ISSN 0361-7688, Programming and Computer Software, 2024, Vol. 50, No. 5, pp. 343–355. © Pleiades Publishing, Ltd., 2024.
Russian Text © The Author(s), 2024, published in Programmirovanie, 2024, Vol. 50, No. 5.

Unified Processing of Events and Co-routines in Embedded Program

P. V. Minin[a,*]

[a] *DORS R&D LLC 7-2, Federativny prospect, Moscow, 111399 Russia*

**e-mail: p.minin@dors.ru*

Received March 24, 2023; revised November 4, 2023; accepted May 12, 2024

Abstract—A new architectural approach to real-time embedded programming is described. A bare-metal program is written in C/C++ and combines event-driven technique with concurrency based on co-routines. Events are processed by soft real-time core at the priority level of software generated interrupts. Event is first posted to input queue of the core and then processed by invocation of its event handler. A special case of event is co-routine, its resumable function being a co-routine event handler. The co-routine that either yielded or needs to be resumed is queued for event processing once again. As a result, it is processed multiple times until execution of resumable function comes to the end of its operator sequence. Different levels of processing priority may be assigned to an event. Soft real-time core could be further expanded to run on symmetrical multiprocessor hardware. A combination of co-routines and basic events could easily be used in fork/join model. Concurrency constructs resemble those of Go and occam languages. Virtually all classic types of synchronization primitives could be implemented. The new approach was implemented for various ARM and Blackfin processors in C++ language as portable DORSECC library. This library was further used to program real-time embedded systems for mass-produced banknote sorting machines. One type of systems was used to recognize and validate banknote images by the method of cascade of one-class classifiers. The other system worked as a motion controller and used finite automata to control sensors and actuators. The total number of systems in operation is currently over 20000. The event and co-routine core in these systems provides average event processing time in the range of dozens of microseconds with sub-microsecond overhead time per each event.

Keywords: real-time, embedded, bare metal, concurrency, event-driven, co-routine, C++, software interrupt, synchronization, threaded code

DOI: 10.1134/S0361768824700154

1. INTRODUCTION

Embedded systems are often built using bare metal program written in C or C++ or a combination of these languages. Bare-metal (self-sufficient) program contains all required software components of the system and needs no operating system (OS). Embedded programs running in a real-time OS (FreeRTOS, ThreadX, WxWorks, LynxOS, PikeOS etc.) and even a general-purpose OS like Linux are used much less frequently. Rejection of OS in favor of bare-metal program is most often caused by resource limitations of the system or difficulties of real-time hardware control. A bare-metal program avoids the overhead associated with the operating system. It provides the solution that is most optimal in terms of performance and required hardware resources. However, the drawback of this optimization is complexity of programming. The greatest difficulties are associated with concurrent code execution as well as asynchronous processing.

Real-time operating systems provide the ability to use threads as the main mechanism of concurrency and parallelism. Up to now, for a C/C++ bare-metal program, the programming language by itself provided no concurrent execution means. Only recently did C++ implement the co-routine mechanism, starting with the C++ 20 specification. However, C++ co-routines have not yet found widespread use in embedded systems, partly due to the complexity of the concept and unfamiliar terminology used to describe it.

1.1. Co-routines

Co-routines [1] differ from threads since they implement concurrent execution of several sequences of operators where control is explicitly transferred between these sequences without the possibility of preemption. Library implementations of co-routines in C [2, 3] and C++ [4–6], which are divided into stack-based and stackless, have been known for quite a long time. The concept of a stack co-routine is as close

as possible to a traditional thread in terms of expressive power. The downside of high expressiveness is the high overhead of co-routine management. For stackless co-routines, the stack variables of one co-routine could not be saved for the period when control is transferred to another co-routine. This imposes significant restrictions on the programming style. On the other hand, stackless co-routines provide minimal memory requirements and time overhead because switching from one co-routine to another does not require saving and restoring the stack and associated context elements. The comparison of stacked and stackless co-routines inspired intense discussion that is partially reflected, for example, in [7]. An overview of the applicability of co-routines in embedded systems is given in [8].

Some implementations assign a priority value to a co-routine. Prioritized launching of co-routine execution could be achieved in two independent ways: either with co-routine dispatcher or using the priority of the execution thread. When a co-routine pauses or suspends its execution, the dispatcher transfers control to the highest priority co-routine ready for execution. In case the OS provides the program with execution threads of different priorities, a higher priority thread can be allocated to execute higher-priority co-routine. Examples include the Kotlin language [9] and the FreeRTOS real-time OS [10]. Note that a bare-metal program always has a single available thread of execution, so it is impossible to use different priority threads to prioritize co-routines.

There are dozens of C/C++ language implementations of co-routines currently known, none of which has yet become a de facto standard. Noteworthy is the fact that in the vast majority of cases these implementations do not involve such a powerful asynchronous real-time mechanism as the processor interrupt system. Moreover, interrupt processing is usually completely isolated from the execution of the deterministic sequence of processor instructions generated by the compiler for the co-routine code.

1.2. Event Processing

A significant part of physical device control tasks are described using a concept of finite state machine. A natural representation of a finite state machine is an event-driven system [11]. For many problems, the use of threads leads to complicated and suboptimal solutions, while an event-driven approach to the same problems provides the most simple solutions [12, 13].

Hardware events in simple embedded real-time systems are traditionally handled using interdependent interrupt handlers. This approach allows for maximum performance, but is based on programming techniques that often resemble tricks, generate hidden errors and are difficult to verify. Specialized frameworks like Quantum Leaps [14, 15] are used for functionally complex event-driven embedded systems. In practical programming, a design pattern known as Observer [16] is often used as the basis for an event dispatcher. For each event received, such a dispatcher performs a procedural call to the corresponding handler.

General-purpose operating systems employ a special middle-priority mechanism for deferred processing of data generated in hardware interrupts. In fact, this mechanism is closely related to event handling. Examples include deferred procedure calls (DPC) in MS Windows [17], or deferred tasks in Linux [18], which are executed in a kernel thread (Workqueue) or in the context of a software interrupt (Tasklet).

To organize sequential processing of events occurring asynchronously, an input queue of an event processing system is usually used.

2. RESEARCH MOTIVATION

Stackless co-routines are of significant interest as a multitasking instrument in embedded systems that do not use an OS. The main advantage of a stackless co-routine compared to OS thread is its significantly lower overhead of task switching. This fact is primarily related to the size of the context to be switched, which in the case of a stackless co-routine can be reduced to one or two processor words.

However, the classic co-routine does not suit well for real-time operation because it cannot provide a guaranteed response time to an asynchronous stimulus such as a hardware interrupt. This can be seen in comparison with prioritized preemptive threads (tasks or processes) used in real-time OS. The wakeup latency of a thread that was previously suspended and is waiting for an interrupt could be expressed as $T_{TL} = T_{IL} + T_{TSW} + T_{OVH}$ (1), the preemption being taken into account. Here T_{IL} designates interrupt handler invocation latency and T_{TSW} is the context switching time between the threads. T_{OVH} designates the net overhead duration including interrupt handler execution time to the moment of signaling the thread, and also thread synchronization and dispatching overheads. As it can be seen, T_{TL} is a sum of durations of OS calls that are well determined and generally small.

Classic co-routines in a bare-metal program are executed in a single thread and provide a cooperative type of multitasking. To respond to a stimulus by launching respective co-routine, this single thread should run a dispatcher controlling execution of a set of co-routines. The dispatcher can send the next co-routine for execution only after the currently executing co-routine decides to yield the processor. The stackless co-routine resumption latency in response to a stimulus is $T_{CL} = T_{IL} + T_{CY} + T_{OVH}$ (2), where T_{CY} is the current co-routine execution time from the moment the stimulus is processed by the interrupt handler until the processor is released. The current co-routine yields the processor when its execution sequence