user thread can receive the *Running* state from the OS and will start running in the processor.

Standard Java classes have some obstacles to support real-time programming. The designer must consider that very often a Java runnable thread could be blocked unpredictable way by the operating system, and this can influence the overall timing. Additional indeterminism

generates the garbage collector, which is a user thread with a highest priority and preempts the running thread. Standard Java implementation also has a limited view to the memory – the class variables could only use references to the memory. And finally, thread interaction is limited.

Java real time specification extends the standard definition and supports deterministic execution and access to the memory and the hardware configuration.

Memory management declares new memory areas that can be accessed by objects declared in those areas: *ImmortalMemory* (objects live until the end of the application) and *ScopedMemory* (objects are dereferenced only when they exit the area). Practical recommendation is to define the timing parameters in such memories: constant timing parameters in immortal memory and changeable timing variables in scoped memory.

Real-time thread management is based on new classes and includes preemption with 28 new priorities with respect to the deterministic garbage collector. Real-time threads could be periodic or aperiodic with corresponding release parameters. The user threads are now three types:

a. Regular Java threads that are subclasses of the *Thread* class.

b. *RealtimeThread*s that keep some deterministic latency, can cooperate and interrupt the garbage collector.

c. *NoHeapRealTime* threads cannot access the heap and are mostly not preemptive by the garbage collector. Such threads can be successfully used to manage the sensitive timing parameters especially for network time quantum.

The real-time threads must be short and could be combined only with threads that follow high-resolution time. It is not recommended to create objects in those threads that do not manipulate timing behavior.

The scheduling of the user threads strictly enforces true and fixed priorities. The threads are scheduled in a predictable way for the same conditions. Priority inheritance prevents priority inversion and ensures that the higher priority thread will be executed without latency.

Interrupt handlers are converted into preemptive real-time threads that can participate in the priority hierarchy like any other thread. The interrupt latency is predictable low.

Interthread communication is one of the most challenging topics in real-time programming especially in real-time Java where the handlers are threads. The cooperative mechanism is based on a Boolean flag in every thread that can be set from another thread by an *interrupt()* method, which is only an invitation for interruption. A thread, which issues the interrupt, tries to communicate to the interrupted thread by setting the flag. If the interrupted thread executes some blocking method, it can terminate and go into a blocking state. In case of non-blocking method, the interrupt request can be analyzed, and this is a signal for additional processing. Interruption supports cooperative actions to reorganize any processing in progress, recovers some data and initiates other, notifies other activities and afterword terminates.

Communication is concentrated in *InteruptedException* which must be thrown and analyzed very carefully [18]. Different ways are recommended to process such exception in an invocated hierarchy of methods. Ignoring the interrupt request is strictly not recommended in real-time programming. If the invoking hierarchy contains blocking methods, then the most useful technique is to propagate the exception to the invoking blocking method without catching it. In case of catch clause, some cleaning or thread-specific work can be done before rethrowing the exception. Performing some calculations before rethrowing is the most reliable approach. After needed processing the current method can restore the just cleared flag and invoke *interrupt()* again to inform the method higher in the call stack that some interrupt took place. Recommendation for methods that do not throw the exception is to re-interrupt the current thread, which is some initialization of the flag and is a popular technique in interthread communication.

Real-time systems run in a sophisticated environment with a significant number of signals that are not deterministic in time and frequency. Java provides asynchronous event handling (*AsyncEvent* and *AsyncEventHandler*) and transfer of control by *AsynchronouslyInterruptedExceptions. AsyncEventHandlers* are special threads that can be bounded with different events (several *AsyncEvents*) and can be scheduled and executed asynchronously [19].

Signals from sensors/actuators need immediate action to switch the running execution to an appropriate service. Such transfer of control is done by processing of the *AsynchronouslyInterruptedExceptions*, which can be thrown explicitly by firing or by interrupting the current thread in some pending techniques. The asynchronous interrupt becomes pending in case when the current thread executes a deferred section. Propagating a pending asynchronous interrupt happens later during the next invocation of the asynchronously interruptible method. An asynchronous exception can override all other exceptions at the same time. Several asynchronous exceptions can be generated during the execution of the first of them.

Cooperative interruption mechanism can provide flexible interthread communication for control transfer and necessary cancelation of the current processing. Real interruption takes place not immediately but after the decision to cancel or to continue processing in the scope of the current real-time thread. The interruption can be deferred to perform specific action without violating the functional integrity. In case of not necessary interruption the status of the interruption flag must be restored and this way the calling method cannot lack the knowledge of the interruption. Some popular techniques are also to restore some timing parameters that influence the current thread.

Asynchronous transfer of control based on exception looks like a strange technology, but it is powerful in case of careful programming. Deferring, pending, cancelation of the asynchronous exceptions can generate some indeterminism in the timing behavior of the real-time processes. Interrupt handlers are threads that can be preempted, which makes the transfer of control using exceptions incredibly challenging with a floating latency, and not deadlock-free.

Additional specific is the attempt to interrupt a thread that runs in a monitor (synchronized method or block statement). The exceptions and the restoring of the timing parameters are postponed.

VIII. DISCUSSION

Recent attempt of analysis is based on several examples of system corruption because of some weak timing control. The accumulated practical experience of a team of developers is the motivation to share partly some techniques to achieve smoot timing.

Designers must put the critical timing parameters in mostly protected address spaces. We already mentioned that Java real-time programming provides two protected memories but manipulating the data in those memories needs more professional skills. Other development stacks provide similar techniques. IoT needs web programming, in most cases Web services are based on XML vocabularies, which makes the needed exchange of timing information more vulnerable.

In low level programming (C language and assembly language) threads and handlers store the timing parameters in memory protected sections. Handlers of input/output interrupts must avoid manipulating the timing variables in the memory area for the peripheral devices because data from the environment could be manipulated. In C99 language there is a *volatile* keyword, which informs the compiler and RTE, that this variable is exchangeable between different parts of the whole functionality:

a. from/to ports where the in/out values are independent from the software control;
b. as global variables used to cooperate information between the interrupt handlers and the main thread.

Timing control of the most peripheral devices is organized by separate control registers. Dynamical change of the timing bounds is performed by logical instruction (AND, OR, XOR) using some masks. Let us present some example of changing the main clock frequency (all timers depend on main frequency). The following simple code shows how to manipulate the main frequency (we recommend the presentation in [5]). The SYSCLOCK is a control register that has bits to manipulate the main clock frequency. The next C- language statement sets the clock generator source:

```
SYSCLOCK = SYSCLOCK & ~0XFFFFFF0F;
                    // select the oscillator
```

In ARM assembly language the instructions are the following:

```
LDR R1, = SYSCLOCK        ; load the address
LDR R0, [R1]              ; store SYSCLOCK
AND R0, R0, #0x000000F0   ; change the field
```

The AND instruction uses immediate operand stored in the instruction. Loading the instruction in the instruction cache is vulnerable - the immediate operand could be corrupted by some attack. It is recommended not to use immediate operands and to store such constants in a memory protected area or in ROM.

Widely accepted prevention of timing parameters is not to use recursion. Recursion pushes partial calculated values in the stack, which needs uploading of the stacked data in the data cache. Switch context after preemptive interrupt can leave the timing values in the cache open to some malicious change. Here the designer must wind down the cache to the memory or to use non-cacheable memory for timing control. RTOS kernel provides some memory consistency protection that issues the requirement to execute the timing control in the threads that run in the core. Drastic approach in avionics system is to run the time manipulating threads only in one core, the other cores run utility code.

Micro-rebooting (after fault or periodically) to recover the stable state is another technique, when the system runs in noisy environment.

We share in Table 1 our view how critical is the violation of the timing control by corrupting different parameters.

TABLE 1
CRITICALITY OF THE TIMING PARAMETERS

Timing parameter	**Critical** *(0 – not critical,* *10 – catastrophically critical)*
Main clock frequency	8 to 10 (catastrophic failure).
Time parameter in the System Timer, that synchronizes the information exchange with peripheral devices	6 (control operations could be destroyed). 9 (progressive degradation of the whole functionality).
Timer's control parameters	7 (input/output failures), 5 (partial degradation).
Time quantum	6 (WCET influence, violated fairness, starvation, and deadlock possible).
Thread deadline, execution time, period of the periodical threads	6– 8 (for the current thread). 3 to 5 (for the whole functionality). 6 (WCET not met of some current thread).
Timeout parameters	6-8 (chaotic execution of some operations, degradation).
Change of the user thread priority	4 (partial or global change of the timing).
Bugs that prevent immediate action after sensor/actuators signal	4 (partial disfunction of the current thread, delay that can violate the WCET).
Lock's swap	5 (shared data not available, functional degradation of the current thread).
Barrier counters (barrier events)	4-6 (wrong execution, data corruption).
wait() and *signal()* functions	7 (shared data destroy, possible deadlock).
Deadlock's prevention	5 (blocking of a thread).
Corrupted interthread asynchronous communication	5 (the dialog with some sensor/actuators could be timed out).
Astronomical time	4-7 (functional degradation in time, could be recovered at networking level).
LAN Networking: – mutual declaration of the time intervals. – bandwidth reservation.	Performance disbalance: 5 (data frames corruption, congestion possible). 6 (more littering, stalled sessions).
Internet – real time IP and QoS.	5 (heavy retransmission, disbalance of the information, packet loss).

We have considered mostly systems in advanced and cloud-based manufacturing. Here we can explain our motivation and some comments about the given values of the degree of criticality.

Most catastrophic failure is the change of the main clock frequency – very often it happens because of bugs. The table for frequency selection and the modes need special concern.

All input/output operations are timed. The System timer supports all operations and interrupts with integrated devices and timing change could be catastrophic. A number (hundred and more) peripheral devices are integrated: serial and parallel ports (programmable devices that provide analog or digital connection to the selected devices), several timers to be selected for every input/output operation, synchronized and hand-shaking interfaces, analogue comparators, digital- to-analogue and analogue-to-digital converters, power width modulators, USB, Ethernet, and Controller Area Network, etc. Change the timing fields in control registers destroys the current operation and during the time the global behavior.

Time quantum is dynamically changed during the scheduling. It is an internal value in the kernel of RTOS, but it is loaded as a timeout parameter and can be maliciously changed.

Period, execution time, deadline, user thread priority are characteristics of some thread and usually have influence on that thread but again after some time their influence becomes dominating especially for the periodic threads.

Synchronization parameters (locks, barrier counters, synchronization functions) are important for the whole functionality because they change the right distribution of the resources during the time.

Finally, LAN networking and Internet timing depend on factors that are not easy to be controlled by the running RTOS and applications and need special attention.

Astronomical time must be adjusted periodically. Timeout check of the QoS parameters needs to be implemented – it is an overhead procedure.

IX. Conclusion

Timing of cyber-physical systems with real-time and IoT functionality needs more detailed strategy during the architectural development. Significant corruption is possible by underestimation of the timing controls.

In this paper, we have focused on the following topics.

We emphasize that the timing control depends on some more important timing parameters (main clock frequencies, the group execution time-deadlines-periods, networking bandwidth and time intervals, synchronization locking, astronomical time synchronization, deadlock time influence, etc.).

Development trends are supposed to accelerate the stability of the timing by usage of real-time oriented programming techniques. Classical software is to be plugged- in with some patches where a balance between RTOS and application specifics guaranty the reliable timing functionality.

The discussed processing related to the timing control hopefully would provide structured knowledge and assistance during the system development and testing lifecycle.

References

[1] Iliya Georgiev, Ivo Georgiev, **"**Some Analysis of the Timing Parameters in Real-time Embedded Systems", In 2020 International Conference on Information Technologies (InfoTech), September, Varna, IEEE Explore, doi: 10.1109/InfoTech49733.2020.9211071.

[2] D.P.F. Möller, Intrusion Detection and Prevention, In: Cybersecurity in Digital Transformation, SpringerBriefs on Cyber Security Systems and Networks, Springer, Cham, pp 47-75, 2020, https://doi.org/10.1007/978-3-030-60570-4_4.

[3] J. A. Yaacoub, O. Salman, H. N .Noura, N .Kaaniche, A. Chehab, M.Malli , "Cyber-physical systems security: Limitations, issues and future trends", Microprocess Microsystems, **77**:103201, 2020, doi:10.1016/j.micpro.2020.103201

[4] Y.Z.Lun, A.D'Innocenzo,F. Smarra, I. Malavolt, M.D. Di Benedetto, "State of the art of cyber-physical systems security: An automatic control perspective", Journal of System and Software, Volume 149, , Pages 174-216, March 2019, doi.org/10.1016/j.jss.2018.12.006.

[5] Arm® Architecture Reference Manual, https://developer.arm.com/documentation

[6] J. Valvano, Volume 2, Real-Time Interfacing to ARM Cortex-M Microcontrollers (fifth edition), 2017, ISBN: 978-1463590154.

[7] Protected Memory System architecture.Security Architectures – Arm Developer

[8] Yifeng Zhu, Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C; 3rd Ed, 2017; ISBN 978-0982692660.

[9] J. Chen, C. Du, P. Han and Y. Zhang, "Sensitivity Analysis of Strictly Periodic Tasks in Multi-Core Real-Time Systems," in IEEE Access, vol. 7, pp. 135005-135022, 2019, doi: 10.1109/Access.2019.2941958.

[10] Y. Choi, J.Kwon, S. Jeong, H. Park, Y.Eom, "Lightweight deadlock detection technique for embedded systems via OS-level analysis: work-in-progress", in Proceedings of the International Conference on EmbeddedSoftware, Article No. 2 Pages 1–2, September 2018, https://doi.org/10.1109/EMSOFT.2018.8537214

[11] ISO 11898-1:2003, Road vehicles, Controller area network (CAN) , Part 1: Data link layer and physical signalling, ISO 11898, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=33422

[12] IEEE 802.1Qbv. Enhancements for Scheduled Traffic, http://www.IEEE802.org, 2016.

[13] J. Valvano, Real-Time Operating Systems for ARM Cortex-M Microcontrollers (fifth printing), ISBN: 978-1466468863, 2019.

[14] Colin Walls, Embeded RTOS design, 1st edition, Elsevier, E-book ISBN: 9780128228524, 2020.

[15] N. Carreon, S. Lu, R. Lysecky, "Probabilistic estimation of threat intrusion in embedded systems for runtime detection", ACM Transaction on Embedded Computing Systems, January 2021, Article No.: 14, https://doi.org/10.1145/3432590

[16] B. Ngabonziza, D. Martin, A. Bailey, H. Cho and S. Martin, "TrustZone Explained: Architectural Features and Use Cases," *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, Pittsburgh, PA, USA, 2016, pp. 445-451, doi: 10.1109/CIC.2016.065.

[17] N. Koutroumpouchos, C. Ntantogian , C. Xenakis. "Building Trust for Smart Connected Devices: The Challenges and Pitfalls of TrustZone", Sensors 21(2):520 January 2021, DOI: 10.3390/s21020520

[18] Oracle. InterruptedException (oracle.com), https://docs.oracle.com/javase/8/docs/api/java/lang/InterruptedException

[19] Oracle. AsynchronouslyInterruptedException (oracle.com), https://docs.oracle.com/javase/realtime/doc_2.2u1/.