逢 甲 大 學
資 訊 工 程 學 系 碩 士 班
碩 士 論 文

以*u*C/OS-II核心設計與實作飛控系統之可調變構型硬即時作業系統

A Configurable Hard Real-Time Operating System Based on *u*C/OS-II Kernel for Flight Controls

指導教授：鍾葉青

研 究 生：張傳鑫

中 華 民 國 九 十 一 年 七 月

摘要

近年來，在電子資訊的應用領域裏，即時系統的重要性已逐漸增加，其中硬即時的要求更通常是國防、航太、醫學及工廠自動化等即時系統設計上的基本規格。有關硬即時系統的設計，除硬體性能的配合外，在軟體設計上，通常包含工作分割、排程演算法設計及即時作業系統的支援等考量，其中即時作業系統的支援為整體硬即時系統的設計與運作基礎。本篇論文係利用一個精簡且為開放程式碼的即時核心，藉由相關工作排程機制的修改，強化其支援硬即時系統的功能，同時並增加其可調變構型的功能，以提供設計者更大的彈性，最後結合修改後的作業系統核心與一個實際的硬即時系統，藉由完整的系統開發環境以測試整體系統功能，同時並討論系統設計人員設計硬即時系統應考量的因素。

關鍵字：即時系統，嵌入式系統，硬即時，安全關鍵性，排程，可調變，即時核心，*u*C/OS-II。

Abstract

Recently, real-time systems have become more and more important in the field of electronic and information applications. And the requirement of hard real-time has often become the basic specification for the real-time systems used in national defense, space technology, medication application and industrial automation.

Many applications deal with sets of activities or tasks and make them real-time systems. They need an appropriate real-time operating system to support their development and operation.

The basic design criteria for hard real-time systems includes hardware selection, tasks arrangement, scheduling design and support of real-time system. Among them, the real-time operating system is the most critical and essential issue.

*u*C/OS-II is a real-time operating system with open source code. Because of the characteristics of priority-driven and preemptive for *u*C/OS-II, it is only used to develop the soft real-time system. So we tried to revise the original kernel and increase its hard real time performance.

In this paper, we discuss how to make *u*C/OS-II a configurable hard real-time operating system. We verified our proposed solution on a simplified standalone simulation system. And an existing hard real-time system is implemented base on our revised *u*C/OS-II. The whole system is tested in the integrated development environment. Finally, the design criteria to build a hard real-time system are also evaluated.

Index term: Real-time system, embedded system, hard real-time, safety-critical, scheduling, configurable, real-time kernel, *u*C/OS-II

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time systems have become more and more important these days. There are a lot of real-time systems in many applications, such as industrial and automation systems, computing network systems, medical instruments and devices, military defense systems, aviation equipments and entertainment applications. For those systems, the correctness of the systems depends not only on the logical result of the computation, but also on the time at which the results are produced and completed.

Traditionally, there are two kinds of classifications for real-time systems. They are soft real-time and hard real-time systems. Soft real-time systems have no strict rules to deal with the timing requirement. It is acceptable to miss the deadlines occasionally. On the other hand, the timing requirement is very important for hard real-time systems. Missing the deadline does not only make the system crash, but also will lead to the catastrophic result. So the development of hard real-time systems has always been a critical challenge.

Among the hard real-time systems, safety critical systems have direct influence to human's life. They are found in a wide range of industrial applications, such as nuclear, chemical, aerospace and defense industries.

Safety critical applications often involve several distinct activities, each of which has 'hard' inviolable timing constraints.[1] Developing such systems to meet critical deadline requirement is a big challenge for system designers. Those activities, also called tasks, should be carefully designed in the early stage of system development. From the

industrial experience, different tasks are often divided into several software modules. The reason for this strategy is that it is convenient for maintenance and easy to integrate team efforts.

With the fast development of electronic industry, the performance of the electric hardware has become more and more powerful. The computing speed has also become tremendous fast. There is a common mistake that many people think only the powerful hardware can achieve the real-time requirement. In fact, the function and performance of the whole system comes from the operating combination of hardware and software. And the software often plays a critical role.

For a complicated system, the load for the system to handle the tasks is often very heavy. To meet the system's design criteria, it needs the strict principles to arrange the software modules and system resources optimally.

To design a hard real-time system, it often involves several design issues in the system development process. For a system designer, it is essential to break the whole system into several software modules. Then the scheduling mechanism should be carefully designed to meet the real-time requirement.

The scheduling mechanism is operated according to the specific scheduling theory. Hard real-time scheduling theory uses a simplified, abstract computational model to represent the behavior of a preemptive, multitasking system. [2] Solving the scheduling problems is essential for designing the real-time systems. For decades, there have been a lot of studies in designing the scheduling theory and methodology. But there is still no optimum solution. The real design is always tradeoff between the performance and

system operation.

A system often consists of hardware and software. And the effort of software design is mainly focused on the application layer. For convenience, the application is often built on an off-the-shaft operating system. To build a hard real-time system, there needs the full support from the operating system.

The operating system is responsible for the resource management. It supports the application layer to access the hardware and manage the different tasks to operate timely and correctly.

The recent trend is to use the non-commercial operating system with open source code such as Linux and uC/OS-II to build the real-time system. The reason is not only free of charge for proprietary fee, but also convenient in revision to meet the system's unique requirement. The most important is that the system designers can have fully understanding and control about the detail of the whole system.

*u*C/OS-II, which stands for MicroController Operating System Version 2, is built by Jean J. Labrosse in 1998. It is based on *u*C/OS that is the first version and was published in 1992. The *u*C/OS-II is a real-time kernel. It is a highly portable, ROMable, very scalable, preemptive real-time, multitasking kernel for microprocessors and micro controllers. It can manage up to 63 application tasks and provides the following services: semaphores, message mailboxes, message queues, task management (create, delete, change priority, suspend/resume etc.), fixed sized memory block management, time management, mutual exclusion semaphores (to reduce priority inversions) and event flags. And the latest version is V2.51. [3] Because it is a kernel with open source code, it can be

tailored to meet the requirements of different applications.

The reason we choose the *u*C/OS-II is because it is simple, compact, well organized and documented. It is a good choice to be used to develop an embedded real-time system.

Although *u*C/OS-II is called a real-time operating system, its original design has limit to support the hard real-time requirement. To make *u*C/OS-II a hard real-time operating system, we revise the system functions of the kernel about task creation. And a parameter is used as a reference to check the real-time property of the task. The hard real-time task's execution is protected to meet its deadline. Furthermore, we use a predefined timing parameter to check the execution of the hard real-time task to prevent the deadlock situation. And we also design a mechanism to make the kernel become re-configurable on-line. All the revision about the *u*C/OS-II only adds its capability to support the hard real-time features, but does not affect its original function and performance.

The rest of the paper is organized as follows. In Chapter 2, we discuss the background of the problems. We present our design, analysis and experiment results in Chapter 3. The paper is concluded in Chapter 4.

Chapter 2

Background

2.1 The Hard Real-time Issues

Hard real-time is the essential requirement for safety critical systems. To meet the requirement, it is necessary to have an adequate scheduling mechanism and the full support of the real time operating system.

A real-time system has a known set of tasks. It is also often desirable that time-critical and non time-critical tasks coexist in a same real time system. For hard real-time systems, the time-critical tasks must complete their jobs within their required deadline. Traditionally, to guarantee the tasks' real-time requirement, the software programs are often designed as infinite loop containing several tasks and execute by sequence as the example shown in Figure 2.1. [4]

In such a program, all the tasks can complete their jobs to the end. But the problem with such design is that it does not scale and won't meet the requirement of modern complicated systems. As real-time systems get more complex, they need more complex support and design philosophy.

```
Counter=500;
While(1){
    If(data_on_sensor()){
        Read_sensor();
        Compute_output();
        Counter--;
    }
```

```
        if(!counter){
        output();
        counter=500;
        }
    }
```

Figure 2.1 An example of a simple program loop

Recently, real-time engineers often use the multi-tasking or multi-thread techniques to build their systems. And they also use the priority parameters to handle the sequence of the tasks' execution. To increase the system's capabilities, they also build their systems as preemptive. So, with the fast development of electronic hardware and software engineering, the real-time systems have become more powerful. However, the engineering efforts needed to design a real-time system also become more and more complicated.

Basically, there is a contradiction between hard real-time requirement and preemptive design. For hard real-time systems, all the time-critical tasks have to meet their individual deadline. But, for preemptive systems, the task with higher priority can preempt the current executing task with lower priority. Under such condition, not all the tasks can finish their deadline as required. So, there should have a refinement based on the real need.

In order to build a hard real-time system with the functions of multi-tasks and preemption, the system designers have to arrange the predefined tasks carefully in the early stage of system development.

On the other way, if the operating system can support the development of hard real-time system, the system designers can focus their efforts on the problem solving and the whole design process will become efficient and time saving. So an appropriate operating system will prove to be a big help to the system designers. That is the reason we choose the *u*C/OS-II as our target of research.

2.2 The Architecture of the *u*C/OS-II

Jean Labrosse publishes the real time operating system *u*C/OS-II, which stands for Micro Controller Operating System Version 2, in 1998. It is upward compatible with *u*C/OS first published in 1992. The new version provides many improvements, such as the addition of a fixed-sized memory manager; user-definable callouts on task creation, task deletion, task switch, and system tick; Task Control Block (TCB) extensions support; stack checking etc. [5] Mr. Jean also added components to adjust every function, and made *u*C/OS-II much easier to port to different processors. So the *u*C/OS-II kernel is a good choice for developing the embedded real-time system. The system architecture built on the *u*C/OS-II is shown as Figure 2.2.

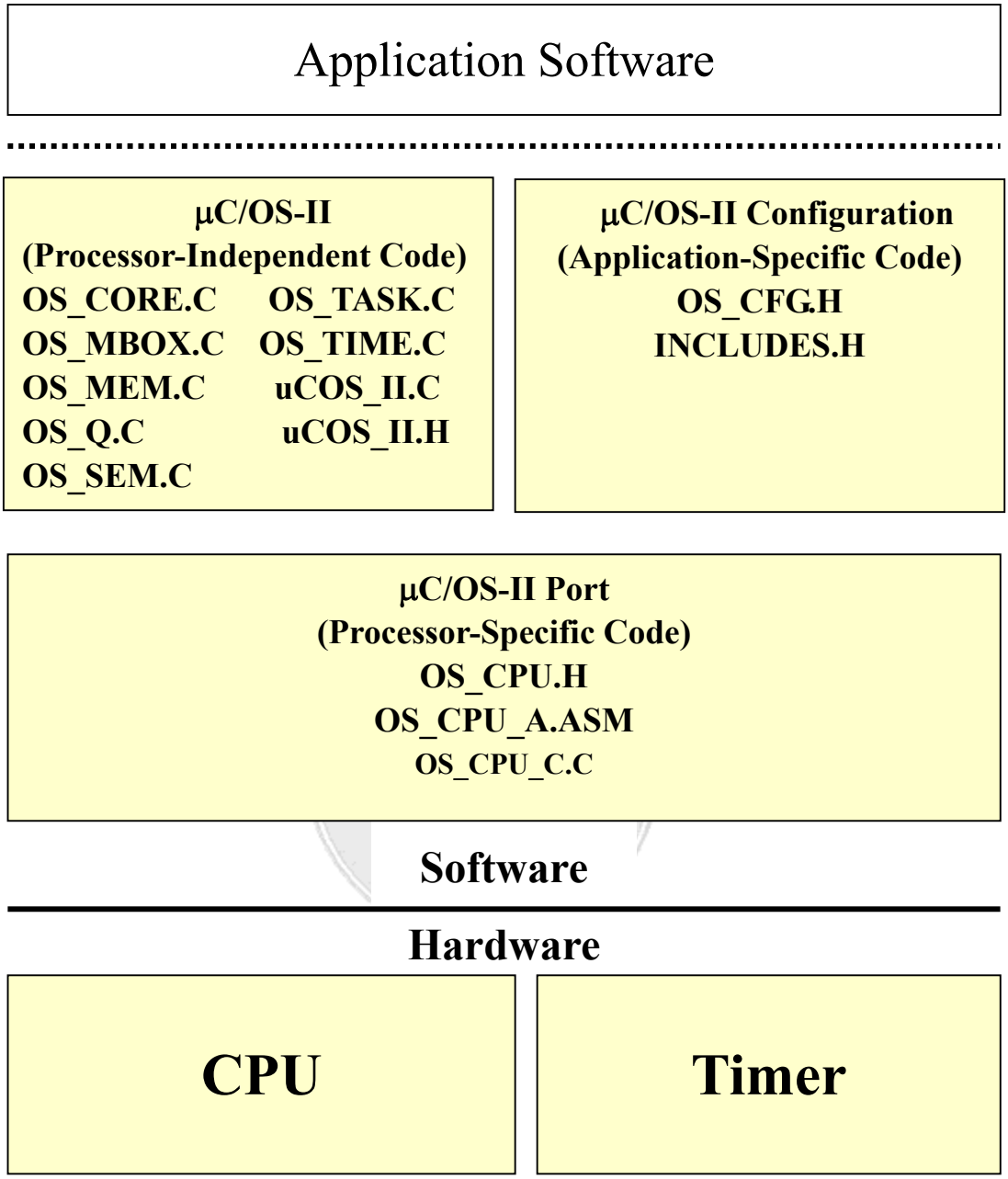

Figure 2.2 *u*C/OS-II hardware/software architecture

2.3 The Operation of the *u*C/OS-II Kernel

*u*C/OS-II is designed to support the embedded systems and can be ported to any microprocessor as long as the microprocessor provides stack ability and interrupt enable/disable functions.

It can manage up to 64 tasks. However, the kernel uses two tasks for system use, which are the idle task and the CPU usage statistics task. Every task has to be assigned a priority and the priority numbers 0, 1, 2, OS_LOWEST_PRIO-3, OS_LOWEST_PRIO-2, OS_LOWEST_PRIO-1 and OS_LOWEST_PRIO are reserved for future use. Therefore, 56 application tasks are available.

In order for *u*C/OS-II to manage the application task, the task has to be created by system call: OSTaskCreate() or OSTaskCreateExt(). There are two kinds of formats for the tasks of *u*C/OS-II. One is the infinite loop task and the other is the simple program code that can be deleted by the system. These two tasks can be declared as shown in Figure 2.3.

Infinite loop:

```
void TaskA(void *data){
    while(true){
        [user code;]
        Call uC/OS-II service to delay or pend;
        [user code;]
    }
}
```

Simple code:

```
void TaskB(void *data){
```

```
        [user code;]
        TaskDelete();
    }
```

Figure 2.3 Task Format of *u*C/OS-II

When a task is created, a Task Control Block (TCB) is assigned and used to maintain the task's states when the task is preempted. The preempted task will be resumed when it regains the CPU control. Each task under *u*C/OS-II's control can be in any one of six states: Dormant, Ready, Running, Delayed, Waiting and Interrupt.

*u*C/OS-II is a priority-driven kernel. Each task is assigned a unique task priority level from 0 to 63. Priority 0 is the highest and priority 63 is the lowest one assigned to an idle task in all cases.

To initialize the *u*C/OS-II, we should call function OSInit() before calling any of its other services. OSInit() initializes all *u*C/OS-II variables and data structures. It creates the idle task OSTaskIdle(), which is always ready to run. The priority of OSTaskIdle() is always set to OS_LOWEST_PRIO. Under some conditions, OSInit() also creates the statistic task OSTaskStat() and makes it ready to run. The priority of OSTaskStat() is always set to OS_LOWEST_PRIO-1.

Multitasking is started by calling OSStart(). OSStart() runs the highest priority task that is ready to run. All tasks that are ready to run are placed in a ready list. The scheduling mechanism always checks the ready list to arrange the task execution based on the task's priority.

The main program for any application built on the *u*C/OS-II is shown as Figure 2.4.

```
void main (void)
{
    OSInit();
    ...
    Create application tasks
    ...
    OSStart();
}
```

Figure 2.4 A sample application main program built on the *u*C/OS-II

2.4 Shortcomings of *u*C/OS-II to Support the Hard Real-Time System

Comparing to the general operating system, the real-time operating system has the advantage that the kernel can monitor the status of the system's resources and control the utilities more precisely. So, in order to build a real-time system, the system designers often choose an off-the-shafts real-time operating system (RTOS) as a platform.

It is common to classify real-time systems into soft real-time and hard real-time. But there is no hard RTOS or soft RTOS. A specific real-time operating system can only allow you to develop a hard real-time system. There are several basic requirements a real-time operating system should have, which are: support for multi preemptive threads, thread priority, synchronization between threads, deterministic timing and support to prevent priority inversion. [6]

Following the above ground rules, the *u*C/OS-II has all the basic RTOS features. It really can be called real-time operating system. But there are many considerations for the system designers to pay much attention to when building a real-time system on it.

The parameter used by the uC/OS-II to manage the tasks' scheduling is only the priority. And there is not any scheduling theory implemented. It completely depends on the application programmers to assign the priorities to the tasks in the system design process. The performance and efficiency mainly depends on the system designers' strict and overall consideration.

Based on the operation of the *u*C/OS-II's scheduler, any running task can be preempted if another task with higher priority is ready to run. This won't meet the

requirement of hard real-time system because that the task's deadline cannot be guaranteed.

However, any real-time operating system supporting the preemptive multi-tasking function will encounter this contradiction. So a specific scheduling mechanism needed to be implemented to the real-time operating system. And there is also some design principles needed to be followed by the system designers in building a hard real-time system.

There are a lot of applications built on the *u*C/OS-II, such as UPS monitor (American Power Conversion Corp., USA), On-Board-Unit (OBU) of a road pricing system (FELA Management AG, Switzerland), embedded controller for small and medium-sized electrical installations (Lexel group, Denmark), electronic news gathering analog/digital microwave transmitter/receiver (Microwave Radio Communications, USA), credit card processing unit (Monitor Business Machines, New Zealand), 3-axis motor control card (National Optronics Inc., USA), motorized self service payment terminal (Prism Holdings Ltd., South Africa), embedded controller (X-traWeb Inc., USA), standalone static power switch (Siel, Italy), satellite monitoring and management system (Vistar Datacom, USA), and racing-car MPEG-2 video system (Wescam, USA) etc. [7]

The above systems are all commercial products. And the companies are all over the world. Those products are all real-time embedded systems. This proves that the *u*C/OS-II is suitable for building a real-time embedded system.

On the other way, there are many researches and solutions about how to build a hard-real time kernel. Current solutions to make the operating system support the hard

real-time systems are on many ways. There is approach on the design of the scheduler. [8] However, more efforts are focused on the revision about the existing operating systems, such as Linux.

Linux is a general-purpose operating system. The reason for its popularity is that it is open source code. More and more people around the world use it and make change to it. The community is growing tremendously fast. Various efforts are contributed to enforce the performance and solve the problems of Linux. To build the real-time capability into the Linux has also attracted a lot of works.

The Real-time Linux (RT-Linux) project at the New Mexico Institute of Mining and Technology is the first promising contribution. [9] They minimize the changes made to Linux itself. They design a new real-time component as a kernel module to handle all real-time tasks. And they also design an interface called *real-time fifo* to communicate with the original Linux kernel. The new design makes Linux to be preemptive and meet the requirement of hard real time.

Before using RT-Linux, an embedded system designer has to be sure that all of the needed functionality could be fit into one of the two domains, real time or non real time. If a low-level device driver is implemented in the specific domain, then the task in another domain cannot use it.

The current design lacks the function of priority inheritance to prevent priority inversion.

RED Linux is originally a research project leaded by Kwei-Jay Lin in University of

California. [10] Recently the researchers set up a company to develop different products based on the original research. Unlike RT Linux, RED Linux has almost new scheduling design. The scheduler is divided into two components: the Allocator and the Dispatcher. The Dispatcher implements the different basic scheduling mechanism, and the Allocator implement the policy that manages the CPU time and the system resources to meet the real-time constraints of user jobs. They use preemption point to improve the kernel's response time and ensure the real-time requirement is met.

RED Linux provides a general scheduling framework to support different real-time scheduling paradigm. It still has the shortcoming of large scale. To be used in the embedded system, it needs a lot of efforts to tailor the kernel.

KURT means KU Real-Time Linux and is developed by the university of Kansas. [11] They avoid the defect of RT Linux that the real-time tasks have no access to any Linux kernel services. The design of KURT follows a modular approach by providing a core which is responsible for the scheduling of real-time events and allowing the addition of real-time modules which implement the functionality of a specific real-time task. Those functions can be called as system calls.

KURT only supports firm and soft deadlines. The reason for that is because it is limited by the original Linux structure. It cannot be sued as the kernel to build the hard real-time system.

All of the above revisions about Linux still face a same problem that Linux is a heavyweight ordinary operating system and not suitable for embedded system. So there are a lot of research and effort in progress to shrink the Linux kernel and make it real-time at the same time. But most of the works are done in the industry and they are

propriety products.

Other real-time operating systems are commercial products, such as LynxOS [12] and QNX [13]. Except for the cost of royalty, their internal structure are black boxes, the only way the designers have to do is to follow their rules and limits. It is not suitable to be used as the basis to build the safety critical systems.

For the scheduler design, the feasibility depends on the efficiency of scheduling algorithms. But they are all based on the assumption, which simplifies the tasks' operation. And the models are all too simple to be implemented in real applications. So there is a big gap between the academic research and industrial application.

However, for those revisions of existing operating system, most of them are all focused on general applications. Based on the original design, there are limits for some specific application, such as embedded system. Sometimes it is not appropriate to build system on them because of the existing limitations.

In the industrial and commercial market, there are a lot of off-the-shafts real-time operating systems to support the development of hard real-time systems. But the price is high. There is an urgent need for free open source code, such as the Linux and *u*C/OS-II. Recently, there are a lot of efforts to make the Linux become a real-time kernel. It is because that the Linux is very popular in the community of real-time systems. But the *u*C/OS-II has more advantages than the Linux in some applications. The *u*C/OS-II is more compact and has smaller size.

The target of our research is focused on the *u*C/OS-II. Our solution is to revise the

original source code of *u*C/OS-II to add its capability to support the operation of hard real-time systems instead of affecting its original functions.

Chapter 3

Proposed Solutions

3.1 Revisions of *u*C/OS-II to Support the Hard Real-Time System

3.1.1 Hard Real-Time Guarantee

*u*C/OS-II is a priority-driven kernel. It always executes the highest priority task ready to run. The determination of which task has the highest priority is made by the scheduler. According to the original design, the task-level scheduling is performed by the system call OSSched() and the ISR-level scheduling is handled by another function OSIntExit().

To make *u*C/OS-II a hard real-time operating system, we have to revise the scheduler. The revised scheduler will check the properties of every task before executing scheduling arrangement.

For all the tasks, the system designer or programmer has to make a classification in the beginning of system design. Basically, two groups are recommended. They are hard real-time tasks and non-hard-real-time tasks. For the hard real-time tasks, deadline guarantee is the basic requirement. So the interruption is disabled and there is no preemption during their execution. The task can execute within deadline. And the hard real-time requirement is achieved.

In order to guarantee the hard real-time tasks' execution, we try to let the system enter the critical section during their execution. There are two functions used to protect critical section of code, which are OS_ENTER_CRITICAL() and

OS_EXIT_CRITICAL(). OS_ENTER_CRITICAL() disables interrupts and OS_EXIT_CRITICAL() enables interrupts.

We add new parameter **OS_HRT** in the function OSTaskCreate() or OSTaskCreateExt() as shown in Figure 3.1 and 3.2. So the programmer has to set the parameter of task's property before it is created.

```
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U
prio, INT8U OS_HRT)
{
    void *psp;
    INT8U err;

    OS_HRT = TRUE;                    /*TRUE for the hard real-time tasks
    If (prio > OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    .....
    .....
}
```

Figure 3.1 Code for function OSTaskCreate()

```
INT8U OSTaskCreateExt (void        (*task)(void *pd),
                       void         *pdata,
                       OS_STK      *ptos,
                       INT8U        prio,
                       INT16U       id,
                       OS_STK      *pbos,
                       INT32U       stk_size,
                       void         *pext,
                       INT16U       opt,
```

```
                                        INT8U OS_HRT)
{
    void          *psp;
    INT8U          err;
    INT16U         i;
    OS_STK    *pfill;

    OS_HRT == TRUE;                          /*TRUE for the hard real-time tasks
    If (prio > OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    .....
    .....
}
```

Figure 3.2 Code for function OSTaskCreateExt()

After the task is created, its property is set. Upon return from OSTaskCreate() and OSTaskCreateExt(), the main program continues to execute. The function OSStart() is called, multitasking has started.

The code for OSStart() is shown in Figure 3.3. When it is called, it finds the highest priority task from the ready list. Then, OSStart() calls OSStartHighRdy() for the processor being used. OSStartHighRdy() restores the CPU registers by popping them off the task's stack then executes a return from interrupt instruction, which forces the CPU to execute your task's code.

In order to ensure the hard real-time tasks to execute within its deadline, we add a check function in OSStart(). Before OSStartHighRdy() is called, the task's property is checked. For hard real-time systems, we force the kernel enter the critical section by calling OS_ENTER_CRITICAL(), then the task is executed. After the task completes,

OS_EXIT_CRITICAL() is called to leave the critical section. And for the non-hard-real-time tasks, there is no need to enter the critical section during their execution.

```
---------------------------------------------------------------------------------------------------
    void OSStart (void)
    {
        INT8U   y;
        INT8U   x;

        if (OSRunning == FALSE) {
            y             =  OSUnMapTbl[OSRdyGrp];
            x             =  OSUnMapTbl[OSRdyTbl[y]];
            OSPrioHighRdy =  (INT8U)(( y << 3 ) + x );
            OSTCBHighRdy  =  OSTCBPrioTBl[OSPrioHighRdy];
            OSTCBCur      =  OSTCBHighRdy;
            If (OS_HRT == TRUE) {
                OS_ENTER_CRITICAL();
                OSStartHighRdy();
                OS_EXIT_CRITICAL();
                }
                Else {
                OSStartHighRdy();
                }
        }
    }
---------------------------------------------------------------------------------------------------
```

Figure 3.3 Code for function OSStart()

For our design, there is also a convenient way to change the task's property. Like the function OSTaskDel() is used to delete a task after it is created, the task's real-time property can also be changed at any time just by setting the parameter ***OS_HRT*** to false.

3.1.2 Deadlock Check

For the hard real-time tasks, the deadlines are guaranteed by entering the critical section. But there could cause the deadlock problem. So the execution time of these tasks should be monitored.

The problem is how to check the execution time of the tasks and terminate the critical section. During the execution of the critical section, the code of the task can complete its all jobs. When there is an error or any possible problem caused by computation or coding, the kernel should have a mechanism to avoid the system's failure. We use a software test point to check the correctness of the task. When the execution time of a specific task is over the predefined deadline, the kernel should terminate the task and enable the following task ready to run.

We add new parameter **OS_ExecTime** in the function OSTaskCreate() or OSTaskCreateExt() as shown in Figure 8 and 9. So the programmer has to set the parameter of task's property before it is created.

The execution time is like the software test point to keep the task not reaching a deadlock.

---

*INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio, INT8U **OS_HRT**, INT8U **OS_ExecTime**)*

{

*void *psp;*

INT8U err;

```
    OS_HRT = TRUE;                    /*TRUE for the hard real-time tasks
    OS_ExecTime=predefined value;     /*assign the task's maximum
                                        execution time

    If (prio > OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    .....
    .....
}
```

Figure 3.4 Code for function OSTaskCreate() with time check

```
INT8U OSTaskCreateExt (void        (*task)(void *pd),
                       void        *pdata,
                       OS_STK      *ptos,
                       INT8U       prio,
                       INT16U      id,
                       OS_STK      *pbos,
                       INT32U      stk_size,
                       void        *pext,
                       INT16U      opt,
                       INT8U OS_HRT
                       INT8U OS_ExecTime)
{
    void          *psp;
    INT8U         err;
    INT16U        i;
    OS_STK        *pfill;

    OS_HRT == TRUE;                   /*TRUE for the hard real-time tasks
    OS_ExecTime=predefined value;     /*assign the task's maximum
                                        execution time

    If (prio > OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
```

```
    …
    …
}
```

Figure 3.5 Code for function OSTaskCreateExt()with time check

After the task is created, its maximum execution time is set. In order to ensure the hard real-time tasks to execute within their deadline, we add a check function in user created tasks. Before OSStartHighRdy() is called, the task's property is checked. For hard real-time tasks, we check the execution time. When the execution time exceeds the predefined value, the task is forced to leave the critical section. And for the non-hard-real-time tasks, there is no need to check their execution time because interrupt is allowed. The pseudo code for the task will become as Figure 10.

Infinite loop:

```
void TaskA(void *data){
    while(true){
        start=clock();
        for(run_times = 0; run_times < N; run_times++){
        [user code;]
        Call uC/OS-II service to delay or pend;
        [user code;]
        }
        end=clock();
        exe_time=(end-start)/N;
        if( exe_time > OS_ExecTime) exit;
    }
}
```

Simple code:

```
void TaskB(void *data){
    [user code;]
    TaskDelete();
}
```

Figure 3.6 Task Format of *u*C/OS-II with execution time check

The above method uses the ANSI C *clock()* function. We can obtain the time more precisely by increasing the bound of run_times (i.e. N).

The *u*C/OS-II also provides system functions to measure a task's execution time, which are PC_Elapsedstart() and PC_ElapsedStop(). These two functions are implemented by using the PC's 82C54 timer#2. Before using these two functions, we need to call the function PC_ElapsedInit() to initialize the elapsed time module by determining how long the start and stop functions take to execute. The execution time (in microseconds) returned by PC_ElapsedStop() consists exclusively of the code we are measuring. The pseudo code for the task will become as Figure 11.

Infinite loop:

```
void TaskA(void *data){
    while(true){
    PC_ElapsedInit()
    If (OS_HRT == TRUE) {
      PC_ElapsedStart();
        [user code;]
        Call uC/OS-II service to delay or pend;
        [user code;]
      exe_time= PC_ElapsedStop();
      if( exe_time > OS_ExecTime) exit;
    }
```

}

Simple code:

*void TaskB(void *data){*
[user code;]
TaskDelete();
}

Figure 3.7 Task Format of *u*C/OS-II with execution time check(2)

3.1.3 Real-Time Metrics

For the development of real-time systems, it is important to have measuring tools to test the evolution of the executing tasks. Although *u*C/OS-II has already provided several parameters to check the timing characters of the executing tasks, such as calculation of tasks, CPU usage and task switch time etc., there are no parameters used for checking the status of specific hard real-time tasks. So we follow the idea proposed in RT-Minix [14] to build a data structure that is accessible to the user via a system call. The structure includes the following items:

typedef struct {
INT8U **OS_NoHRT**; /* Number of hard real-time tasks*/
INT8U **OS_NoMisDln**; /* Number of missed deadlines or forced leaving critical section*/
…
} rt_status

The parameter **OS_NoMisDln** comes from the calculation that the executions of hard real-time tasks exceed their predefined deadline. We can add up the number after it reaches a predefined value and increase the deadline of the hard real-time tasks that

experience missing deadline. Another threshold for the deadline increment has to be defined in advance to activate a false alarm or shut down the whole system for the safety issue.

There are other timing parameters can be defined in the data structure depending on the requirements of the system designers. All the statistics can be monitored online easily.

3.1.4 Reconfiguration

There is need for the operating system to be re-configurable during operation. The reason for this function is because the tasks' characteristics need to be modified based on the specific requirement of operation or consideration of system performance.

For an embedded real-time system, the function is usually designed to meet the specific requirement. Sometimes, in certain circumstances, the executing sequence and priority of the tasks need to be re-arranged. This situation can be activated by the operator or the system itself based on the predefined condition. We can add the criteria check point in the task execution module and change its real-time characteristic by setting the parameter **OS_HRT** true (for hard real-time) or false (for non hard real-time). Also, we can change the task's priority by change the value of the parameter **prio.**

The algorithm for operator-triggered reconfiguration is shown as Figure 3.8.

---

set checkpoint for task X;
check criteria(ex. specific system status or critical mission);
accept the operator's command to change the task's real-time character;
while meeting the criteria, do

set OS_HRT true;
end do;
task X becomes hard real-time and deadline guarantee;
accept the operator's command to restore the task's real-time character;
set OS_HRT false;

Figure 3.8 Algorithm for operator-triggered reconfiguration

On the other way, when there are hard real-time tasks missing the deadline very often during operation. It is necessary to re-configure all the task's timing characteristics to maintain the original required performance.

In order to prevent the deadlock of the hard real-time tasks, we increase their execution time. But there needs further scheduling evaluation for all the tasks to get the optimization.

The original scheduling policy for *u*C/OS-II is priority-driven. The priority of the task is defined by its importance. For the practical design philosophy, the tasks of a real-time system are often arranged in a specific time frame. And the job of the system designer is to keep all the tasks occurring, executing and completing inside the main time frame. Our on-line mechanism of re-configuration is to ensure the predictable behavior. And the baseline is under the constraint of the original time frame for all the tasks.

There are several parameters for the tasks can be used as the reference of re-configuration. We increase the execution time to prevent the deadlock of hard real-time tasks. Because the execution of the hard real-time tasks is guaranteed, no other tasks can preempt during critical section. This will affect other tasks' execution. So, in order to

ensure the whole system's performance, we lower the rate or increase the occurring period of the hard real-time tasks whose execution time has been increased. We add new parameter **OS_Period** in the function OSTaskCreate() or OSTaskCreateExt(). Its value has to be set before the task is created. And when the execution time of the task has been forced to change, we change the period at the same time.

The algorithm for system-automatic reconfiguration is shown as Figure 3.9.

---

set checkpoint to check the number of missing deadline for task X;
set threshold for deadline increment ⊿t and iteration number(which are related with other task's period);

execute deadline check for hard real-time task X;
while missing the deadline, do
    exit task X;
    deadline = deadline + ⊿t;
    count = count + 1;
    if count >= predefined value, then
        exit task X;
    end if;
end do;
execute deadline check for hard real-time task X;
while missing the deadline, do
    activate hardware backup or operator's manual control(redundancy design);
end do;

set threshold for refined period for critical non hard real-time task Y;
count the executing occurrence of task Y;
revise OS_Period for task Y until it operates normally;
while task Y s' period is greater than refined period, do
        activate hardware backup or operator's manual control(redundancy design);

end do;

Figure 3.9 Algorithm for system-automatic reconfiguration

The principle of the on-line re-configuration can also be used in off-line evaluation to test system correctness without actually running it.

3.2 Analysis

To check the effectiveness of the revised *u*C/OS-II, we use an existing hard real-time system as the based target, which is a typical Flight Control System (FCS) and is built on *u*C/OS-II kernel by Chen. [15]

3.2.1 A Typical Flight Control System

The FCS discussed in this thesis is primarily designed for the Unmanned Aerial Vehicle (UAV). It is built with an embedded Operational Flight Program (OFP) executed on the Flight Control Computer (FCC). Because the UAV is controlled by the remote Ground Control Station (GCS), there is no complicated user interface for the pilots. So the software OFP is simpler than traditional avionics system. The major function of the FCS is to get the external sensor data, and perform the computation of control laws to achieve aircraft flight dynamics control. In addition, the FCS contains the communication data link interface to receive uplink commands from the GCS, and to transmit aircraft's operational status as downlink message to GCS for pilot monitoring.

Traditionally, the FCS can be considered as a closed loop control system and implemented with a Proportional-Integral-Derivative (PID) controller. Its major task is

responsible for the plane's maneuver and maintenance of the flight stability. The basic idea for the Flight Controller is to get the sensors' input, conduct the control-law computation and output commands to activate the actuators continually. The design of the controller will seriously influence the flight quality of the plane. But it belongs to the field of control theory and practice. We will only focus on the design of software architecture.

The simplified model for flight controller is shown as Figure 3.8. Such a system is called a feedback control loop and can be implemented as an infinite loop in Figure 3.9.

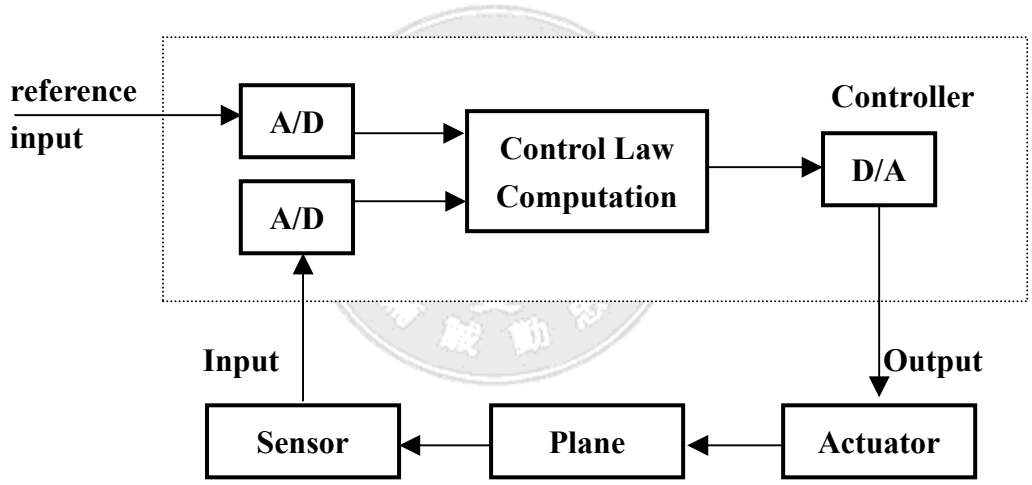

Figure 3.10 A simplified flight controller model

---

set timer to interrupt periodically with period T;
at each timer interrupt, do
sample and digitize sensor readings to get measured values;
compute control output from measured and state-variable values;
convert control output to analog form;
estimate and update plane parameters;

compute and update state variables;
end do;

Figure 3.11 Implementation of A Generalized of Flight Controller

In the above figure, we consider the FCS as a multi-input/multi-output type and the assumption is that the system can provide a timer to set the sampling period T. The selection of sampling period is mainly dependent on the perceived responsiveness of the overall system (i.e. the plane and the controller) and the dynamic behavior of the plane. In general, the faster a plane can and must respond to change in the reference input, the faster the output to its actuator varies, and the shorter the sampling period should be.

The FCC is an 80486-based Industrial Personal Computer (IPC) with several input and output interfaces as shown in Figure 3.10.

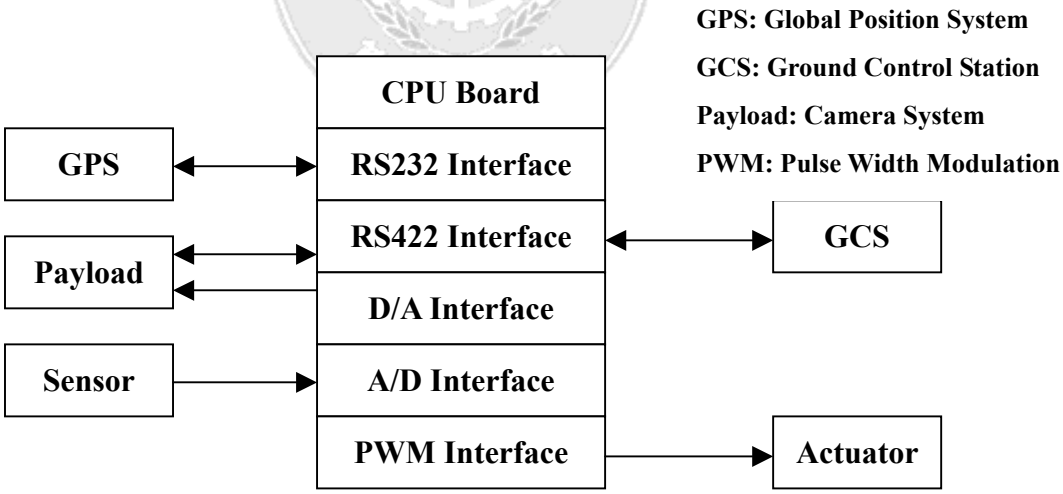

Figure 3.12 Hardware modules of a typical flight control system

The FCC gets sensor data including altitude, altitude rate, airspeed, heading, attitude data, fuel quantity and engine data through the A/D interface, GPS data through

RS422 interface, and receives uplink commands from GCS by way of the RS422 interface. After data processing and computation, it then sends actuator outputs including surface, fuel, nose wheel and brake commands through the PWM interface, payload's rotation commands through the D/A interface, and transmit downlink messages to GCS by way of the RS422 interface.

The hardware interrupt mechanism in the FCC is similar to that of the general PC system. It contains two interrupt controllers (Intel 82C59A PIC) to provide 15 sources of interrupts to CPU. Interrupts are labeled IRQ0 through IRQ15. The priority order of the interrupt is according to its IRQ number: the lower the IRQ number, the higher the priority order. The interrupt arrangement for FCC is shown in Table 3.1.

Table 3.1. Hardware interrupts used for FCC

IRQ number	Interrupt Vector	Description	Function	Priority Order
IRQ0	8	Timer	*u*C/OS-II Real-time Clock	**1**
IRQ9	121	COM4 (RS422)	Data Link	**2**
IRQ15	127	Spare	Watch Dog Timer	**3**
IRQ3	11	COM2 (RS232)	DGPS	**4**
IRQ4	12	COM1 (RS232)	GPS	**5**
IRQ5	13	COM3 (RS422)	Payload	**6**

Based on the hardware architecture and system's function requirement, the software OFP can be built with consideration for modular design. The software modules of the system are designed as shown in Figure 3.11.

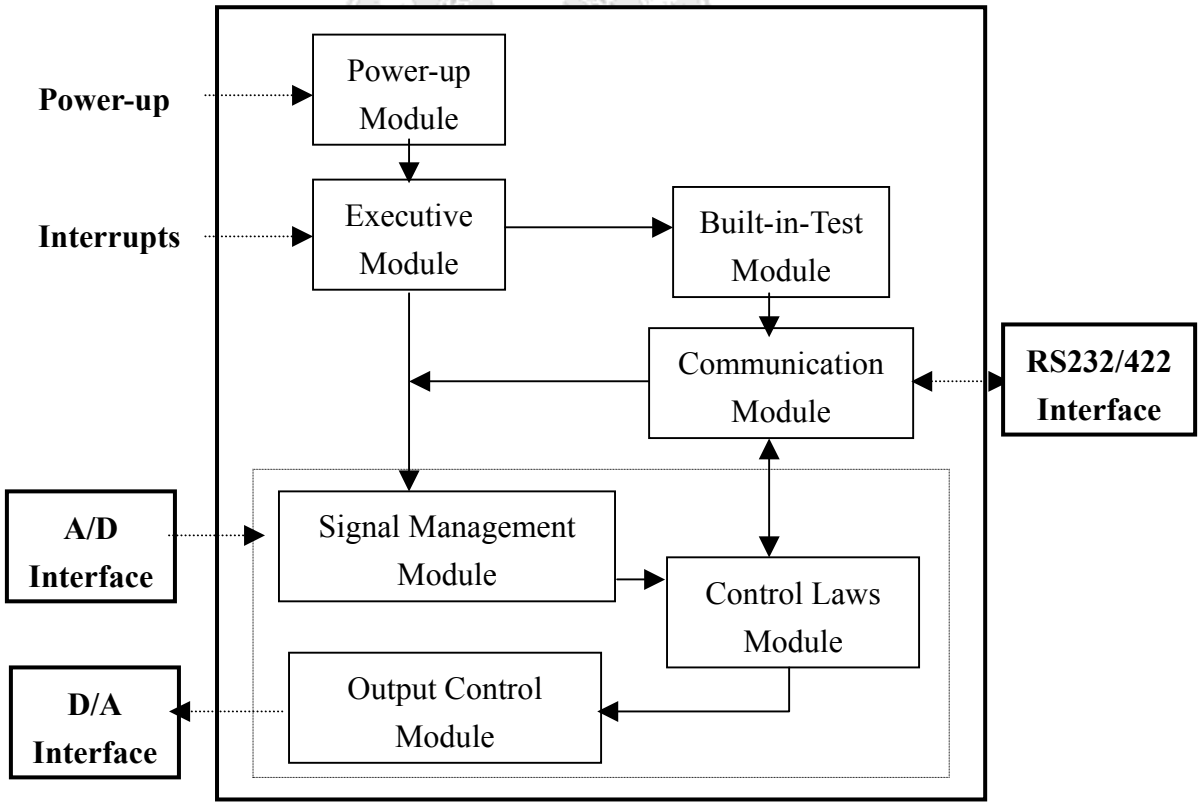

Figure 3.13 Software modules of a typical flight control system

Basically, the flight control system should execute different tasks to meet the requirements of flight dynamics control. Each individual task contains different specific functions or subroutines. They are designed to get and convert the sensor data, to perform the control law computation, and to convert and output the actuator control data. So the flight vehicles can fly smoothly and safely in the air.

From Chen's design, the main program of the simplified flight control system built on the *uC*/OS-II is shown as Figure 3.12 and the function, task type and priority order are shown as Table 3.2. [11]

```
void main (void)
{
    OSInit();
    ...
    OSTaskCreate(MajorTask,...,Major_PRIO); //Create MajorTask
    OSTaskCreate(COM1Task,...,COM1_PRIO); //Create COM1Task
    OSTaskCreate(COM2Task,...,COM2_PRIO); //Create COM1Task
    OSTaskCreate(COM3Task,...,COM3_PRIO); //Create COM3Task
    ...
    OSStart();
}
```

Figure 3.14 Main program of the simplified flight control system built on the *u*C/OS-II

Table 3.2 Tasks of the simplified flight control system built on the *u*C/OS-II

Task Name	Main Function	Working Rate	Task Type	Priority order
Major Task	● Input signal computation ● Control law computation ● Output control	40Hz	Periodic	4
COM1Task	Receive the GPS data	10Hz	Periodic	2
COM2Task	Receive the payload feedback data (GPS transmits the position data in 10Hz)	10Hz	Periodic	3
COM3Task	Receive flight command: ● Manual & Autopilot Mode ● Navigation & Return Home Mode (Command-driven)	 ● 10Hz —	 ● Periodic ● Aperiodic	1

The PID flight controller for flight dynamics control is implemented in the Major Task. This task contains jobs to get and convert the sensor data, to perform the control law computation, and to convert and output the actuator control data. There are specific functions or subroutine to complete all the jobs. Referring to Figure , the Major Task for the flight controller will contain the Signal Management module, the Control Laws module and the Output Control module. As required, the sampling rate of the flight controller is 40 Hz. So the period of the Major Task is 25ms, and the deadlines of the jobs inside the task should coincide with this period. The pseudo code for the MajorTask() is shown in Figure 3.13.

```
void MajorTask (void)
{
    /* Set the clock tick */
    OS_ENTER_CRITICAL();
    PC_VecSet(0x08, OSTickISR); /* Install uC/OS-II's clock tick ISR
    PC_SetTickRate(OS_TICKS_PER_SEC);/*Reprogram tick rate
    OS_EXIT_CRITICAL();
    ……
    PreTickCount = OSTimeGet();
    While (true) {
        TickCount = OSTimeGet();
        If (TickCount != PreTickCount){
            FrameCount %= 40; /*40 Hz task loop
            40 Hz task
            If (FrameCount % 4 == 0){    /*10Hz task loop
                10 Hz task
            }
            ……
            FrameCount ++;
            PreTickCount = TickCount;
        }
    }
}
```

Figure 3.15 Pseudo code for MajorTask()

The reason for using COM1, COM2 and COM3 as the task's name is because the flight control system is a computing system with multi-channels communicating with different sensors and actuators of the flight vehicle. And the data coming into the flight control computer is handled by the interrupt mechanism.

According to the system requirement, the GPS, payload data and manual &

autopilot mode command are all transmitted to FCC in 10Hz rate separately, they can be considered as periodic tasks. On the other hand, the Navigation & Return Home Mode Command is command-driven. So it is an aperiodic task.

To implement the interrupt service routine (ISR) under *u*C/OS-II, the function OSIntEnter() should be called. The pseudo code for an ISR is shown as Fig. 3.14.

YourISR:
Save all CPU registers;
Call ***OSIntEnter()****;*
Execute user code to service ISR;
Call ***OSIntExit();***
Restore all CPU registers;
Execute a return from interrupt instruction;

Figure 3.16 ISRs under *u*C/OS-II

3.2.2 Evaluation

When we reexamine Chen's original design, COM1Task, COM2Task and COM3Task are all activated by the interrupted devices, so they have higher priorities than the MajorTask. Because *u*C/OS-II allows interrupt nesting, the COM3Task can interrupt all other tasks at anytime. On the other way, the MajorTask has the lowest priority, so it can be interrupted and the deadline is not guaranteed.

Basically, the above priority order comes from the limitation of the hardware instead of the system requirement. Meanwhile, the frequency for the GPS, payload data

and GCS command are all based on the individual equipments, which don't synchronize with the FCS. From the viewpoint of FCS, all the data coming from interrupted devices will all be aperiodic tasks. So the real situation will contradict the designer's original idea. And the operation becomes unpredictable.

In Chen's Design, the task's execution rate is controlled by the parameter frame count number, which comes from the tick rate increment. The critical issue to guarantee such system operate as expectation and not crash is in the design of every task's load. It depends on the system programmers' experience and lots of try-and-error processes in the development period.

The most serious problem for Chen's design is that the MajorTask should have the highest priority among all the tasks instead of COM3Task. It is because that the MajorTask does the most important computation for the whole system. It converts the input data and executes the control law computation to give the air vehicle best control. Its deadline should be guaranteed and not interrupted during execution. But for the original design of *u*C/OS-II, even we set the MajorTask with highest priority; we still cannot guarantee it not interrupted by other tasks. When the software module accesses the shared data, it is dangerous if the code is not protected. Because the *u*C/OS-II use different mechanism to handle the interrupt service routine and task level scheduling. And the general task does not get any protection from any outside interruptions.

So, we make use of the features of the revised *u*C/OS-II to set the real-time property for every task. In our example, the MajorTask should be defined as hard real-time task to guarantee its execution. The function, type, priority order and real-time

property are revised as Table 3.3.

Table 3.3 Tasks of the simplified flight control system built on the revised *u*C/OS-II

Task Name	Main Function	Working Rate	Task Type	Priority Order	Real-time Property
Major Task	● Input signal computation ● Control law computation ● Output control	40Hz	Periodic	**1**	**Hard real-time**
COM1Task	Receive the GPS data	10Hz	Periodic	**3**	**Non hard real-time**
COM2Task	Receive the payload feedback data	10Hz	Periodic	**4**	**Non hard real-time**
COM3Task	Receive flight command: ● Manual & Autopilot Mode ● Navigation & Return Home Mode	 ● 10Hz —	 ● Periodic ● Aperiodic	**2**	**Non hard real-time**

We can compare the tasks' timing relationship of the original and the revised design as shown in Figure 3.17 and 3.18.

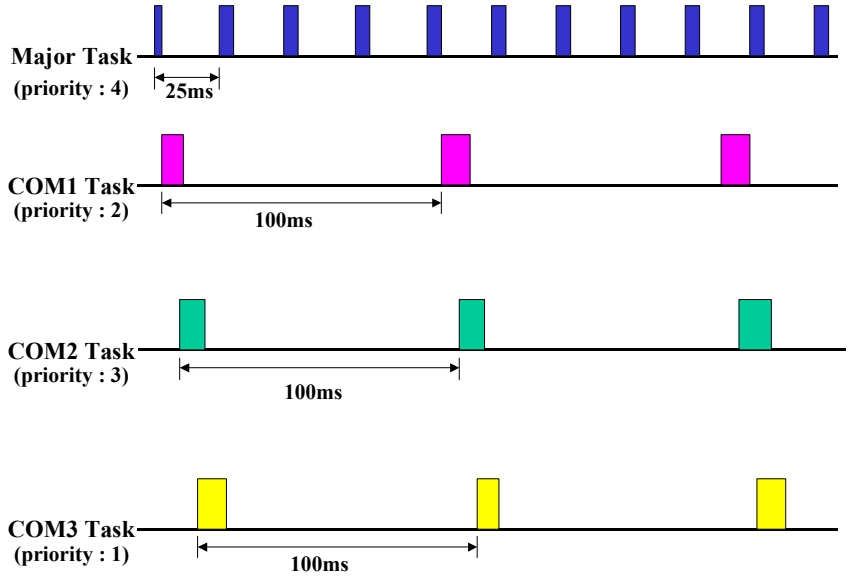

Figure 3.17 The tasks' timing relationship of the original design

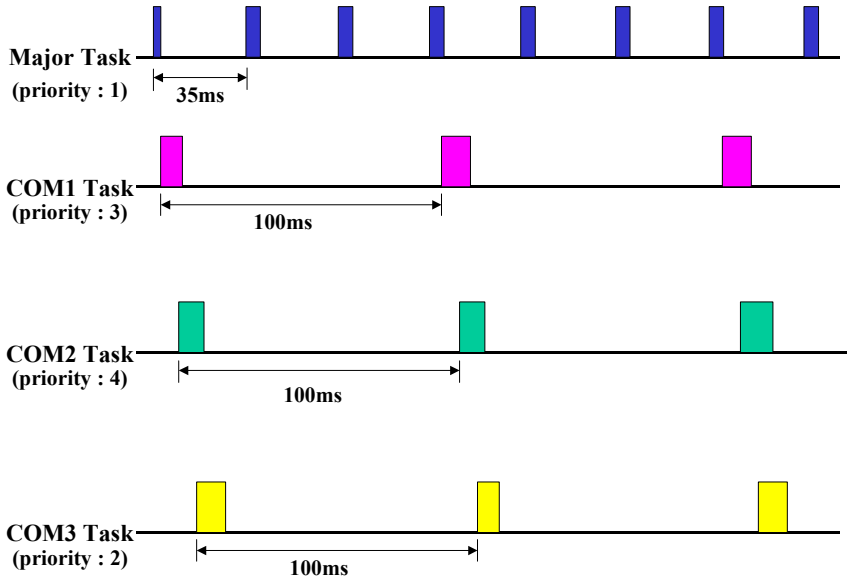

Figure 3.18 The tasks' timing relationship of the revised design

From the above figure, we can see that all the four tasks do not synchronize in the system initialization. So it is difficult to guarantee the deadline requirement during run time. For the original design, the COM3 task is the only one can get deadline guarantee

because it has the highest priority. For other tasks, there is no deadline guarantee and make the system not meet the requirement of hard real-time and safety critical.

Based on our method, we change the task's priority first to let the operation criteria make sense. The hard real-time task can complete its job before deadline. And changing its period, other tasks can also have the fair chance to finish their jobs on time. After several iteration and continuous evaluation, we can fine-tune the system to operate smoothly and meet the design requirements.

3.3 Design Criteria

Unlike Windows or Linux programming which all come along a vast community, *u*C/OS-II does not support a convenient programming environment for system developers. So, using the *u*C/OS-II as the basis to build a real-time system deserves the designers to pay much attention to some design criteria.

We revise the *u*C/OS-II to guarantee the hard real-time tasks complete their jobs within deadline by forcing the kernel enter the critical section. But, using *u*C/OS-II macros to disable and enable interrupts, the interrupt latency should be considered. Disabling interrupts too long will affect the system's response to interrupts. So based on our revised *u*C/OS-II, the more hard real-time tasks are defined, the longer the system's response to interrupts. It is the system designer's responsibility to define the appropriate number of hard real-time tasks.

On the other way, although the tasks have been grouped into hard real-time and non hard real-time, they still have different priorities. The system designer has to arrange the

task's priorities carefully. The basic rule is that hard real-time tasks should always have higher priorities than non-hard real-time tasks.

To use the deadline parameter as a checking point during the critical section, the designers should test the execution time of an individual task. It will involve many individual module tests to get the timing estimation.

Finally, the whole system's performance should also be considered. In order to guarantee all the hard real-time tasks to complete their jobs, the execution time for all tasks should be pre-calculated to evaluate the system's performance. On some conditions, high performance hardware and dual processors may help to meet the system's strict real-time requirements.

For a real-time system, multi-task arrangement is always the programmer's first choice. The system designers have to build appropriate data structure for shared data. Also, even supported by a powerful and stable operating system, the application code for every task should be designed carefully to prevent the system from crash.

3.4 Integrated Development and Verification Requirements

For hard real-time systems, especially safety critical systems, the test to verify the design criteria is very important. The verification is not just to check if the system's function really meets the requirement, but also gains the end users' confidence. Sometimes, it has to demonstrate the whole test process to the authorities that are responsible for the safety issue.

There is always practical to have a complete integrated environment to develop a

hard real-time system for the industry. To perform debugging and testing of the software such as the FCS OFP, a development environment including the Flight Control Test Station (FCTS) and Six Degree-freedom Simulator (SDS) was built and provided. The development is PC-based and the architecture is shown as Figure 3.19.

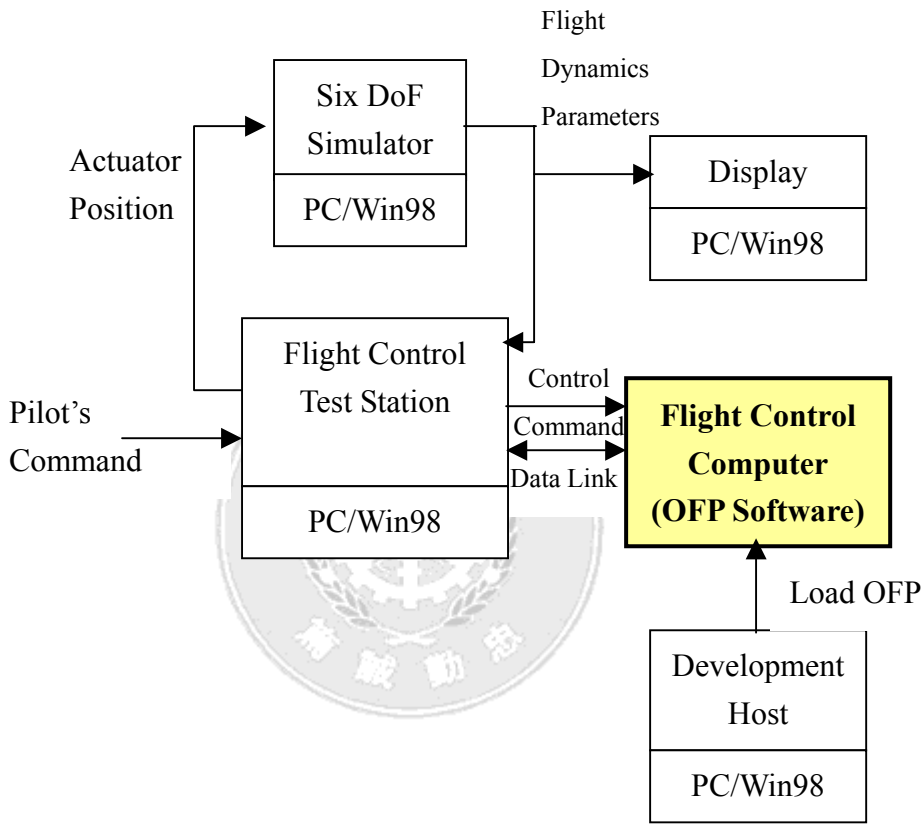

Figure 3.19 Development Environment for FCC OFP

Based on the development environment as Figure 3.19, the flight status of the FCC can be simulated and all the required parameters can be monitored on time. The different operational modes of the air vehicle, which are manual, autopilot, navigation and return home mode, can all be tested. There is a specific test procedure to conduct the software and hardware integration test for the FCC and there are also different test cases designed to verify all the functions of the FCC OFP Software.

We assume that both the hardware and application level software have been correctly designed. Our goal is just transfer the existing system to our revised *u*C/OS-II kernel. We have to follow the procedure made by Chen and use the pre-designed development environment to test our new system. The test items are designed to cover all the operation mode of the system as Table 3.4. Due to the busy schedule for the integrated test station of the Flight Control System, we won't be able to test the real FCC OFP, so we scaled down our experiment to a standalone simulation.

Table 3.4 Test items of the FCC OFP built on the revised *u*C/OS-II

Verification and Validation Test	
Item No.	Test Item
1	Manual Mode Test
2	Autopilot Mode Test
3	Navigation Mode Test
4	Return-Home Mode Test

3.5 Experiment Results

In order to verify the feasibility of our proposed design, we use a simplified model to simulate the function of the revised kernel. The simplified model is based on Jean J. Labrosse's example program. The original program is used to demonstrate the *u*C/OS-II's new features, such as the user-defined context switch hook and statistic task hook. We use it as a platform to simulate our proposed functions.

As we mention above, due to the lack of the integrated test station of the Flight Control System, we scaled down our experiment to a standalone simulation. The hardware is a Personal Computer with Intel 550 MHz Pentium III CPU and 64 MB RAM. The *u*C/OS-II kernel is revised and has the ability to guarantee the hard real-time tasks and support the flexibility of reconfiguration. The application run on the revised *u*C/OS-II kernel has multi-tasks but no I/O. The user interface is as Figure 3.20.

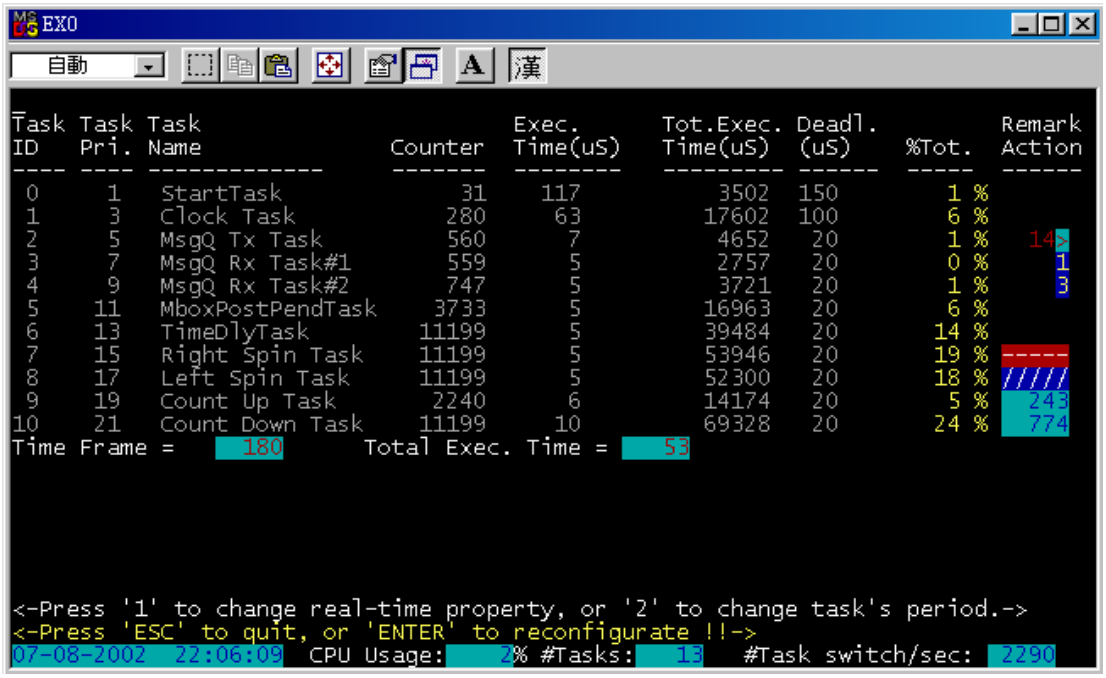

Figure 3.20 The user interface of the experiment simulation

The program structure is just like the sample application mentioned in Chapter 2. There are total 13 tasks in the program, including the idle task, statistic task that are created by the kernel and the other 11 tasks. Among the 11 tasks, the ‘ StartTask’ is responsible for the display of the test result and creation for other tasks. The ‘Clock Task’ gets the system time. So we don’t consider those two task’s behaviors, and set their

priorities higher than the other 9 tasks, which are our experimented targets. We continually monitor all the 9 tasks' execution time, and their deadline. We also accumulate the total execution time and time frame.

Following our proposed solution, the function to guarantee a specific task hard real-time property and to change the task's period of occurrence is designed to be operator-driven. During the simulation program's execution, the operator can change any task's hard real-time property or period at any time. And the whole system's behavior such as other tasks' execution time change can be monitored continuously. On the other way, the adjustment of all the tasks' deadline and priorities is executed automatically. We implemented a mechanism to rearrange the priorities of all the tasks based on the execution time of every task. We let the shortest task run first, i.e. have higher priority. In practice, other scheduling algorithm can also be implemented depending on the system requirements for different applications.

There are three cases to be verified. First, the 9 tasks execute continuously. And in the beginning, we set the deadline of every task a bigger value as shown in Figure 3.20. After the fine adjustment, the deadline for every task is tuned to an optimized value. And the time frame is also set and meets the design requirement. The test result is as shown in Table 3.5.

Table 3.5 Test results #1

Task Name	Original Priority	Revised Priority	Original Deadline	Revised Deadline
MsgQ Tx Task	5	**13**	20	**10**
MsgQ Rx Task#1	7	**9**	20	**5**
MsgQ Rx Task#2	9	**7**	20	**5**
MboxPostPendTask	11	**11**	20	**4**
TimeDlyTask	13	**5**	20	**4**
Right Spin Task	15	**17**	20	**5**
Left Spin Task	17	**15**	20	**5**
Count Up Task	19	**21**	20	**5**
Count Down Task	21	**19**	20	**7**

After adjustment, the time frame is controlled to 50 and the total execution time is always less than the deadline limit.

There is a situation when the time frame is required to be smaller than the recommended value. And we have to readjust the time frame to meet the tasks' requirement. The test result is as shown in Table 3.6.

Table 3.6 Test results #2

Task Name	Original Priority	Revised Priority	Original Deadline	Revised Deadline
MsgQ Tx Task	5	**13**	5	**5**
MsgQ Rx Task#1	7	**11**	5	**6**
MsgQ Rx Task#2	9	**9**	5	**4**
MboxPostPendTask	11	**7**	5	**3**
TimeDlyTask	13	**5**	5	**4**
Right Spin Task	15	**17**	5	**5**
Left Spin Task	17	**15**	5	**5**
Count Up Task	19	**21**	5	**7**
Count Down Task	21	**19**	5	**6**

The original required time frame is 45. After we change the time frame to 48, the timing requirement is met.

Sometimes, if the designers insist to maintain a predefined time frame, and even we readjust the time frame, it still cannot meet the requirements, than we have to make a suggestion to the designers to reconsider the timing arrangement.

All the test results show that an application with multi-tasks built on our revised *u*C/OS-II kernel meets the functional and operational requirements of hard real-time and reconfiguration.

Chapter 4

Conclusions and Future works

Developing safety critical systems is a real challenge for all the system designers. The essential requirement is to guarantee the hard real-time tasks to meet their deadlines. Building an embedded hard real-time system on the *u*C/OS-II kernel has many advantages such as open source code, compact and modulated.

We use an existing kernel *u*C/OS-II to implement new features to support the hard real-time system. In order to guarantee the hard real-time requirement of the system, we revise the scheduling mechanism of the kernel. For our approach, the programmers need to separate their tasks into two groups: hard real-time tasks and non-hard real-time tasks. Although the hard real-time tasks can be guaranteed to meet their deadlines, the system designers have to determine the task's property very carefully. Too many hard real-time tasks can degrade the system's overall performance.

Although the *u*C/OS-II is simple, well documented and came with source code, very appropriate in building the embedded system, it still has many shortcomings. First of all is the lack of graphic and windows support. So it can be used as an embedded control system and not appropriate for the current personal information appliances.

On the other hand, when the system designers use the *u*C/OS-II as the platform to build their application, there is a big chance that they have to face more new problems than their original target problems. The reason is that there are still no enough support for system development, such as device drivers and different processors porting problems. It still needs more and more support to strengthen the community. If the author can make

the *u*C/OS-II to be more commercial, an IDE (Integrated Development Environment) will give the system designers more help.

Often, to build safety critical systems needs to meet the strict certification. The *u*C/OS-II has already been certifiable for use in safety critical systems. Supported by a software testing company Validated Software Corporation in USA, the *u*C/OS-II Validation Suite which contains source codes with complete software design documents, test plans, test procedures and test reports has been certified to the civil aircraft software standard RTCA DO-178B Level B by the Federal Flight Agency of USA in July, 2000. This makes the reliability of *u*C/OS-II more convincible.

From our design, we make the whole system operate in a predictable way. It is essential to design a hard real-time and safety critical system. The approach we proposed can really be implemented in an existing system. But the performance of the system needs be further evaluated. Moreover, the fault tolerance mechanism is also an open issue for the future research.

References

[1] Burns, A. and Wellings, A.J., "Real-Time Systems and their Programming Languages," Addison-Wesley, 1990

[2] N. C. Audsley A. Burns M. F. Richardson A. J. Wellings, "Incorporating Unbounded Algorithms Into Predictable Real-Time Systems," 1993

[3] http://www.ucos-ii.com/, Web site of *u*C/OS-II

[4] Victor Yodaiken, "The RTLinux Manifesto"

[5] Jean J. Labrosse, "MicroC/OS-II The Real-Time Kernel," Prentice Hall Inc, 1996

[6] Martin Timmerman and Jean-Christophe Monfret, "Windows NT as Real-Time OS?," Real-Time Magazine, Feb. 1997

[7] http://www.ucos-ii.com/prod.htm, Web site of *u*C/OS-II

[8] Iain John Bate, "Scheduling and Timing Analysis for Safety Critical Real-Time Systems," Department of Computer Science, University of York, 1998

[9] Victor Yodaiken and Michael Barabanov, "RTLinux Version 2," 1999

[10] Yu-Chung Wang and Kwei-Jay Lin, "Enhancing the real-time capability of the Linux kernel," 1998

[11] Robert Hill, Balaji Srinivasan, Shyam Pather and Douglas, "Temporal Resolution and Real-Time Extensions to Linux," 1998

[12] http://www.lynx.com/, Web site of LynxOS

[13] http://www.qnx.com/, Web site of QNX

[14] Pablo J. Rogina and Gabriel Wainer, "New Real-Time Extensions to the MINIX operating system," August, 1999

[15] Wu-Fong Chen, "A Real-Time Flight Control Embedded System Based on *u*C/OS-II Kernel," 2000

感謝詞

能於工作之餘順利完成資訊工程研究所的課業及研究論文，特別要感謝指導教授鍾葉青老師的諄諄教導，讓我能從學習中建立獨立研究與解決問題的能力，而所內各個領域學有專精的老師們，則拓展了我資訊工程的豐富知識，另外，還要感謝工作上的長官及同仁給我的鼓勵，研究所畢業僅僅是人生一個小階段的結束，未來，又是一個新的開始，回首來時路，謹對一路上幫助我的師長、同事、家人及好友，表達個人無限的感謝。

作者簡介

姓　名：張傳鑫

生　日：民國 54 年 10 月 28 日

出生地：新竹市

學　歷：新竹民富國小

新竹成德國中

新竹高中

中正理工學院機械工程學系

逢甲大學資訊工程研究所

經　歷：國防部中山科學研究院助理研究員