Real-time Program Models Used in Tiny Embedded Systems

J. Kopják[1], J. Kovács[2]

[1]Óbuda University/Kandó Kálmán Faculty of Electrical Engineering, Budapest
[2]Széchenyi István University/Faculty of Engineering Sciences, Győr
e-mail: kopjak.jozsef@kvk.uni-obuda.hu, kovacs@sze.hu

Abstract: The paper present and compare some real-time control program models used by programmers programming in tiny embedded systems, microcontrollers. The paper does not demonstrate all programming models, only the most important variations based on writer's opinion. The writer tried to show the advantages and disadvantages of each programming models discussed in article. The paper begins with traditional models like sequential model, cooperative multitasking and continues with an alternative cooperative multitasking method, timed cooperative model and finish with event-driven model.

Keywords: cooperative multitasking, timed cooperative model, event-driven model, real-time

1. Introduction

The improvement of microcontrollers using in embedded systems is very strong. Some performances of microcontrollers are on same level like in personal computers. The improvement of eight and sixteen bit controllers does not stop too. The controllers have more and more peripheries. The tasks of microcontrollers do not stop on control functions. Microcontrollers with cheep price have small cpu core with small ram size and with rich number of peripheries. With rich number of peripheries is possible to satiate user demands like LCD control, sensor-less BLDC fan motor control, lots of type of serial communication with eight bit controllers too. To resolve these new problems programmers needs to have new program modelling knowledge.

2. Real-time systems

Non real-time system are usually using in office technologies. In non real-time systems incoming events followed each other in time in a same priority level are not served in a queue followed each other. In these systems are known the average of processing time of events, but we do not know the maximum time of answer of occurred event.

Real-time control is very important in solutions using in industrial and medical environment. Every event has own priority in real-time system. Consecutively becoming events in a same priority level are served in one after the other. The maximum of answer time of incoming event is known and defined.

A critical aspect of real-time systems is how time itself is handled. The design of a real-time system must identify the timing requirements of the system and ensure that the system performance is both correct and timely. The three types of time constraints on computation are: hard, soft, firm [1].

3. Sequential programming model

Sequential programming models are used in industrial solutions usually. Sequential programming model is named linear programming model too. These programs are composed from four software component: initialization, reading status of the inputs, calculation or processing, refreshing values of the outputs. Figure 1 illustrates the simplified flow chart of program based on sequential programming model.

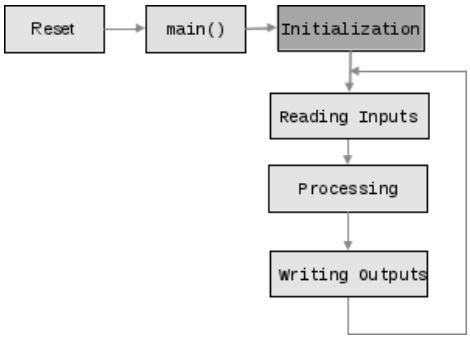

Figure 1: Simplified flow chart of program based on sequential programming model

The advantage of sequential programming model is the design of program is very similar to classical logical circuit design method. Simultaneously (in time) reading of all inputs and writing of all outputs exclude the hazard known in logical circuits.

The disadvantage of the model is the source code contains lots of if-else control structures and therefore is incomprehensible case of writing complex programs. Other disadvantage of the model is the poor utilization of processor. The program always recalculate the values of outputs, regardless of there is any change on inputs. The program based on this model is always running, therefore uses all CPU time. The reaction time of program depends on the program complexity due to constant recalculations.

4. Cooperative multitasking

In cooperative multitasking the tasks are working together with each other. The tasks are simple functions without input parameters and return value. The main program first initializes the peripheral devices, after the initialization the program calls the task functions in loop. Figure 2 illustrate the simplified flow chart of program based on multitask programming model.

Source code written in cooperative multitasking model is more transparent than the codes written in sequential method. The tasks can be easily separated, each task have different job. These tasks are in different functions, and these functions are located in different source files usually. If the task-functions are located in different source files

the program developers can be work together easily, because they do not interfere with each others work. Each developer has to modify only their own files, and don't need to change content of files created by other developers.

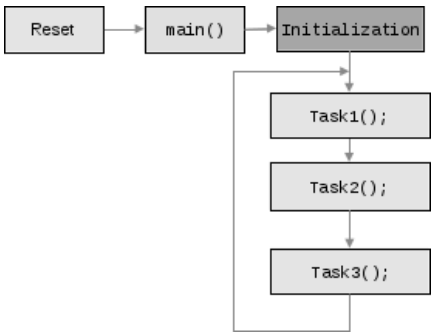

Figure 2: Simplified flow chart of program based on cooperative multitask programming model

Following piece of C-language source code represents the main loop responsible for calling the task functions.

```
int main ( void )
{
  vInitSystem();// Initialization of peripherals
  while(1)      // Infinitive function calling loop
  {
    vTask1();  // Calling task function named vTask1
    vTask2();  // Calling task function named vTask2
    vTask3();  // Calling task function named vTask3
  }
}
```

Developing source code in cooperative multitasking model required more software planning time than sequential solution. The designer has to divide the whole program-task to small individual working tasks in a first phase of development. Tasks are communicating with each other on global variables or with another name: via mailboxes. The cooperating task functions usually read its inputs values at the beginning and writing they output values at end of each running sequence. The program scans its inputs and updates its outputs values several times during a one circle in the main loop.

The task should complete own running sequence as soon as possible to got the shortest program response time. The tasks should not use blocking or waiting loops. Using blocking loops would drastic incise the response time of running system. Some blocking loops examples: Waiting for releasing a push button or waiting for an UART peripheral register flag. The tasks have to remember the current state of running and return back to the calling function in case when a resource is not available.

The task functions should be structured using state machine model due to lack of a blocking waits. In state machine model the functions are using static local variables to store them next states. The backbone of each task function is a switch-case structure. The task first examines the value of the state description variable, and then jumps to the program code of the state. Following piece of C-language source code illustrates the body of task function based on state machine model.

```
void vTask( void )
{
  /* State description variable. Initial state is:
     INIT_STATE */

  static enum { INIT_STATE, STATE1, STATE2, STATE3 }
  xState = INIT_STATE;

  /* Selection of sub-task based on the value of state
     variable */
  switch ( xState )
  {
    case INIT_STATE:  // Initialization tasks
         /* ... */
         xState = STATE1; // Selection of next state
         break;           // End of actual running

    case STATE1:        // First task
         /* ... */
         xState = STATE2; // Selection of next state
         break;           // End of actual running
    /* ... */
    default:         // Protection state:
         // Normally the program never gets here.
         xState = INIT_STATE; // Re-initialization of
                              // task-function
  }
}
```

The advantage of cooperative multitasking model is the possibility of creation quasi-parallel multitask system without using pre-emptive multitask operation system. The disadvantage of cooperative multitasking model is the pretension of good developer programming skills. The response time of program is depends on the tasks execution time. The execution time of task is not a constant value, because every task state has other execution time. The response time of program will be swinging, but is possible to calculate the fastest and slowest reaction time of system. The shortest reaction time of system is the sum of fastest running time of tasks, and the longest response is the sum of slowest running time of tasks.

Another disadvantage of the cooperative multitasking is that the program is not able to manage energy consumption of the processor. The processor runs the code always, it can not go to energy-efficient mode (sleep or idle), this means the processor will be have maximum energy consumption.

5. Timed cooperative multitasking

The traditional cooperative multitasking model do not deal with time, it means if somebody would like to create timed functions or events then the programmer must to create own timer to measure the elapsed time. In timed cooperative multitasking model the tasks do not run always like in classical cooperative multitasking model, only when they blocking or waiting time has expired.

If somebody would like to implement timed cooperative multitasking model, then he must create own lightweight kernel to manage the task functions. There is some known implementation of cooperative multitasking, with other name co-rutins, like FreeRTOS. FreeRTOS is an open source operating system for embedded systems written in C language.[2] The implemented kernels working with time, but the time is not the basic element of they kernel model, therefore we design a own lightweight cooperative multitask kernel.

To demonstrate the difference between classical cooperative multitasking and the timed cooperative multitasking we made an example program. In our program are two

tasks. The tasks are running quasi-parallel. The mission of each task is very simple: Invert value of one output pin. Next code example written is C language shows the implementation of these tasks.

```
/* Task function of task #1*/
void vTask1 ( void )
{
    mTurnOnLED( 3 );          // The task starts here
    mToggleLED( 1 );          // Toggle LED #1
    mTurnOffLED( 3 );         // The task stops here
}

/* Task function of task #2*/
void vTask2 ( void )
{
    mTurnOnLED( 0 );          // The task starts here
    mToggleLED( 2 );          // Toggle LED #2
    mTurnOffLED( 0 );         // The task stops here
}
```

Our initial kernel is very simple. Next code example shows our initial, not timed kernel.

```
/* endless loop */
while (1)
{
          vTask1(); // Call Task1 function
          vTask2(); // Call Task2 function
}
```

The Figure 3 illustrate the result of program running is simulator. In picture you can find 4 lines. The line with label “Task #1” is in high level when the task #1 is running, and is in low level when the task is in “not running” state. The line with label “Task #2” demonstrates the actual state of the task #2 (the conditions of high and low level is similar like in previously described line). Line with label “PIN #1” and line with label “PIN #2” shows the output of Task #1 and Task #2 (toggle LED on output pin). We can see in the figure that the both tasks are running always in circa half of the processors time.

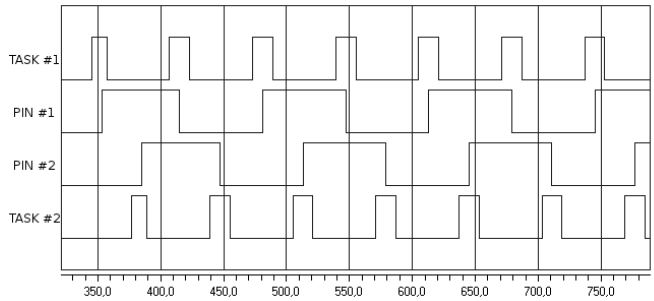

Figure 3: Result of running program write in traditional cooperative multitask model

To create a timed task calling we need one global timer, which measure the continuous global time in tick. Operating system usually measure the time in tick. The tick counter is 16 or 32 bit long unsigned variable, but it can be an 8 bit counter in very small systems. We are using 16 bit wide architecture to test our kernel; therefore our tick counter is 16 wide too. In this example the timer increase the value of tick counter in every one millisecond.

We need an array list too. The array list contains the task descriptors. One element of array is a structure, which represents one task. The descriptor structure has three fields:

the pointer of task function, the recall interval in tick and the time stamp of last call. Next code example shows the declaration of task descriptor structure.

```
/* Type define: Task Entry structure */
typedef struct
{
  void (*pvTaskPointer) ( void );//Pointer of task func.
  TICK_TYPE tickRecallInterval;  //Recall interval
  TICK_TYPE tickLastTimeStamp; //Time stamp of last call
} TASK_ENTRY;
```

Next piece of code shows the description list of tasks. The list must be closed with predefined closing entry where the value of function pointer is NULL. When the array is closed with closing entry, the programmer do not need to predefine the size of array list, because the parser part of the program could determinate the size of given array.

```
static TASK_ENTRY xTaskList[] =
{
    {vTask1, 3, 0},
    {vTask2, 6, 0},
    {NULL, 0, 0}  // End of Task List (closing entry)
};
```

The kernel first calculates the difference of current time and the time stamp of last call, and when this value is greater than or equal to tasks recall time then calls the task function. Next piece of code shows the implementation of kernel.

```
/* Local variables */
TICK_TYPE tickCurrentTime;
HTASK_ENTRY hTask;
UBASE_TYPE uxTaskIndex = 0;

/*  endless loop*/
while (1)
{
  hTask = &xTaskList[uxTaskIndex++];//Get handle of task
  if( hTask->pvTaskPointer != NULL )
  { /* If task entry is not closing entry: */
    tickCurrentTime = tickGetCurrentTime();
    if ( (tickCurrentTime - hTask->tickLastTimeStamp)
          >= hTask->tickRecallInterval )
    {/* If task time was expired */
      hTask->pvTaskPointer(); // Call task function
      hTask->tickLastTimeStamp = tickCurrentTime;
    }
  }
  else
  {
    uxTaskIndex = 0;  // Restart loop counter
  }
}
```

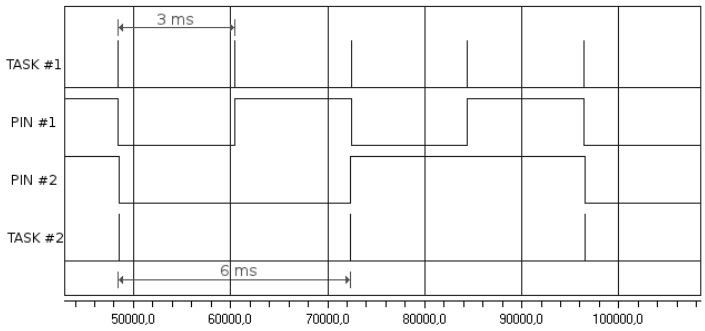

Figure 4: Result of timed cooperated multitasking

The Figure 4 illustrates the result of program running in simulator. In picture you can find 4 lines. The meanings of the lines are same like in Figure 3. You can see in the figure, that the both tasks are running only when they blocking time has expired.

6. Event-driven programming model

Before analyzing the event-driven programming model let's look at the concept of the event. The event is an occurrence (incidence), which may change the status an object [3]. 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 queue is full or semaphore was taken.

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 are using a simple function calls to transmit the messages.

In traditional event-driven model based programs are only passive objects. The objects recalculates they output values do the effect of change they input values. The event-driven program runs only when the input values are changing. The response time of program depend on the number of changing input values in same time, and do not depend on the controlled network complexity.

The traditional event-driven system consists the following actors:

Interrupt service routines (ISR) – 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, which are posted to event queues by ISRs.

Event-telegraphs – The event-data structures consist of a minimum of two parts: The signal of the event conveys the type of the occurrence, and event parameters convey the quantitative information about the occurrence [4]. Following piece of C-language source code illustrates example for event-telegraph structure.

```
typedef struct EventTag
{
  unsigned int uiEventID;  // Event ID
  unsigned int uiParam;    // Event parameter
} Event;
```

Event queue – Event queue is a list (first in first out type list) of event telegraphs. The interrupt service routines put the event telegraphs to the end of queue and the event handler loop gets events from queue. Care should be taken when implementing the event queue because is shared resource. The event telegraphs arrive to the queue asynchronously. All operation with event queue should be made in critical section; exclude multiple writes, or read and write, in queue in same time.

Event message processing loop – The event message processing loop located in the main program. The event message processing loop is an infinite loop. The loop first tries to get message for event queue. If there is a waiting message in the queue, then the

loop process it based on message ID and message parameters. If there is no message waiting to be processed, then the loop puts the processor to the power saving mode. The interrupts will be wake up the processor from power-saving mode.

Not only the interrupt service routines can put events to the queue, but the event processing loop too. It can be happens the actual event processing ends with generation of new event. In such cases, the generated event is added to the end of the queue of events. Figure 5 illustrates the traditional event-driven program structure.

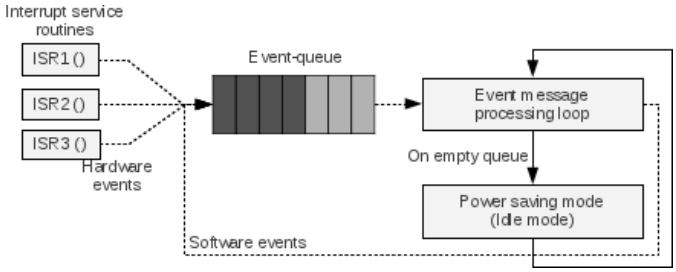

Figure 5: Traditional event-driven program structure

Event-driven systems require a distinctly different way of thinking than traditional sequential programs [4]. The event-driver program do not wait for an external event, the program react to events [5]. 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.[6]

The structure of event processing loop is similar to the functions developed basis on state-machine model. The core of loop is an switch-case structure. The switch tag jumps to the selected function description based on incoming event ID. Following piece of C-language sample code illustrates the body of event processing loop.

```
while(1)
{
  /* Getting event from event queue */
  if( xQueueReceive( xEventQueue, &xReceivedEvent, 0 )
      == pdPASS )
  { // On successful reception:
    /* Jump to selected task based on  message ID */
    switch ( xReceivedEvent.uiEventID )
    {
      case BUTTON:   // The button has bean sent
           /* ... */ // the message
           break;       // End of message processing

      case TIMER:       // The timer has bean sent
           /* ... */ // the message
           break;       // End of message processing
    }
    else  // There is no event waiting to be processed:
    {
      Idle(); // The processor goes to power saving mode
    }
}
```

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. The disadvantage is that the program can not rank events based on their importance and is not suitable to create modular program.

7. Conclusions

All programming models presented here have their own reason for existence. For smaller tasks may be useful to use design models presented at the beginning of the article, for complex systems are better solutions described in the second half of the article. Complex models are using more processor for administrative tasks instead of real problem, thereby reducing the effectiveness of the processor. On battery powered solutions should be used solution where the models automatically manage the processors energy consumption.

References

[1] Bruce Powel Douglass: *Real-Time UML: developing efficient objects for embedded systems*, 1998.
[2] Richard Barry: *Using The FreeRTOS Real Time Kernel*, 2009.
[3] Angster Erzsébet: *Az Objektumorientált tervezés és programozás alapjai*, 1997.
[4] Miro Samek: *Practical UML Statecharts in C/C++*, Second Edition, 2009.
[5] Dr. Kondorosi Károly, Dr. László Zoltán, Dr. Szirmay-Kalos László: *Objektum-Orientált Szoftverfejlesztés*, 2003.
[6] József Kopják, Dr. János Kovács: *Event-driven control program models running on embedded systems*, 2011.