DOI: 10.35598/mcfpga.2020.008

The Use of Percepio Tracealyzer for the Development of FreeRTOS-based Applications

Maksym Khomenko
ORCID 0000-0001-9084-3527
EMT Department
Bonn-Rhein-Sieg University of Applied Sciences
Sankt-Augustin, Germany
maksym.khomenko@h-brs.de

Oleksandr Velihorskyi
ORCID 0000-0002-8256-7339
Biomedical radioelectronic apparatus and system department
Chernihiv National University of Technology
Chernihiv, Ukraine
oleksandr.veligorsky@inel.stu.cn.ua

***Abstract*—This paper discusses some problems of development and testing of FreeRTOS-based application. The use of Tracealyzer software tool is proposed to make this process more convenient. The benefits of such usage have been shown on typical issues that can be met in development and debug phase.**

Keywords—microcontroller, RTOS, debug, embedded system, tracealyzer.

I. INTRODUCTION

Microcontrollers play a key role in the majority of nowadays embedded systems. Among the variety of 8-, 16- and 32-bit microcontrollers the last ones (which mostly have an ARM core) take a dominant position in the embedded world [1]. They usually have more memory, different peripheral modules and of course higher frequencies and computational power compared to 8- and 16-bit microcontrollers for the reasonable price. The software of embedded systems also becomes more complex and often utilizes real time operational system (RTOS) such as FreeRTOS [1, 2] to gain flexibility and multitasking. However the development, test and debug of RTOS-based programs brings a new challenges to the software design engineers. Some of these challenges are task synchronization, resources sharing between tasks, task priority management and other.

Helping engineers to make the development and debug process less problematic Swedish company Percepio have developed software tool Percepio Tracealyzer. This tool together with library which should be linked with FreeRTOS-based program, visualize all the objects and events inside operational system that makes understanding of program flow and debug process easier and convenient

II. TRACEALYZER MAIN CONCEPTS

The tool consists of two separate parts as it is mentioned above: the trace recorder C library and main program with graphical user interface. The library is to be compiled with user FreeRTOS-based program for the target platform. It includes three configuration header files (*trcConfig.h*, *trcSnapshotConfig.h* and *trcStreamingConfig.h*) that can be used to set up one of two recorder mods of operation and various parameters of data tracing. One mode of operation is called “Streaming” in this mode the data collection and visualization is performed in real time. Collected data are constantly transmitted to the computer for the visualization through some communication interface (USB or Ethernet) or through hardware debugger (not all debuggers are supported). Another mode of operation is called “Snapshot” in this mode data are stored in the previously allocated (statically or dynamically) memory buffer on the target device and can be read to the computer trough any hardware debugger tool. The detailed description of how to include this library to custom FreeRTOS-based project and setup preferable tracing mode are given in [3].

The Tracealyzer itself is a powerful tool of RTOS data analysis. It contain huge batch of different views that can be open and allocated at the screen as part of main program or as separate window. So, the detailed description of all views can’t be done in the scope of this paper. However most commonly used views are present in the Fig. 1.

Trace view (marked with a green frame in the Fig. 1) shows all tasks and RTOS events on the time line. It gives major information about system behavior. CPU load graph (marked with a yellow frame in the Fig. 1) represents information about CPU usage by different tasks in time. Selection details window (marked with a blue frame in the Fig. 1) shows some major parameters of the task slice or system event selected in the Trace view. Filter window (marked with a violet frame in the Fig. 1) can be used to enable or disable single objects or service events on other views of Tracealyzer. Such possibility makes inspection of the program flow more convenient because information that have no interest for the current analysis can be switch off. The brown frame in the Fig.1 shows different time related parameters of the tasks, the parameter of interest can be selected from the drop-down menu of this window (execution time parameter is selected in the Fig.1). Toolbar panel is placed on the left side of the main program window (red frame in the Fig. 1). It provides quick access to all other possible views of the program which also can be found in “Views” menu. In addition, toolbar contains control buttons that are used to start/stop streaming in streaming mode and to make snapshot in snapshot mode.

Concerning selection of the operation mode is depend on the purpose of analysis. In development or in debugging phase for the easily reproduced bugs the snapshot mode can be used. Its main drawback is relatively short time period (usually some seconds) that can be stored due to device RAM limitation. On the other hand streaming mode of operation can be used when long time period should be

stored for analysis. Consequently this mode is useful for the debugging of randomly and rare appeared bugs.

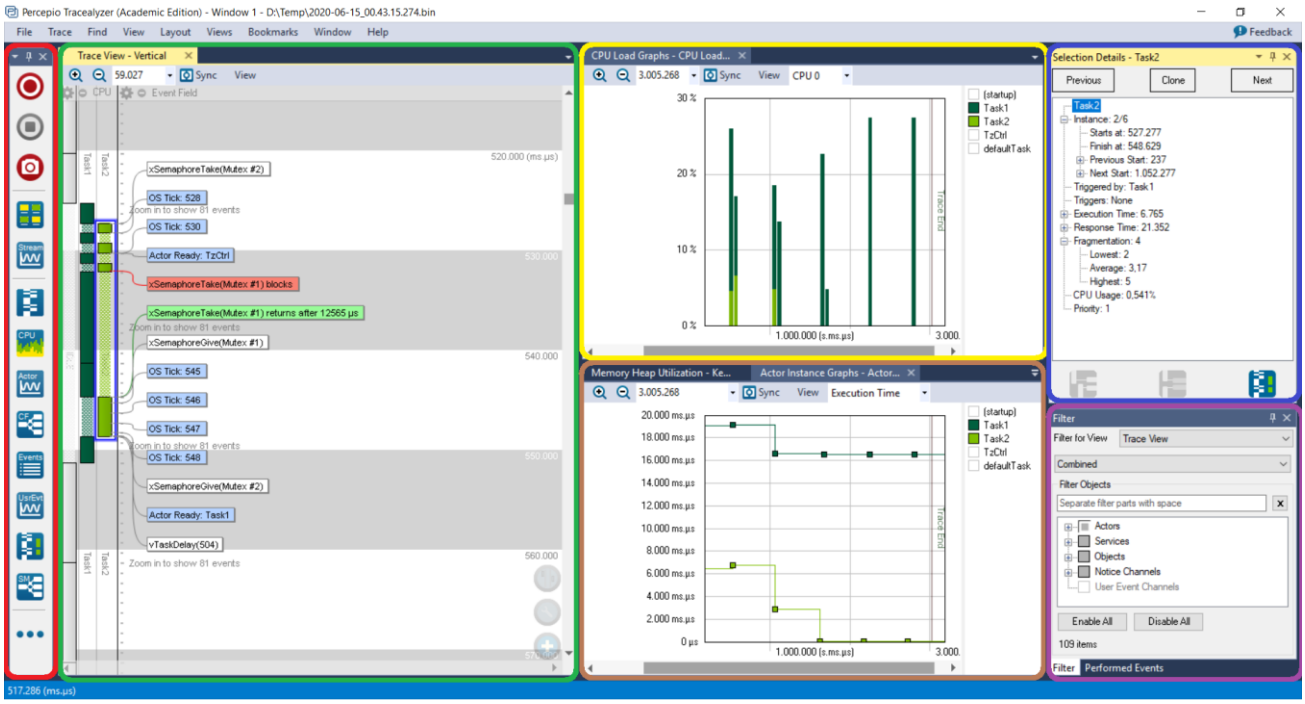


Fig. 1. Percepio Tracealyzer window layout

III. TRACEALYZER USE CASES

All use cases listed in this section utilize the Nucleo-F429 board (build on STM32F429 microcontroller) as a hardware base. Also all data traces have been taken in snapshot mode.

A. *Periodicity of Task Execution*

In RTOS-based programs most of the tasks do not need to run all the time. Usually they are triggered by other tasks and interrupts or use RTOS time management API functions to execute periodically and then go back to the blocked state till the next synchronization event. The demand of task execution period stability can be different for various tasks. For example, in control application the data sampling task should have low jitter to avoid system performance degradation [4].

FreeRTOS has two API functions to perform time delay: *vTaskDelay()* for the tasks where period jitter is not critical, *vTaskDelayUntil()* for the tasks with strict requirements to the period stability [2].

To compare the results of using different time delay API functions the test program was developed. It has two tasks one of which uses *vTaskDelay()* and other uses *vTaskDelayUntil()*. Both tasks have same time delay value (150 ms) and the same priority. The resulting periodicity of tasks execution is shown in the Fig. 2 (a). The first task shows stable period of execution as expected, while the second task has variations and shift in period due to use of simple *vTaskDelay()* API function which does not take in to account the time of task execution.

Fig. 2 (b) shows the results of the same program but with slightly changed conditions. Now the second task has priority higher than the first one. Despite the fact that the first task uses *vTaskDelayUntil()* function to provide a stable period it suffer from period deviation at the times when it overlap with the second task having higher priority.

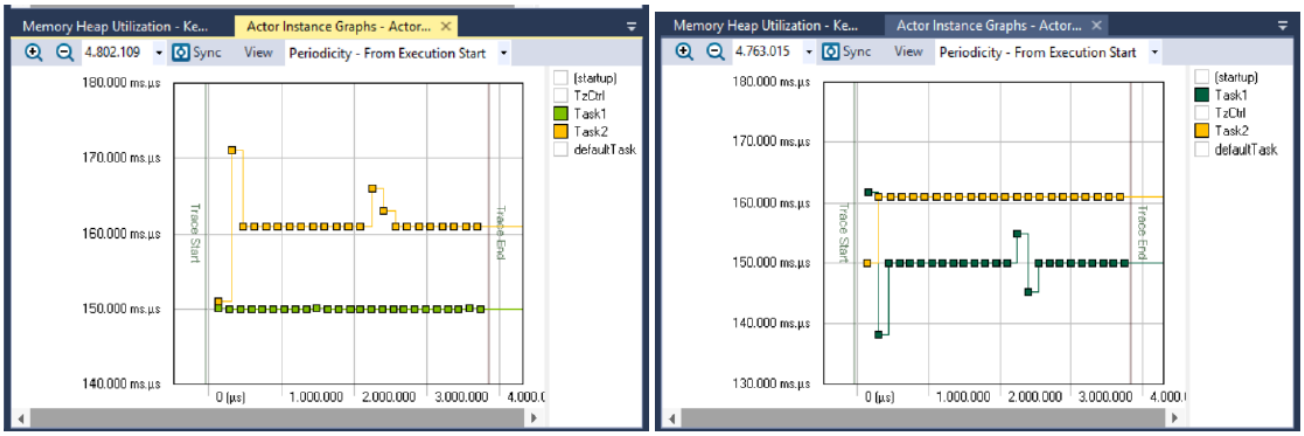
In the real system such inaccuracy in priority assignment can lead to the tricky bug in system behavior. But as it is shown in the Fig. 2 using Tracealyzer helps easily find this problem early in the development stage.

B. *Mutual Execution Problems Accessing Shared Resources*

In the multitasking system a special care should be taken when tasks want to access shared resources (this can be global variables, peripheral modules internal or external) to prevent simultaneous change of such resources from different tasks.

One of the commonly used approaches treating this issue in RTOS is to protect the access to shared resource with special mechanism called mutex (abbreviation of mutual execution). Each task before doing something with shared resource should take a mutex that protect this resource. If the mutex is successfully taken than task can access the resource but if not this means that some other task has already gain access to it and the first one should wait or skip the actions with such resource. When task finishes actions with shared resource it should give the mutex so that other tasks waiting for this resource can gain an access to it.

In the FreeRTOS mutex is a special type of binary semaphore therefore an API function for taken a mutex is *xSemaphoreTake()* and API function for given a mutex is *xSemaphoreGive()*.The potential issue using mutex is possibility of so called “Deadlock”. “Deadlock” occurs when two (or more) tasks can’t continue execution because they are waiting for the resources held by each other. Fig. 3 illustrates such kind of situation that has occurred with two tasks and two mutexes.



(a) (b)

Fig. 2. Periodicity of tasks execution when tasks priority is equal (a) and when task1 has lower priority than task2 (b)

Test program showing this issue was developed. It consist of two tasks with equal priority that shares two resources global array and UART peripheral module and use two mutexes respectively. The first task 1 writes the message to the UART and fill global array with pseudo random numbers. The second task also writes the messages to the UART, and calculates the sum of array elements. At some point task 1 tries to get access to the UART while it is used by task 2. As the result the terminal stops displaying any messages (see Fig. 4). But what is the reason of such behaviour, how to find the roots of this problem? The Tracealyzer can help to find the solution.

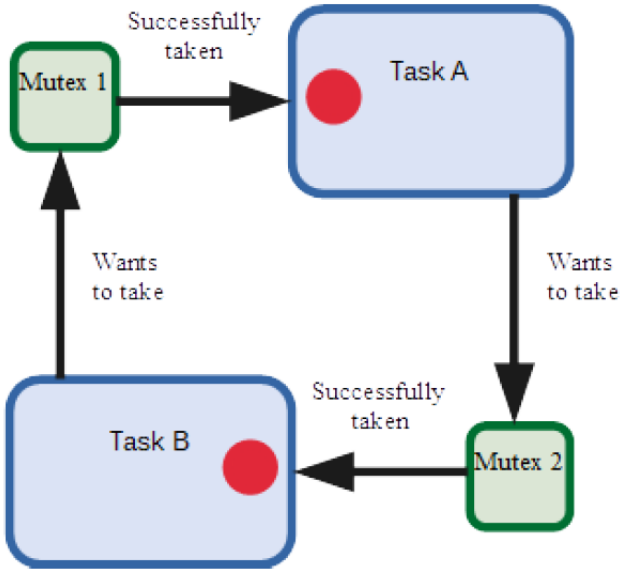


Fig. 3. Example of Deadlock situation with two tasks and two mutexes

From the trace view shown in the Fig. 5 can be clearly seen that both tasks stop the execution. Also there is shown “Mutex ownership diagram” next to the tasks on the trace view which were activated through “Intervals and State Machines” view by selecting diagrams of interest from the predefined ones. This diagrams shows that both mutexes are not free from the moment of tasks stop and till the end of trace. Global array access mutex is held by the first task while UART access mutex is held by the second task.

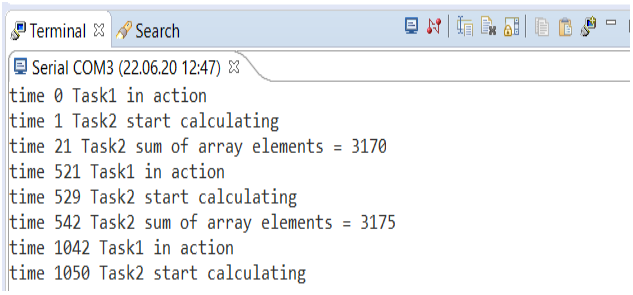


Fig. 4. UART messages from the test program in terminal window

Such analysis of system behaviour definitely shows that the “Deadlock” has occurred in the program. And now when the problem has been identified and localized it can be solved by the means of tasks logic reorganization in the part where they interact with mutexes getting access to shared resources.

IV. Conclusions

This paper discusses some issues that can be met by the embedded system developers dealing with the FreeRTOS and multitasking environment. The Tracealyzer software is proposed as useful tool that can make development and debug processes easier and faster. The benefits of this software tool have been proved on examples where it helps to figure out the problems with periodicity of task execution and “Deadlock” state of two tasks using shared resources and mutexes. As the result, Percepio Trasealyzer can be recommended to be used not only in development process but also in education courses for embedded system engineers.

References

[1] References“2019 Embedded Markets Study. Integrating IoT and Advanced Technology Designs, Application Development & Processing Environments.”, Embedded.com, 2020. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf. [Accessed: 20- Jun- 2020]

[2] R. Barry, “Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide”, Freertos.org, 2020. [Online]. Available: https://freertos.org/Documentation/161204_Mastering_the_FreeRTO

S_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf. [Accessed: 20- Jun- 2020].

[3] "Quick Start Guide – Tracealyzer for FreeRTOS", Freertos.org, 2020. [Online]. Available: https://percepio.com/gettingstarted-freertos/. [Accessed: 20- Jun- 2020].

[4] P. Marti, J.M. Fuertes, G. Fohler, K. Ramamritham, "Jitter compensation for real-time control systems", Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001), December 2001, pp 39-48.

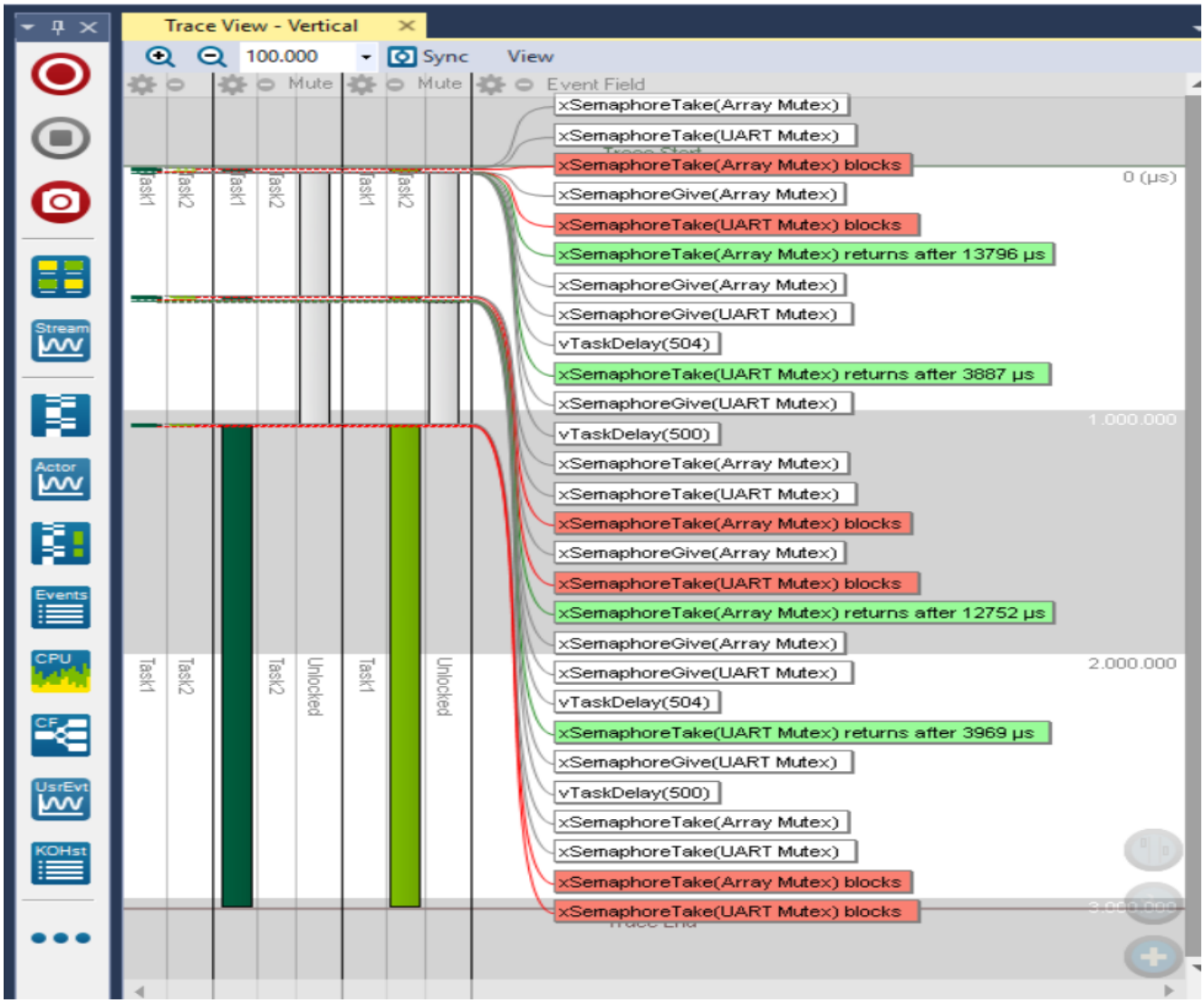


Fig. 5. Trace view of program with "Deadlock"