Implementation of Event Driven Software Gates for Combinational Logic Networks

József Kopják*, János Kovács**
* Óbuda University/Kandó Kálmán Faculty of Electrical Engineering, Budapest, Hungary
** Széchenyi István University/Faculty of Engineering Sciences, Győr, Hungary
kopjaj.jozsef@kvk.unibuda.hu, kovacs@sze.hu

Abstract—The paper presents one implementation possibility of event driven software model of distributed or classical combination logic networks. The paper first reviews the basics of event-driven systems. Greater part of article presents the mathematical basics and architecture of event-driven combinational logic and example software implementation of the model. The paper shows the software implementation of AND and OR gate. After the introduction of basic gates of combinational logic the paper introduces the concept of the universal event-driven gate. The paper ends with software implementation of symmetrical event-driven XOR gate to create odd and even functions.

I. INTRODUCTION

Discrete logical control systems have two subfamilies: combinational and sequential logical system. Traditional way to implement combinational logical system is using Sequential programming model. Sequential programming model are used in industrial solutions usually. Sequential programming model is named linear programming model too.[6]

Event driven program model is a very good tool to write discrete control programs based on sequential logic. The control objects have own state descriptors. The software engineer have to examine first the possible states of controlled system, after find the transaction possibilities from one state to another. To create combinational logic based controls is problematic. Engineers have to design their program in if-then conditions in event-driven program model software. To model combinational logic with if-then condition is not so easy. This article present one implementation of modeling of combinational logic based control networks using event-driven program model.

II. EVENT-DRIVEN PROGRAMMING MODEL

This section contains statements formulated by the [5] literature too. Event-driven systems require a distinctly different way of thinking than traditional sequential programs[7]. The event-driver program does not wait for an external event, the program react to events [4]. The program must be designed in if-then conditions, for example: if the set pin of set-reset flip-flop change to true, then the output of flip-flop must be set to true; if the reset pin of set-reset flip-flop change to true, then the output of flip-flop must be set to false. The event driven model based programs can be work as parallel systems. The event driven system can be works as data-flow model base systems. The data flow computation model enables us to use the program’s natural parallelism, which, in turn, shortens the time required to perform a calculation [1].

Before analyzing the event-driven programming model let's look at the concept of the events. The events are signals from the outside of world in generally. Events are describing dynamic changes of inputs. Events can be external – hardware – events such as a button press or a tank becomes full or internal – software – events such as time expired, result of mathematical calculation, changing state of software resources.

The objects are informed about events through messages. The message in complex, distributed system can be objects too, such as a communications packet (MODBUS message or TCP / IP packet). In single-processor systems we use a simple function calls to transmit the messages.

In event-driven model based program contains active and passive objects too. The passive objects are running only when they receive some messages. The passive objects can generate messages only, when they have received some message too. The active objects are running always. The active objects can generate messages without receiving messages. The active objects are usually hardware supported interrupt service routines. The processors interrupt periphery calls the interrupt service routines in case of external or internal interrupt event. Hardware interrupt can be a change of the value of input pin (Interrupt on Change), incoming data on communication port or end of data transmission. Internal hardware event can be a timer overflow or end of analog-digital conversion. The interrupt service routines create event-telegrams (messages), which are posted to event queues by ISRs.

The advantage of program based on traditional event-driven model is that the model includes the automatic control of processor power consumption. If the message comes when the event queue is empty the response time of program is the message processing plus the answer generation time.

III. CONSTRUCTION OF EVENT DRIVEN COMBINATIONAL LOGIC NETWORKS

This section contains statements formulated by the [5] and [6] literature too Some distributed systems are not easy to control with programs based on traditional control models. Wireless or wired automated productions systems or wireless home automation solutions can be implemented with mash network [8]. This type of network can contains not only active members but lot of passive end device sensors too. The passive end device sensors almost are in sleeping mode; they wake up only when they must to send some message. Example network for using passive devices can be a battery powered distributed sensor network.[2][8] To control theses system are not so
easy with traditional program models; good solution to control they with event driven models.

Our presented event driven combinational logic model is based on active object computing model. The active object computing model addresses most problems of the traditional event-driven architecture, retaining its good characteristics.[7] The model combines the benefits of the pre-emptive multitasking operating systems and event-driven scheme.

The active objects are individual task functions with own event queues. Each task starts running with initialization after tries to get event message from its event-queue. In case the event-queue do not contain any messages, then the tasks loses its running state and go to blocking (waiting) state. Only newly received event message can change back task state to "ready to run" state. If task successfully gets message from event-queue the task processes the received message. Figure 1. shows the simplified flowchart of task functions.

The objects are connected together with event message queues. Some tasks responsibility to generates messages to the queues. These tasks are inputs handling tasks. The control tasks gets and generates message too. The outputs handling tasks read they own message queues and put the information to the hardware outputs.

Messages used by this model are composed from two parts:

Type of message – The message type can be initialization or change notification message. Initialization messages: The objects are sending initialization messages as first message. The object can send initialization message only once. Change notification messages: The objects are sending change notification messages when they output value has been changed compared to the last sent message.

Value of message – The value of message can be true or false. The true means the object's output has been changed to true. The false means the object's output had been changed to false.

Following piece of source code illustrates the event structure in C language.

```c
typedef struct {
  BOOL bValue;
  BOOL bInitValue;
} BOOL_EVENT_TYPE;
```

IV. FINITE STATE MACHINE MODEL BASED SOFTWARE AND GATE

In discrete logical system the values of inputs and outputs can be only one of two possible states at any given time. [3] [11] In discrete logical system are only two binary logic operation that underlies all of logic design and Boolean algebra. These are the AND and OR operation. Next table represents the truth table of AND operation. The zero values represent logical false, the one values represent logical true.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The AND gate realize the AND operation, therefore AND gate has same truth table like AND operation. When we look at the AND gate's truth table we can say some conclusions. An AND gate's output is false(0) if and only if any of its inputs is false(0). [10]. The reverse of the statement is also true. An AND gate's output is true(1) if and only if all of its inputs are true(1). From the second statement follows the AND gate's output can be only in one state true, when the all inputs of gate are true, otherwise the gate's output is false.

If we would like to define event driven operation of AND gate we have to create new behavior descriptor statements. First we need an internal state descriptor. We need to define the distance (d) from AND gate's true output value. The value of distance variable is natural number (d ∈ N). The distance represents the numbers of inputs which need to be changed to true therefore the AND gate's out can be changed to true. The maximal value of distance equal with number of gate's inputs. In summary: If the number of AND gate's inputs equal with n, then the 0 ≤ d ≤ n, where the n,d ∈ N.

Using the distance variable we can newly define the behavior of AND gate:

1. An AND gate's output is false if and only if distance number is bigger then zero.
2. An AND gate's output is true if and only if distance number equal with zero.

The definition shows that the AND gate do not need to now its number of inputs, only the number of false inputs to decide its output value. With distance the gate can be modeled with final state machine as well. Event driven SOFTWARE gate built up form distance counter algorithm, one input message queue and connection possibility to output message queue. Figure 2. represent the structure of event driven software AND gate.

V. IMPLEMENTATION OF EVENT DRIVEN SOFTWARE AND GATE

The fine state model based software AND gate is passive object point of view of behavior. The gate can
generate message to the output message queue only when the
gate has been received message from the input
message queue.

The gate has two internal state descriptors: current
distance and last distance.

After reset, the AND gate first initialize itself. The gate
set its distance value to zero, and last distance value to
minus one. The minus one means there is no previous
distance value. After initialization the gate tries to receive
message from input message queue. If the input queue is
empty the gate goes to blocked state.

If the gate receives message from input queue the gate
examines the content of the message. If the value of the
received message is false, then increment the value of the
distance. The messages with false value are dominant
messages. The dominant message mans the gate increment
its distance value without any conditions. If the value of
the received message is true and the message is not
initialization message, then decrement the value of the
distance.

After processing the message the gate examines that the
previous distance value is different from the current value
or not. If the two values do not match, and the previous or
the current distance is less than or equal to zero, then the
gate sends message to the output message queue. If the
current distance value is zero, then the content of sent
message is true, otherwise is false. If the previous distance
value is less than zero, then the gate sends initialization
message, otherwise change indicator message. After
sending the message the gate goes to blocking state. The
flowchart int the Figure 3. shows one example algorithm
of event driven software AND gate.

The following code illustrates implementation of event
driven software AND gate in C language.

```c
void xGateTask( void *pvParameters ) {
    GATE_PARAMETER xGateParameter= *(GATE_PARAMETER*)pvParameters;
    BOOL_EVENT_TYPE xReceivedEvent, xEventToSend;
    /* Counter count the incoming "dominant" values */
    // Initialization value of counters
    BASE_TYPE xDistance = 0, xLastDistance = -1;
    for( ;; ) {
        /* Receive Event from MessageQueue */
        vReceiveEventMsg( &xGateParameter, &xReceivedEvent );
        /* Evaluating receaved message*/
        if( xReceivedEvent.bValue == true ) {
            xDistance++;  // Increasing value of distance
        } else {
            if ( xReceivedEvent.bIsInitValue == false ) {
                xDistance--; // Decreasing value of distance
            } else {
                break;
            }
        }
        /* Calculating gate output*/
        if( xDistance == xLastDistance ) {
            // If value of distance has changed */
            if( xDistance == 0 ) {
                xEventToSend.bValue = (xDistance == 0) ? true : false;
                xEventToSend.bIsInitValue = (xLastDistance < 0)?true:false;
                /* Send Event to MessageQueue */
                vSendEventMsg( &xGateParameter, &xEventToSend );
            }
            // Last distance equal with current distance value
            xLastDistance = xDistance;
        }
    }
}
```

Figure 3. Flowchart: algorithm of event driven software AND gate.

VI. FINITE STATE MACHINE MODEL BASED SOFTWARE
OR GATE

This chapter deals with finite state machine model
based software OR gate. Next table represents the truth
table of OR operation. The zero values represent logical
false, the one values represent logical true.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The OR gate realize the OR operation, therefore OR
gate has same truth table like OR operation. When we
look at the OR gate's truth table we can say some
conclusions. An OR gate's output is true(0) if and only if
any of its inputs is true(0). [10]. The reverse of the statement is also true. An OR gate's output is false(1) if and only if all of its inputs are false(1). From the second statement follows the OR gate's output can be only in one state false, when the all inputs of gate are false; otherwise the gate's output is true.

If we would like to define event driven operation of OR gate we have to create distance descriptor statements like we made at AND gate too. In case of OR gate the distance represents the numbers of inputs which need to be changed to false therefore the OR gate's output can be changed to false. The maximal value of distance equal with number of gate's inputs like at AND gate. In summary: If the number of OR gate's inputs equal with n, then the \(0 \leq d \leq n\), where the \(n,d \in \mathbb{N}\).

Using the distance variable we can newly define the behavior of OR gate:

1. An OR gate's output is true if and only if distance number is bigger then zero.
2. An OR gate's output is false if and only if distance number equal with zero.

![Figure 4. Structure of event driven software OR gate](image)

The definition shows that the OR gate do not need to now its number of inputs, only the number of true inputs to decide its output value. With distance the gate can be modeled with final state machine as well. Event driven OR software gate built up form distance counter algorithm, one input message queue and connection possibility to output message queue. Figure 4. represent the structure of event driven software OR gate.

The fine state model based software OR gate is passive object point of view of behavior like AND gate. The internal state descriptors are same like at AND gate – current distance and last distance.

The first difference between AND and OR gate is the meaning of dominant message. As opposed to the AND gate, the OR gate increments its distance value at true messages. – The messages with true value are dominant messages. – If the value of the received message is false and the message is not initialization message, then the OR gate decrement the distance value.

The second difference between AND and OR gate is the output message generation in the function of the value of the distance. In case of OR gate, if the current distance value is zero, then the value of sent message is false, otherwise is true. The flowchart in the figure shows one example algorithm of event driven software OR gate. Differences between Figure 3. and Figure 5. are highlighted.

VII. EVENT DRIVEN UNIVERSAL SOFTWARE GATE

At study of implementation event driven AND and OR software gate we may notice that the implementation of two type of gates are very similar. The AND gate increments its distance value at incoming false messages, the OR gate increments its distance value at true messages. When the gate's distance value change to zero the AND gate sends true message to the its output message queue, the OR gate sends false message. If the gate's distance value changed from zero, then the AND gate sends false message, the OR gate send true message to the output message queue. The other parts of gate implementation are same. In summary: If we would like to crate from AND gate an OR gate, or from OR gate an AND gate we have to invert values gate's input messages and output messages too. The inversion of messages means, the messages with false value would change to true, the messages with true value would changes to false.

![Figure 5. Flowchart: Algorithm of event driven software OR gate](image)

The universal gate – which can work like as AND or OR gate in function of a gate's configuration parameters – can build from two configurable inverter and with one distance counter unit. The configurable inverter means, the inverter can pass through or invert the value of messages. The distance counter counts the incoming dominant messages. The value of dominant message is true, it means the gate's distance counter would increment by one at incoming true messages. When the value of distance changes to zero, the distance counter unit would
send message with true value. If the value of distance changes from zero, then the counter sends message with false value. Figure 6. represents the structure of event driven universal software gate.

![Universal Event Driven Software Gate]

Figure 6. Structure of universal event driven software gate.

The universal event driven software model can behave like an NAND (Not-AND) and NOR (Not-OR) gate too. Next table represents the input and output inverter configuration table of universal gate. The table's first column shows the gate type of behavior, the second column shows the state of input message inverter, and the third column shows the state of output message inverter.

<table>
<thead>
<tr>
<th>Gate</th>
<th>Input Inverter</th>
<th>Output Inverter</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>Inverting</td>
<td>Inverting</td>
</tr>
<tr>
<td>OR</td>
<td>Pass Through</td>
<td>Pass Through</td>
</tr>
<tr>
<td>NAND</td>
<td>Inverting</td>
<td>Inverting</td>
</tr>
<tr>
<td>NOR</td>
<td>Pass Through</td>
<td>Pass Through</td>
</tr>
</tbody>
</table>

VIII. FINITE STATE MACHINE MODEL BASED SOFTWARE XOR GATE

This chapter deals with finite state machine model based software XOR (eXclusive-OR) gate. Next table represents the truth table of XOR operation. The zero values represent logical false, the one values represent logical true.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

The XOR gate realizes the exclusive-or operation, therefore XOR gate has same truth table like XOR operation. LOT of time the exclusive-or operation is marked with ⊕ symbol. The XOR operation is not a basic operator in Boole algebra, it is means we can describe the XOR operation with AND, OR and Negation operators.

\[ A \oplus B = \overline{AB} + \overline{A}B \]

The original XOR operator is defined only for two inputs. If we would like to create XOR gate with more than two inputs, then we have to define the ODD function. XOR function output true(1) if and only if the number of true(1)’s in the input combination is odd.

![Disjunctive Karnaugh Map of ODD Function]

Figure 7. represents the disjunctive Karnaugh map of ODD function with four inputs.

\[ Output = A \oplus B \oplus C \oplus D \]

The minimization of ODD function with classical methods is impossible, because there are no adjacent terms in same logical level. Around the trues the adjacent terms are false; around the falsies the adjacent terms are true.

To implement event-driven XOR gate (ODD function) is easier to create new gate from scratch rather than to assemble gate from event-driven universal gates.

If we would like to define event driven operation of XOR gate we have to create distance descriptor statements like we made at AND, OR and universal gate too. If the gate receives true message, the gate increments its own distance value; if the gate receives false change notification message, the gate decrements the distance value. Figure 8. represents the value of variable in four input based gate. The figure represents the meaning of distance value too; how many adjacent steps we need to make to get back to base state – to the zero state. If we analyze the map we can conclude that, the event-driven XOR gate has to send true output message when the value of distance variable becomes odd and has to send false output message when the value of distance variable becomes to even.

Using the distance variable we can define the behavior of XOR gate:

1. An XOR gate's output is true if and only if distance number is odd.
2. An XOR gate's output is false if and only if distance number equal with even.

The definition shows that the XOR gate do not need to now its number of inputs – like at AND, OR and Universal Gate too –, only the number of true’s to decide its output value. With distance the gate can be modeled
with final state machine as well. Event driven XOR software gate built up form distance counter algorithm, one input message queue, configurable inverter to implement odd and even functions – to implement XOR, XNOR gate – and connection possibility to output message queue. Figure 9 represents the structure of event driven software XOR gate.

![Event Driven Software XOR Gate](image)

Figure 9. Structure of event driven XOR software gate.

The implementation of XOR gate is very similar to universal gate. The difference of implementation between XOR gate and universal gate is the distance zero test. The universal gate changes its output when the distance value is changes to or changes from zero. The XOR gate changes its output after every received change notification message because the output of XOR gate is parity of distance variable. The flowchart in the Figure 10. shows one example algorithm of event driven software XOR gate.

![Flowchart: Algorithm of event driven software XOR gate](image)

Figure 10. Flowchart: Algorithm of event driven software XOR gate.

**IX. SUMMARY**

With programming model presented in this paper is possible to create event-driven combinational logic based control networks. The main advantage of model is the software gates do not need to know the values of all input at given time moment to calculate they output values. The software model could be a good solution for controlling wireless distributed sensor networks. Using presented model the sensors and control unit do not need all time communication connection, enough if the sensors send initialization a change notification messages to control unit.

**REFERENCES**

[4] Dr. Kondorosi Károly, Dr. László Zoltán, Dr. Szirmay-Kalos László, Objektum-Orientált Szoftverfejlesztés, 2003