Designing a New Real-Time Kernel with a Hybrid Scheduler

Qiang Huang, XiaoFeng Liang, WeiHua Xu
College of Information (Software) Engineering, Shenzhen University P.R. China 518060
lxf_szu@yahoo.com.cn

Abstract

Traditional embedded systems employ a time-triggered scheduler or an event-triggered scheduler, however, the truth is that time-triggered scheduling architecture is more dependable but lack of responsiveness to external events and event-triggered scheduling architecture is responsive. In order to get those advantages at the same time, a real-time kernel with a hybrid scheduler, named SinewOS, is developed. In SinewOS time-triggered scheduling and event-triggered scheduling are employed and in order to make this kernel more predictable and high efficient, then cooperative and preemptive scheduling approaches are adopted.

The features and the implementation of SinewOS are described in detail. This new kernel has been successfully implemented on LPC2129. And to evaluate the performance of an embedded system employing SinewOS, a testing case is carried out with a simulation, which is a hardware-in-the-loop simulator and based on an adaptive cruise control system (ACCS) platform.

1. Introduction

Reliable embedded systems play an increasing role in modern life, especially in modern automotive designs. Many studies have sought to develop reliable and dependable embedded software architectures or make improvements on the existing software architecture. For example, Pont have proposed a time-triggered cooperative (TTC) architecture which is implemented with a hardware timer [1], and in order to reduce jitter, Teera Phatrapornnant and Pont argue that it is effective to reduce jitter in an embedded system employing a time-triggered software architecture and dynamic voltage scaling [2]. However, the software architectures with which many embedded systems are implemented to date still have their disadvantages. For example, the TTC architecture is lack of fast responsiveness to external events, while a system employing ETP architecture can’t exactly carry out tasks at regular or particular instants of time, as a result task jitter occurs. To improve the reliability, safety and responsiveness of embedded systems, based on the research made by Pont, a new real-time kernel SinewOS is designed. The most important characteristic of SinewOS is that it is not a pure TTC architecture but is incorporated TTC, event-triggered architectures and features of preemptive scheduling. The implementation will be presented in this paper.

The paper is organized as follows: In section 2, the reasons for developing such a new real-time kernel are presented. And in order to make the implementation of SinewOS easier to be understood we describe two variables and three chains, which help to implement the scheduler, in section 3. We then go on to explain the implementation of scheduler in section 4 and 5. Section 6 explores how to handle interrupts under SinewOS. And In section 7 we build an evaluation platform based on an adaptive cruise control system (ACCS), and then conduct a series of testing based on this platform.

2. Reasons for developing a hybrid scheduler

2.1. Advantages of cooperative scheduler

Contrary to preemptive schedulers, cooperative schedulers have a number of desirable features, for example, Nissanke notes: other advantages of [cooperative] algorithms include their better understandability, greater predictability, ease of testing and their inherent capability for guaranteeing exclusive access to any shared resource or data [3]. Similarly, Pont notes: cooperative scheduling is simple, predictable, reliable and safe [1].

2.2. Time-triggered and event-triggered architecture

Various studies have demonstrated that, compared to event-triggered architecture, time-triggered architecture is more dependable and can provide a deterministic starting time for task, which can help to produce a system with low levels of task jitter. And as
Scarlett has noted, safety-critical systems demand dependability and as a result, time-triggered protocols have been used to date [4]. However, event-triggered architecture is more responsive. To design a system with high responsiveness and low levels of task jitter, both of them are implemented in SinewOS.

### 2.3. A new hybrid scheduler implemented in SinewOS

In terms of software architecture desktop OS and many embedded systems adopt a form of super loop. With such architecture many embedded systems (such as applications employing uC/OS-II) try to carry out tasks at particular instants of time in a way which is illustrated in Listing 1. But it is difficult to satisfy this requirement exactly. Pont states: this type of architecture [super loop] is suitable only for use in a restricted range of very simple applications, in particular those where accurate timing is not a key requirement [1]. A better solution is to employ a time-triggered cooperative (TTC) scheduling. As Teera Phatrpornnant and Pont have noted, TTC designs are particularly attractive. These static schedulers can, if used appropriately, help to produce systems with highly predictable patterns of behavior, minimal resource requirements and low levels of task jitter. [2].

```c
void TaskFunc(void)
{
    while(1)
    {
        TaskX(); // Perform the task
        Delay(); // delay according to the time you set
    }
}
```

**Listing 1. Task in form of super loop**

To implement the TTC scheduling, a hardware timer is used to generate interrupts on periodic basis. When this kind of interrupt occurs, we call it “a tick”. The tick interval can be set to 10 ms or others, depending on the specific applications, but the shorter the tick interval, the heavier the overhead of the systems.

A key requirement in applications using a TTC scheduler is that the task duration must satisfy the following condition [1]:

$$\text{Duration}_{\text{Task}} < \text{TickInterval}$$

But in some circumstances there can be a need to run both long infrequent tasks and short frequent tasks, these two requirements conflict in a pure TTC system although Pont has provided various ways of meeting this requirement ([1]). To solve this problem, we introduce preemptive scheduling into SinewOS. We give more details about the implementation in section 4 and section 5.

In order to make the systems employing SinewOS more responsive than a pure TTC scheduler, we incorporate features of event-triggered architecture in SinewOS.

### 3. Structure of SinewOS

In this section, we first provide some information about task and priority assignment between tasks, then two important variables and three chains are described, which help to implement the scheduler.

#### 3.1. Tasks and priority assignment

When developing an embedded system based on SinewOS, tasks are divided into three categories: cooperative tasks, preemptive tasks and ISR_Type task. The tasks which is brief, critical and runs frequently should categorized as cooperative tasks, and the preemptive task should be the long, infrequent tasks, without accurate timing requirement. While the ISR_Type tasks is the tasks that have higher priority than that of preemptive task but lower than that of cooperative task and can be preempted by other tasks. The most important characteristic of ISR_Type task is that interrupt must trigger it first, which is discussed further in section 6.

SinewOS can manage up to 62 cooperative tasks and any number of preemptive tasks, but the more the number of preemptive task, the heavier the overhead of the systems employing SinewOS.

The first level priorities with the number of 64 are assigned to all tasks but all preemptive tasks as a group are assigned to one priority, while every task in other categories has a fixed unique priority. At the second level, priorities are just used to serve preemptive tasks. Notice that the second level priorities are dynamic because they will be changed during the execution of application software.

Obviously, except preemptive tasks, every task has a fixed unique priority, while preemptive tasks have two, the dynamic and static, and the second is the same in all preemptive tasks.

#### 3.2. Core variables

When a task is created, it is assigned a task control block (TCB). A TCB is a data structure which collects together the information required about each task.

An array, TCBPrioTbl[], which has a size equal to the number of the priorities used in an application, is used to store the address of TCB of each task, except
that of preemptive tasks. The index of the array is equivalent to the priority, when a task is created the address of its TCB is stored in TCBPrioTbl[], which is done by indexing into the array using the task’s priority.

However, of the elements of TCBPrioTbl[], only a single element is assigned to all preemptive tasks, the reason is that only one priority is assigned to all of the preemptive tasks, which is just higher than that of the idle task. Using this static priority, the address of TCB of the highest priority task in preemptive tasks is placed in the specific assigned unit of TCBPrioTbl[]. As a result, every scheduler time only the highest priority task in preemptive tasks ready to run is allowed to compete for the CPU resource with other categories of task.

3.3. Three chains in SinewOS

There are three chains in SinewOS, one is a singly linked list of free TCBs, one is used to link the TCB (called OSTskTCBList) when a task is created, and the final one is for the preemptive tasks ready to run, named RdyPreemTskList.

When SinewOS is initialized, the free TCBs is linked in a singly linked list, until a task is created, a free SinewOS_TCB is assigned to it and the pointer, SinewOS_TCBList, which points to the beginning of the free TCB chain, is adjusted to the next. At the same time the TCB of the task (irrespective the category of the task) created is linked in OSTskTCBList, and it is always inserted at the front of the chain.

Every tick time, the scheduler follows the OSTskTCBList to examine the time-triggered task. If a task is made ready and is preemptive, it will be linked to RdyPreemTskList, and OSTskTCBList must make an adjustment. But the header of RdyPreemTskList is always the highest priority task in preemptive tasks, more details is given in section 5.2.

As a result, together with the variables mentioned in section 3.2 the data structure of SinewOS is created as Figure 1.

4. SinewOS scheduler implementation

Considering that in uC/OS-II (a real-time kernel, developed by Jean J. Labrosse) task-scheduling timing is constant irrespective to the number of tasks created in an application and it can rapidly find the highest priority task ready to run, these characteristics are particularly useful when developing SinewOS where low task jitter and time determinability are key design considerations, thus in SinewOS the methods of registering the ready tasks and finding the highest priority task ready to run are partially cited from uC/OS-II. As Jean J. Labrosse has noted: each task that is ready to run is placed in a ready list consisting of two variables, OSRdyTbl[ ] and OSRdyGrp. Task priorities are grouped (eight tasks per group) in OSRdyGrp. Each bit in OSRdyGrp indicates when a task in a group is ready to run. When a task is ready to run, it also sets its corresponding bit in the ready table, OSRdyTbl[ ]. To determine which priority will run next, the scheduler determines the lowest priority number that has its bit set in OSRdyTbl[ ]. To find the highest priority task ready to run, the scheduler is implemented with two constant variables, OSUnMapTbl[ ] and OSMapTbl[ ]. For more details, please refer to [5].

Cooperative task can’t be preempted by other tasks and must run to completion. In order to make the running task the one with the highest priority basically, the TTC architecture works. Every tick time, scheduler examines the tasks following the OSTskTCBList chain, started at the beginning, until it reaches the idle task, which is always linked at the end of the chain. However, only the time-triggered tasks in the chain are done. When the TskDlyTime field of task’s TCB, whose value is initialized as the task’s execution period, is decremented to 0, the task is made ready to run. Until all the time-triggered tasks in the chain have been examined, the scheduler goes on to execute the highest priority task. In a tick interval all the tasks due to run are executed in priority order. Listing 2 shows the examination operation partially.

```c
pListTCB = SinewOS_TCBList; // SinewOS_TCBList is a poiter which points to the beginning of OSTskTCBList
while(pListTCB != (SinewOS_TCB *)0)
{
  if( pListTCB->TskDlyTime != 0 && pListTCB->TskTrgType == SinewOS_TimeTrg )
    { pListTCB->TskDlyTime--; }
  if( pListTCB->TskDlyTime == 0 && pListTCB->TskTrgType == SinewOS_TimeTrg )
    { pListTCB->TskDlyTime--; }
}
```
If all the tasks are periodic and have relative deadlines equal to their periods, the test for task-set schedulability is particularly simple: If the total utilization of the task set is no greater than 1, the task set can be feasibly scheduled on a single processor by the EDF algorithm.

5.2. Implementation

A preemptive task is made ready by either a tick or occurrence of events.

When a preemptive task is made ready by a tick, it is linked in RdyPreemTskList, but it is placed with the following rules: First, the relative deadline fields of the TCBs for all tasks remained in RdyPreemTskList must be minus 1. If the relative deadline field is decremented to 0, the task’s TCB can be deleted from RdyPreemTskList or its relative deadline field can be set back to its original value, depending on the environments where SinewOS is employed. Then insert the new ready task’s TCB into the chain according to the value of its relative deadline, the higher the value of the task’s relative deadline, the lower the priority of the task, and the highest priority task is placed at the beginning of the chain. However, if the task is set by events, it is unnecessary to execute the first operation.

As a result, the task at the beginning of the RdyPreemTskList list is the highest priority task ready to run among preemptive tasks. In order to get this task’s TCB easily and rapidly, every scheduling operation the pointer RdyPreemTskListHead always points to the beginning of the RdyPreemTskList.

6. Interrupts handling mechanism in SinewOS

In most cases, tasks are rather short, and, as we know, it takes a lot of time to handle the stack. A running cooperative task can’t be interrupted by interrupts except tick. Considering such a situation, when a running cooperative task is interrupted, the context interrupted (partially computed results) must be stored in a stack, which allows the task to resume execution where it left off when the task regains the control of the CPU. Storage and retrieval of partially computed results carry great runtime overheads, while a cooperative task (function) is short, it is quite possible that handling stack requires more time than to have the task finished. On the other hand, if the execution of a cooperative task is suspended, while the time taken to handle interrupt is not deterministic, which will have bad impact on the starting time of the rest cooperative task due to run at the same tick.
interval, as a result task jitter can occur. As Pont has noted, in many embedded applications (such as those involving control or data acquisition) jitter in task / function start times can have serious implications [7]. Furthermore, every cooperative task must be assigned a stack space if interrupts (except tick) are enabled during the execution of task. Contrarily, only one stack space is required for all cooperative tasks if without interrupts during the execution of cooperative tasks. This allows you to reduce the amount of RAM needed when developing an application. And such an approach has a number of advantages, concluded as follows: make the scheduler simpler, reduce the overhead and make testing easier.

However, to keep the systems employing SinewOS sensitive to external and internal events, SinewOS provides framework for a function, but the developers are responsible for providing the specific code because it relies on the hardware used in applications. This function is used to check whether there are events occurred or not during the execution of the cooperative task just finished. According to the properties of the events, you can define the corresponding service routines as the cooperative or ISR_Type. The difference between these two types is: a service routine defined as cooperative type has the same properties as that of a cooperative task previously described, such as, it has a higher priority than that of other categories of tasks and can’t be preempted, and so on; however, when a service routine is defined as ISR_Type, it can be interrupted and preempted and has a priority just lower than that of cooperative task. When the occurrence of an event is detected, the function will register the corresponding service routine in the ready list according to its properties. This system function should be invoked after the execution of a cooperative task at every tick interval.

When it comes to preemptive tasks, it is a different thing. When an interrupt occurs, execution of the running preemptive task is suspended, and the ISR takes control of the CPU.

7. Evaluation platform and testing result

To explore the performance of the systems employing SinewOS, we have built a testing platform based on an adaptive cruise control system (ACCS) and conducted a series of test using the platform.

7.1. Evaluation platform

Figure2 shows a hardware-in-the-loop simulation platform. The simulation model is implemented using a host PC, which reproduces the behavior of a car traveling down a motorway. And the embedded system stands for an ACCS implemented using one or more embedded microcontrollers.

7.2. Testing result

This testing is based on the testing platform noted in section 7.1 and an embedded system employing LPC2129. Its aim is to test whether the control application developed with the SinewOS, combined with the simulator, can provide the in-vehicle speed control correctly and reliably.

To test the performance of the system employing SinewOS, we have created three cooperative tasks and two preemptive tasks in the application. The cooperative task set is described as follows:

One task is to receive the speed from the host PC, runs once every scheduling tick and has the highest priority 0.
One task is to calculate the throttle using PID control algorithm, runs once every scheduling tick and the priority number is 1.
One task is to send the required throttle to the host PC, runs once every scheduling tick and the priority number is 2.

And the preemptive tasks are used to flash two LEDs after the execution of the cooperative task set at a tick interval.

The software application is tested on LPC2129, and the hardware is connected with the host PC through two parallel ports. When the embedded system powers up, the application starts. It reads speed from the first parallel port, then calculates the required throttle according to the speed just got, finally sends the throttle to the host PC. These operations are executed every 50 ms as the scheduler tick interval. And the new speed will be calculated in host PC when it receives the throttle from embedded system. In theory the ideal
result is that the speed of the simulated vehicle is approaching to and eventually approximately equal to the setting speed. Figure 3 shows the testing result.

![Figure 3. The curve of vehicle speed controlled by embedded system employing SinewOS](image)

In this testing the desirable speed is set to 30 km/h. As Fig.3 shows, the simulated speed is close to the setting speed. We let the whole system run for 3 days but the result maintained the original state, which demonstrates that SinewOS works correctly and stably and basically satisfies the requirements.

However, this testing is implemented under a comparatively simple environment. More effort are still needed to make further research on the performance of the system employing SinewOS under more complex circumstance.

8. Acknowledgments

This work was supported by Natural Science Foundation of China (60501026), Research and Teaching Foundation of ShenZhen University (200513, A20050311).

9. References