Latency Characteristics of Event-Driven Task Scheduler Embedded in Neuron Chip

Marek Miśkowicz

Department of Electronics, AGH University of Science and Technology, Kraków, Poland

Summary
The paper presents the experimental measurements of the temporal measures of the task scheduler embedded in Neuron Chip microcontroller. The investigated task scheduler dynamically manages sharing the Application Processor between tasks in smart devices developed in LonWorks control networking technology. Since the event-driven architecture does not provide the explicit management of time, the evaluation of the latency characteristics of software has to be handled indirectly. We have constructed a set of testing procedures that connect the scheduler operations with input/output actions. The latter can be analyzed by the standard measurement instrumentation. Finally, some information that can support timing-aware Neuron Chip programming is provided.

Key words:
Scheduling Algorithms, Events, Delay Analysis, Fieldbus, Microprocessor Control, System Architectures, Timing Analysis

1. Introduction

Local Operating Networks (LON, LonWorks) is one of the leading technologies in sensor and control networking addressed to a wide range of applications [3,4]. LON has become a classic solution in building automation, and home networking including all key building automation subsystems: heating, ventilating, and air conditioning, lighting, security, fire detection, access control, energy monitoring, etc. LonWorks platforms are also used, among others, in semiconductor manufacturing, pulp and paper equipment, material handling, textile machinery, petrochemical, food and beverage, automotive, and wastewater treatment.

The core of LonWorks technology (registered also as EIA-709 standard) is the Neuron Chip microcontroller, a system-on-chip designed to provide intelligence and networking capabilities to distributed control devices. The Neuron Chip includes three 8-bit in-line processors (MAC Processor, Network Processor, and Application Processor) that support both communication and application processing [5].

The MAC Processor and Network Processor, execute the lower 6 layers of the LonTalk/EIA-709.1 protocol. The Application Processor executes the user code (the *application program*) together with the operating system services. Using hierarchical multiprocessor system, the Neuron Chip provides the parallel processing of application data and communication messages since multiple concurrent software processes are executed at different stages of protocol stack.

The Neuron Chip is programmed in Neuron C, a language based on ANSI C optimized and enhanced for distributed control applications. One of the crucial differences between the ANSI C and the Neuron C code is a new structure of the application program. As distinct from ANSI C, a program written in Neuron C does not include the `main()` construct. Instead, Neuron C exploits a multitasking real-time scheduler built in firmware that allows the programmer to express logically parallel event-driven tasks, and to control the priority execution of these tasks [6].

More specifically, the scheduler executes user-written tasks in response to events or conditions specified in `when` clauses. When a specified event or condition becomes true, the associated task code is executed. The task responded to important events may be assign to priority `when` clauses. The tasks associated with `priority when` clauses are executed before checking non-priority clauses. The non-priority tasks are executed only if no priority event or condition is evaluated to true. The scheduler checks `when` clauses in the round-robin order according to their appearance in the Neuron C program.

As stated, the Neuron Chip firmware scheduler may be treated as a real-time node operating system that dynamically manages sharing the Application Processor between tasks waiting for the execution. The event-driven scheduling of Neuron Chip tasks is a main paradigm of the event-based architecture of LonWorks applications.

The other fundamental concept in LonWorks/EIA-709 technology are *network variables* that make up the network interface of the application program [6]. Whenever an *output network variable* is modified, its new value is propagated across the network to all devices with *input network variables* connected to that output network variable. Roughly speaking, the network variables are logical inputs/outputs of the nodes. The concept of network variables simplifies data sharing between smart

Manuscript received December 5, 2007
Manuscript revised December 20, 2007

sensor and actuator devices.
The event-driven software architectures refer to object oriented programming techniques where each node is considered as an object. The network variables can be considered as the object *public data* while conventional internal variables are the object *private data*.
The event-driven architecture is built on the causal relationships among external events and the actions executed by a system, i.e., on "why" something happens, whereas the time-driven model focuses on the timing of actions, i.e., on "when" something happens. In the former case, actions are executed "as soon as possible" on the arrival of events, and the time can be handled by timing signals which are treated as external asynchronous events.
Conversely, in the time-triggered scheduling actions are executed "at the right time" according to a schedule, and external events can be handled by polling mechanisms [1,2]. In fact, the time-driven model can be considered as the particular case of the event-driven one since events triggering the task execution might be defined by repeating timer expirations.
The event-driven control can react to events by associating them with tasks to be delivered as soon as possible. The system is supposed to be waiting for incoming events and does not manage the concept of time. Since switching between tasks is caused by events, time is no longer a suitable independent variable in system modelling. The actual execution of actions is often left to the runtime support. However, the time is indeed a critical issue for control. The concepts of time and speed play a major role in the process control area, and especially, in real-time applications [1,2].
The paper presents the experimental results of the temporal measures of the embedded Neuron Chip task scheduler. Since the event-driven architecture does not provide the explicit management of time at the basic level, the evaluation of the latency characteristics of software has to be handled indirectly. We have constructed a set of testing procedures that connect the scheduler operations with input/output actions. The latter might be analyzed by the standard measurement equipment.
We have selected several measures describing the scheduler latency characteristics, e.g. the scheduler restart overhead delay, the minimum inter-`when` delay, the minimum context switching delay. These measures are treated as the exemplification of a set of parameters related to the execution time of various Neuron C instructions that are evaluated using the method proposed in the present paper.
Thus, the contribution of the paper is twofold. First, it relies on the evaluation of task scheduling delay in LonWorks smart devices. Second, it consists in the presentation of a measurement method that is universal and can be used for the evaluation of software latency characteristics in other real-time systems.
Except comments reported in [5], which unfortunately include some mistakes and ambiguities, the studies on the Neuron Chip scheduler timing, to the author's knowledge, have not been published. The present study is the extension of the previous author's conference paper [10].

2. Algorithm Specification

2.1 `When` Clause Definition

The specification of the scheduler algorithm is presented below. As was mentioned, events that control the tasks execution order are defined through `when` clauses. A `when` clause contains an expression which, if evaluated as true, causes the body of code (the *task*) following the expression to be executed to completion [6] :

```
when (event)

{
        task;
                    }
```

A *task* is a sequence of statements in a Neuron C program. The end of the task constitutes a *critical section* of the application program. Once begun each task runs to completion. The critical section of the program controls flow of application messages exchanged between the Application Processor and Network Processor. The following operations are executed at the end of a critical section [6,7] :
- outgoing messages and output network variable updates are sent,
- incoming messages and input network variable updates are processed,
- timers are examined to check if they are expired,
- the watchdog timer is reset to keep it from timing out.

The `when` clauses cannot be nested. Instead, the conventional conditional statements `if`, `while`, and `for` within the `when` clauses might be used.

2.2 `When` Clauses Evaluation Order

The scheduler evaluates `when` clauses in round-robin order. As stated, each `when` clause is evaluated and, if true, the associated task is executed. If the `when` clause is false, the scheduler moves on to examine the following `when` clause. After the last `when` clauses the scheduler returns to the top and moves through the group of `when` clauses again. Thus, the position of the task in the application program does not determine its potential precedence in the access to the Application Processor. However, the order of appearance of clauses in the application program defines the order of checking of corresponding events.

The `priority` keyword can be used explicitly to designate `when` clauses that should be evaluated more often than non-priority `when` clauses. If there are many priority `when` clauses in the application program, they are evaluated in the round-robin order. If none of the priority `when` clauses evaluates to true, then the non-priority `when` clauses are tested according to the round-robin order as described earlier. After the execution of any non-priority task, the scheduler checks the priority `when` clauses.

2.3 Task Execution Order vs. Event Chronology

If the events triggering the tasks occur rare, the order of task executions is determined by the order of corresponding event occurrences. Thus, the scheduler keeps the event chronology if it is lightly loaded, i.e., when no more than one event waits for a task execution. The delay between the event occurrence and the beginning of the task execution is then negligible since the tasks are run "as soon as possible".

If more than one event triggering task request occurs, the Application Processor may *not* run the tasks in the order of corresponding event occurrences. In other words, the subsequent event might be served before some event that occurred earlier. It depends on the phase of event occurrences with respect to the current position of scheduler testing loop as will be shown in Fig. 1.

If the events are more frequent, the tasks wait for execution longer time. The corresponding latency depends on the number of `when` clauses in the application program and the tasks complexity. In the extreme case, *all* the events tested in `when` clauses are evaluated to true which cause all the tasks to be executed. This is the worst-case scheduler load that may be defined as the *saturation condition*. If the scheduler is saturated, the order of task execution is round-robin and corresponds to the order of appearance of `when` clauses in the program listing. Thus, although the Neuron Chip scheduler is *dynamic* in general, its operation becomes *static* under saturation conditions.

We summarize the description of the scheduler operation by analyzing the following code :

```
when (event a)
{
        task A;
                }

priority when (priority_event)
{
        priority_task;
                }

when (event b)
{
        task B;
                }

when (event c)
{
        task C;
                }
```

As demonstrated, the application program consists of three non-priority `when` clauses defined by the events (*a*, *b*, *c*) and the corresponding tasks (*A*, *B*, *C*), and a single priority `when` clause.

The diagram of the scheduler operation is depicted in Fig. 1. The exemplified instants of events occurrences are specified on the time axis. As follows from Fig. 1, the scheduler is saturated since all the events tested are evaluated as true.

In particular, Fig. 1 presents the situation when the task execution order is not consistent with a chronology of event occurrences. Namely, the task *C* is executed before the task *A* although the corresponding event *c* have occurred later than the event *a*. Furthermore, as shown in the diagram, the execution order of the non-priority tasks is static and round-robin.

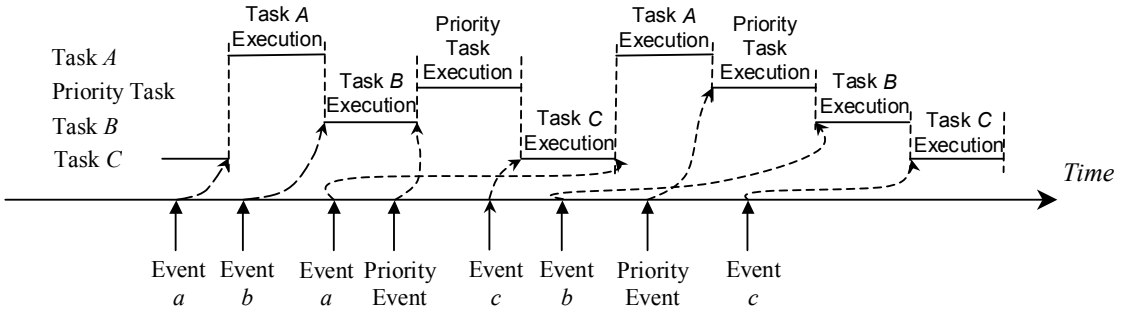

Fig. 1 Task scheduling with frequent events (saturation status).

2.4 Events

The events defined in `when` clauses fall into two categories: *predefined events* and *user-defined events*. The user-defined event can be any valid Neuron C expression.
The predefined events use keywords built into the compiler and encompass the following classes :

- system wide events (e.g. `reset`, `online`, `offline`),
- input/output events (e.g. the value read from the I/O devices has changed),
- timer events (`timer_expires`),
- message and network variable events (e.g. the update of the network variable has been received).

2.5 Classification of Neuron C Scheduling

The Neuron C scheduling is applied to single node operations only so it performs scheduling in the *local* sense. LonWorks/EIA-709 technology offers limited ability to maintain the *global* scheduling because nodes in the LonWorks network have not access to *global time* (i.e. the nodes are not permanently synchronized). Hence, a *timestamp* can only be interpreted within the scope of a single node. Instead, the synchronization is established every time a packet transmission occurs. The transmitter transmits a preamble before sending the packet to allow the other nodes to synchronize their receiver clocks [5].
As a matter of fact, the concept of synchronized LonWorks network has been developed and presented in [9]. However, the extra wire connected the selected I/O pins of the Neuron Chip in all nodes in the network is required. This wire is designed for distribution of the synchronization signal generated by the dedicated node. Since the Neuron Chip does not support a hardware real-time interrupt input, the node processor must poll the hardware synchronization signal pulse. In [9], a periodic scheduler has been implemented with the calculation of the worst-case queuing delay.
From the point of view of the formal classification we can characterize the Neuron C scheduling as :

- *dynamic* since the scheduler makes its scheduling decisions at run-time, selecting one out of the current set of ready tasks,
- *non-preemptive* because the currently executing task will not be interrupted until it decides on its own to release the allocated resources provided that the watchdog timer does not expire,
- with optional priority system (defined by `priority when` clauses).

2.6 Bypass mode

The task scheduler built in the Neuron Chip firmware operates using a predefined algorithm and a user cannot configure or to change its behavior (except the scheduler reset mechanism [6]). However, if the task scheduler consists of a single `when` clause, which always evaluates to true and never returns, then the algorithm is in fact deactivated. In this way a user-defined scheduling algorithm might be developed on the basis of the conventional conditional statements (`if`, `for`, `while`). However, all the scheduling instructions must be included *within* the task associated with the single `when` clause. Moreover, the user is responsible for all event processing, i.e. to update the watchdog timer (using `watchdog_update()` function) and to define explicitly the critical section boundaries in the application program (using `post_events()` function). A method of Neuron Chip programming stated above is called *bypass mode* [6].

3. Definition of Scheduler Latency Measures

In real-time contexts, event-driven languages are effective for systems with sporadic actions that must be executed as soon as possible. However, there is a finite delay associated with each scheduler operation. If the interarrival time between consecutive events is large enough in relation to the scheduler overhead, then the scheduling-induced latency can be neglected. Otherwise, they have to be taken into account in the application development.
The time required for the scheduler to evaluate the same `when` clause in a particular user application code is to a larger extent a function of:

- the size of the user code,
- the total number of `when` clauses,
- the frequency of occurrences of the events associated with those `when` clauses.

The first two factors depend on the complexity of the user code. The latter depends on the application environment and a rate of its state changes in the time [5]. It is therefore impossible to specify a nominal value for this delay in general. Moreover, there is no limit for the upper bound of the scheduling delay.

3.1 Scheduler-Related Timing Measures

However, it is possible to evaluate the minimum (best-case) delay induced by the Neuron Chip task scheduler. It constitutes the lower bound for the task execution time. We have selected the following measures describing the scheduler latency characteristics:

- the scheduler restart overhead delay,
- the execution time of `io_out()` function call,
- the minimum inter-`when` delay,
- the minimum context switching delay,
- the bypass mode timing properties.

The list of measures presented above might be extended.

These measures are treated as the exemplification of a set of parameters related to the execution time of various Neuron C instructions that can be evaluated using the method proposed in the present paper.

3.2 Measurement Method

We have developed a measurement method of selected parameters related to scheduler-induced delay. Our approach belongs to indirect methods where the dedicated code that refers to the input/output operations is used in order to extract the corresponding scheduler parameters from measurements. More specifically, in the proposed method we use a set of testing procedures, written in Neuron C, that connect the scheduler operations with input/output action. Thus, the timing properties of output signals generated on the Neuron Chip input/output port include information about several measures related to the scheduler latency. The measurements of the corresponding signals have been done using a standard digital oscilloscope. The experimental results of the oscilloscope measurements are reported in Section 4.

3.3 Measurement Equipment

We have carried out experiments using NodeBuilder Development Tool, which is an integrated hardware and software equipment operating with a microcomputer that provides a network development environment to prototype LonWorks devices [8]. The hardware consists of LTM-10 LonTalk Module with Neuron Chip, 32 kB flash memory, 32 kB static RAM, 10 MHz crystal oscillator, and custom Neuron Chip firmware. The software includes the Neuron C cross compiler for creating Neuron Chip object code.

4. Experimental Results of Scheduler Timing

4.1 Scheduler Restart Overhead

We have started to investigate the scheduler latency characteristics from a simple procedure consisting of two non-priority `when` clauses that always evaluate to true as shown below :

```
IO_2 output bit test_signal;
when(TRUE)
{
       io_out(test_signal,0);
                                    }

when(TRUE)
{
       io_out(test_signal,1);
                                    }
```

The output object, called `test_signal` referring to the pin IO_2 of the Neuron Chip I/O port, is declared.
The goal of the measurement is to isolate and extract the temporal measures associated with the scheduler. Since the processing of `when` clauses is round-robin, the Neuron C procedure performs alternating activation of IO_2 pin. As a result, the square waveform is generated on the pin IO_2 of the Neuron Chip I/O port (see Fig. 2). The measurements have been performed using digital oscilloscope for Neuron Chip with 10 MHz input clock.

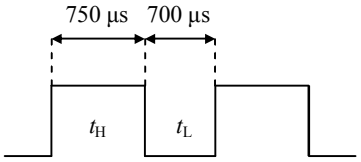

Fig. 2. The evaluation of the scheduler restart overhead delay.

Although a structure of both tasks is the same, the durations of the high t_H and the low logic levels t_L slightly differ. The reason of that is a position of a particular `when` clause in a scheduler loop. During the generation of the low logic level t_L the following operations are run :

- the execution of `io_out()` function,
- the context switching at the end of the task.

When the high logic level t_H is generated, the extra operation is performed apart from the actions specified above, i.e. the scheduler loop is restarted. Thus, the scheduler restart overhead t_{SCH} can be evaluated as the difference between the time intervals t_H and t_L :

$$t_{SCH} = t_H - t_L = 750\ \mu s - 700\ \mu s = 50\ \mu s.$$

Note that the scheduler restart overhead causes an extra delay only between the *last* and the *first* `when` clause in a scheduler loop.

4.2 Minimum Inter-`When` Delay

The testing procedure presented above allows to find the minimum delay between the evaluation of the consecutive `when` clauses that we call the *minimum inter-*`when` *delay*. As follows from Fig. 2, the minimum inter-`when` delay is 700 µs long for 10 MHz input clock.

4.3 Execution Time of `io_out()` Function Call

To estimate the execution time of `io_out()` function, let us analyze timing of the code of a single `when` clause :

```
IO_2 output bit test_signal;
when(TRUE)
{
       io_out(test_signal,0);
       io_out(test_signal,1);
                                    }
```

The diagram of the waveform generated on the pin IO_2 of input/output port is shown in Fig. 3. The narrow

negative pulses are separated by long time intervals because the duration of the high logic level t_H is determined by the *context switching* delay, i.e., the latency related to switching the tasks executed by the processor.
However, since the first call of `io_out()` function in the task is not followed by any extra firmware operations, we can assume that the duration of a single negative pulse, which is 70 μs long, is equal to the execution time of `io_out()` function t_{IO}. In particular, it means that the function `io_out(test_signal,1)` is called at the instant of the falling edge of the test signal on the pin IO_2 (see Fig. 3) because the appropriate rising edge comes with 70 μs delay.
The measured time interval t_{IO}, corresponds to `io_out()` function execution time. However, we can assume that the delay associated with the return from the `io_out()` function is small in relation to the delay of the execution of the function call itself [5]. Thus, t_{IO} can be treated as the `io_out()` function call time.

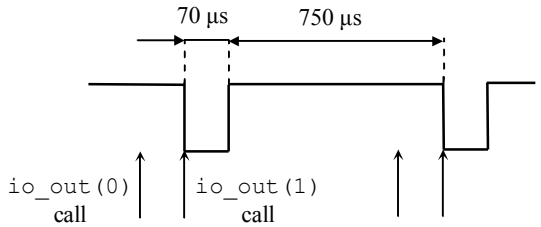

Fig. 3 The evaluation of `io_out()` function execution time.

4.4 Minimum Context Switching Delay

Since the execution time of `io_out()` function is known, we can evaluate the *minimum context switching delay* t_{SW}, which is the minimum time a scheduler needs to switch the execution of the consecutive tasks. It can be found as a difference between the width of the positive pulse, t_H, and the execution time of `io_out()`, t_{IO} (Fig. 3) :

$$t_{SW} = t_H - t_{IO} = 700\ \mu s - 70\ \mu s = 630\ \mu s.$$

Thus, the context switching delay is quite long in relation to the other Neuron C temporal components (t_{SCH}, t_{SW}, t_{IO}). It is because the end of a task defines the *critical section* in the Neuron C application program [6,7]. The operations executed at the end of a critical section are listed in Sect. 2.1. In particular, the outgoing messages and output network variable updates are sent, and the incoming messages and input network variable updates are processed.
Moreover, as follows from the algorithm specification, before execution of each task, the scheduler tests the system wide events (`online`, `offline`, `wink`), the events related to `priority when` clauses and non-priority `when` clauses.
The evaluated minimum switching delay t_{SW} is valid for switching between consecutive tasks except the transition from the last to the first task in the scheduler loop. The latter is lengthen, as we noticed, by scheduler restart overhead which is 50 μs long.

4.5 Context Switching with Active Network Variables

Now we will consider the other version of the application program where the network variable(s) state is modified within the task.

```
IO_2 output bit test_signal;
network output nv;
when(TRUE)
{
        nv=nv+1;
        io_out(test_signal,0);
        io_out(test_signal,1);
                                        }
```

The diagram of the waveform generated on the pin IO_2 of input/output port is similar to that demonstrated in Fig. 3, however, the duration of the positive pulses is significantly longer. The measurement on the corresponding output pin shows that the duration of positive pulse is extended to 3.05 ms. If we assume that the time spent for network variable modification is short, the increase of task execution time is caused due to sending the network variable update at the end of the task. Moreover, as observed, this delay increase is the same both for short (8-bit) and long (16-bit) network variables.
If two or more network variable updates are sent within a task, the corresponding delay increases by 0.9 ms per a network variable and equals 3.95 ms and 4.85 ms for two and three network variables, respectively.

4.6 Execution Time of `post_events()` Call

Using the code presented below we can estimate the execution time of the idle call of `post_events()` function (we define the function *idle call* as a call when neither network variable updates nor explicit messages are ready for sending) :

```
IO_2 output bit test_signal;

when(TRUE)
{
        io_out(test_signal,0);
        post_events();
        io_out(test_signal,1);
                                        }
```

The oscilloscope measurement gives the following results: the duration of the high t_H = 750 μs, and of the low logic

level t_L = 400 μs. Since the `io_out()` function call time is 70 μs (see Sect. 4.3), the execution time of the idle `post_events()` function call equals :

$$t_{\text{post\_e idle}} = 400\ \mu s - 70\ \mu s = 330\ \mu s.$$

Note that the code presented above can be used for evaluation of the execution time of *any* function that will be called between `io_out()` consecutive calls. For example, using the code :

```
IO_2 output bit test_signal;
when(TRUE)
{
        io_out(test_signal,0);
        watchdog_upate();
        io_out(test_signal,1);
}
```

we can evaluate the execution time of `watchdog_upate()` function call equals 30 μs. It is a difference between the negative pulse duration (100 μs) measured on the oscilloscope and the `io_out()` function call time (70 μs).

4.7 Bypass Mode Timing

Next, we investigated the minimum delay of scheduling in the bypass mode. At first, we have analyzed the following procedure:

```
IO_2 output bit test_signal;
when(TRUE)
{
        if(TRUE)// Task 1
   {
        io_out(test_signal,0);
        io_out(test_signal,1);
                                  }
        if(TRUE)// Task 2
   {
        io_out(test_signal,0);
        io_out(test_signal,1);
                                  }
        if(TRUE)// Task 3
   {
        io_out(test_signal,0);
        io_out(test_signal,1);
                                  }
                                       }
```

The pulse train generated on the pin IO_2, as a result of execution of the presented code, is shown in Fig. 4.

Note that although the scheduling presented above is made using `if` instruction in a user-defined order, it does not correspond to the bypass mode, since the program returns from the single `when` clause in each cycle. Therefore, the implicit event processing is done at the end of each task execution. As seen in Fig. 4, the execution time of the `if` statement is negligible. It is because the negative pulse width is near the same (the difference is about 2 μs) as in Fig. 3, where the `io_out()` function call is not associated with `if` statement.

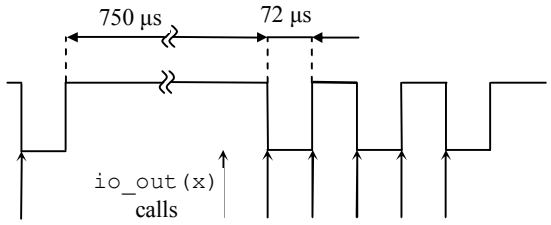

Fig. 4 The evaluation of conditional statement execution time.

Finally, we investigated the application program, where the "full" bypass mode is exploited so the watchdog-timer has to be updated. If network variables are used in the bypass mode, the `post_events()` function has to be called in order to define explicitly the boundary of critical section of the program.

Moreover, we simulate more complex "event" expressions in order to find its influence on timing of the program execution. Some algebraic and logic expressions with the variables a, b, c are used to complicate the evaluation of the events, however, all events still always evaluate to true.

```
IO_2 output bit test_signal;
int a=0;
int b=10;
int c=40;
when(TRUE)
{
   for(;;)
 {
        if(a==0)// Task 1
   {
        io_out(test_signal,0);
        io_out(test_signal,1);
                                  }
        if(b<=60)// Task 2
   {
        io_out(test_signal,0);
        io_out(test_signal,1);
                                  }

        if((c-b)>=25)&&(a=0)// Task 3
   {
        io_out(test_signal,0);
        io_out(test_signal,1);
                                  }
   watchdog_update();
                                       }
```

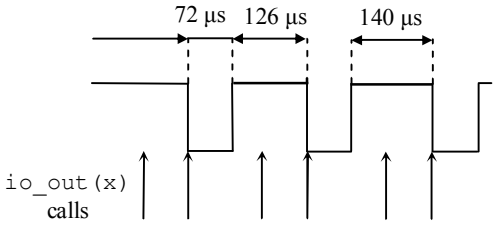

Fig. 5. The evaluation of bypass mode timing properties.

As seen in Fig. 5, testing of "event" expressions causes extensions of the positive pulses, because of finite latency of the evaluation of `when` clauses. The evaluation of the arithmetic expressions lengthens the first pulse (Task 1) by 56 µs and the second one (Task 2) by 70 µs.
All the measurements of the firmware task scheduler have been performed for the Neuron Chip with 10 MHz input clock. Note that the temporal measures are scaled with the input clock frequency, i.e. measured time intervals are directly proportional to the input clock.

4.8 Discussion

Summing up, the total overhead introduced by the Neuron Chip task scheduler is relatively high and approximately equal to 700 µs. The corresponding delay is significantly greater than the processing time of a task of average complexity. In particular, the scheduler-induced latency is an order of magnitude greater than the execution time of a function referring to the input/output port. The operations that are responsible for this latency are called automatically and are not present in the code. Therefore, the scheduler overhead can determine the application program timing if the program consists of a large number of short tasks associated with events which frequently evaluate to true.

6. Conclusion

The Neuron Chip built-in task scheduler defines the operational scenario of each LonWorks/EIA-709 device. The event-driven strategy of the scheduler algorithm is a part of the event-based architecture of the LonWorks networked systems. In most cases the knowledge of scheduler timing is not necessary to design the application program since the interarrival time between events is large enough in relation to the scheduler overhead. However, there are situations when the knowledge of Neuron Chip scheduler latency becomes important, e.g. if system is systematically extended and the number of tasks in the application program increases significantly. Then, the scheduler timing can present the bottleneck in real-time system behavior. It can happen especially if the application program consists of a large number of short tasks associated with events, which frequently evaluate to true. In this paper, some information that can support timing-aware Neuron Chip programming is provided.

References

[1] F. De Paoli and F. Tisato, "On the complementary nature of event-driven and time-driven models," Control Engineering Practice, vol.4, pp.847-854, 1996.
[2] H. Kopetz, Real-Time Systems. Design Principles for Distributed Embedded Applications, Kluwer Academic Publishers, 1997.
[3] D. Dietrich, D. Loy and H.-J. Schweinzer, Open Control Networks: LonWorks/EIA 709 Technology, Kluwer Academic Publishers, 2001.
[4] M. Miśkowicz, R. Golański, "LON technology in wireless sensor networking applications," Sensors, vol. 6, pp.30-48, 2006.
[5] LonWorks Technology Device Data, Rev. 4, Motorola Corp., 1997.
[6] Neuron C Programmer's Guide, Rev. 4, Echelon Corp, 1995.
[7] Neuron C Reference Guide, Rev. 4, Echelon Corp., 1995.
[8] NodeBuilder User's Guide, Rev. 3, Echelon Corp., 1995.
[9] H. Schweins and D. Heffernan, "Retrofitting a deterministic access control policy to a non-deterministic control network,", *Proceedings of Irish Signal and Systems Conference*, 1998.
[10] M. Miśkowicz, "The timing properties of the embedded Neuron Chip task scheduler," *Proceedings of IFAC Workshop on Programmable Devices and Systems*, *PDS 2004*, pp.326-331, 2004.

Marek Miśkowicz received his M.Sc. and Ph.D. degrees in Electronic Engineering respectively in 1995 and 2004 from AGH University of Science and Technology (AGH-UST), Krakow, Poland. Currently, he is an Assistant Professor at the Department of Electronics, AGH-UST. His main research interest have been focused on level-crossing sampling, asynchronous analog-to-digital conversion, and performance modeling of networked sensor and control systems. He has published more than 30 papers in international journals and conference proceedings.