A Real-Time Operating System for Flight Control

Preprint · June 2018

CITATIONS
0

READS
1,328

1 author:

Ming-Yuan Zhu
CoreTek Systems, Inc., Beijing, People's Republic of China
140 PUBLICATIONS **405** CITATIONS

SEE PROFILE

A Real-Time Operating System for Flight Control

Ming-Yuan Zhu

CoreTek Systems, Inc.
1109 CEC Building
6 South Zhongguancun Street
Beijing 100086
People's Republic of China
Email: zhumy@coretek.com.cn

This work is dedicated to the victims of the terrible earthquake disaster in China, 12th of May, 2008

Abstract A typical flight control application consists of periodic software tasks together with a mode-switching logic for enabling and disabling tasks. In this paper, we propose a new model of real-time operating system for flight control applications. A set of system services are specified and two scheduling algorithms are provided in the operating system kernel.

Keywords - Real-time operating systems, flight control systems, embedded software.

1 Introduction

1.1 The Background

We are currently working on a flight control system[11, 9] on a single Intel X86 processor running the real-time operating system **DeltaOS**[12] developed by CoreTek Systems. However, after several successful testing flights, we found that there is a need to design and implement a real-time operating system specifically for flight control applications with the following features and requirements:

- All the tasks in the system are periodic tasks.
- The scheduling algorithm should be non-preemptive and static.
- There is no synchronization and communication mechanisms such as semaphore, message queue and event provided in the system. This means that all the tasks in the system are independent tasks.
- It should reflect our programming practice and experience on flight control applications.

We have seen a number of domain-specific real-time operating system standards in the market such as **OSEK/VDX OS** focusing on automotive control and **ARINC-653** for avionics. However, there is currently neither a commercial real-time operating system designed and implemented for flight control purposes nor a standard focus on this specific area. We have therefore seen the possibility to establish a fundamental framework toward to this direction.

1.2 A Typical Fly-By-Wire Flight Control System

This real-time operating system kernel is designed specifically for embedded flight control applications. Consider a typical fly-by-wire flight control system, which consists of three types of interconnected components: sensors, CPUs for computing control laws, and actuators.

For the cruise mode of the fly-by-wire flight control system, the sensors include

- An inertial measurement unit (IMU), for measuring linear acceleration and angular velocity.
- A global positioning system (GPS), for measuring position.
- An air data measurement system, for measuring such quantities as air pressure.
- The pilot's controls, such as the pilot's stick.

Each sensor has its own timing properties: the IMU, for example, outputs its measurement 1,000 times per second; whereas the pilot's stick outputs its measurement only 500 times per second.

Three separate control laws – for pitch, lateral, and throttle control – need to be computed.

The system has four actuators: two for the ailerons, one for the tailplane, and one for the rudder.

Beside the cruise mode, there are four additional modes: the take-off, landing, autopilot, and degraded modes. In each of these modes, additional sensing tasks, control laws, and actuating tasks need to be executed, as well as some of the cruise task removed. For example, in the take-off mode, the landing gear muse be retracted. In the autopilot mode, the control system takes inputs from a supervisory flight planner, instead of from the pilot's stick. In the degraded mode, some of the sensors or actuators have suffered damage; the control system compensates by not allowing maneuvers which are as aggressive as those permitted in the cruise mode.

1.3 The Contribution

The contribution of this paper is to present a new model of real-time operating system specially designed for flight control applications. This is perhaps the first attempt in this domain-specific area.

1.4 About This Paper

In the Section 1 we describe the backstage for which the paper was written. The Section 2 provides the definition of system service calls in both syntax and semantics. The Section 3 present two scheduling algorithms which can be used by the real-time operating system kernel for flight control. The Section 4 makes some summaries and gives a discussion about future works.

2 The System Service Calls

The set of application programming interfaces (APIs) provided is extremely simple. There are totally four system service calls defined accordingly.

2.1 Tasks

A task `t` has a set of input ports and a set of output ports. The input ports of `t` are distinct from all other ports in the program. The output ports of `t` may be shared with other tasks as long as the tasks are not invoked in the same mode. In general, a task may have an arbitrary number of input and output ports. A task may also maintain a state, which can be viewed as a set of private ports whose values are inaccessible outside the task. Finally, the task has a function `f` from its input ports and its current state to its output ports and its next state. The task function `f` is implemented by a sequential program, and can be written in an arbitrary programming language. It is important to note that the execution of `f` has no internal

synchronization points and cannot be terminated prematurely; in flight control programs all synchronization is specified explicitly outside of tasks. For a given platform, there will be a need to know the worst-case execution time of `f` on each available CPU.

There will be two ways to group the tasks:

- The tasks are classified as the sensing tasks, actuating tasks and the tasks for computing the control laws.
- Each task is constructed into three parts: the part for reading sensors, the part for computation of control laws, and the part for writing actuators.

2.2 Modes

A flight control program consists of a set of modes, each of which repeats the invocation of a fixed set of tasks. The program is in one mode at a time. Possible transitions from a mode to other modes are specified by mode switches. A mode switch can remove some tasks, and add others.

2.3 Creation of Tasks

2.3.1 Task Creation for Multiple-Rate and Single-Cycle Scheduling

To create a task we need to specify the name of task, the frequency, the indexes for which the task to be invoked for the specific cycle, and finally the body of the task which is defined as a procedure.

```
procedure TASKCreate(
                     task_id          : in out Identifier;
                     task_frequency   : in     Nat;
                     cycle_index_list : in     List_Of_Nat;
                     procedure task_body(formal_parameter_list)
                    );
```

There is a condition for `cycle_index_list` which requires that the length of `cycle_index_list` must be equal to `task_frequency`.

2.3.2 Task Creation for Multiple-Rate and Multiple-Cycle Scheduling

To create a task we need to specify the name of task, the frequency, and finally the body of the task which is defined as a procedure.

```
procedure TASKCreate(
                     task_id        : in out Identifier;
                     task_frequency : in     Nat;
                     procedure task_body(formal_parameter_list)
                    );
```

2.4 Creation of Modes

To create a mode we need to specify the name of mode, the period, and the list of tasks contained in the mode. There is no differences in between the multiple-rate and single-cycle scheduling and the multiple-rate and multiple-cycle scheduling.

```
procedure MODECreate(
```

```
    mode_id   : in out Identifier;
    period    : in     Nat;
    task_list : in     List_Of_Identifier
  );
```

2.5 Stop of Tasks

This system service will force a task to be stopped which is normally the last statement in any task body.

```
procedure TASKStop(task_id : in Identifier);
```

2.6 Scheduling of Modes

This system service will force a mode to be scheduled as the active mode.

```
procedure MODESchedule(mode_id : in Identifier);
```

This system service call can be used in the beginning of the program for starting a mode or used in the case of re-scheduling a mode.

In the beginning of the work, we have created two system services: `MODEStart` and `MODESwitch`, after finishing to write the semantic definitions for these two system services, we found out that they are completely the same. So we have changed the design to have only one system service: `MODESchedule`.

3 The Scheduling Algorithms

A flight control system is not an interrupt-driven system. It will collect signals from sensor systems such as radar systems during a defined time period and process it regularly. So it can be called a clock-driven system.

Definition 3.1 *The following terminologies are defined:*

- *The period* `p` *of a mode is defined as a duration in which all the task in the mode must be executed at least once.*
- *The frequency* `f` *of a task is defined as the execution time during a period of a mode.*
- *The rate* `r` *of a task is defined as* `r` = `p` / `f`.
- *The minimal rate* `mr` *of a mode is defined as the minimal rate for all tasks inside of the mode.*
- *The cycle of task* `t` *is defined as the minimal rate of* `t`. *The cycle number* `c` *of task* `t` *is defined as the maximal frequency of mode* `m` *divided by the frequency of* `t`. *The maximal cycle number of mode* `m` *is defined as the maximal frequency of* `m` *divided by the minimal frequency of* `t` *which is equal to the maximal frequency of* `m`.

3.1 Multiple-Rate and Single-Cycle Scheduling

The multiple-rate and single-cycle scheduling makes the following additional assumptions:

- All the tasks with arbitrary rate will always terminate within a minimal cycle.

For instance if a mode `A` contains eight tasks and its execution period is 120ms, the frequency of each task contained in `A` may be 8, 4, 2, and 1, then the cycle of `A` is 120ms / 15 = 8. All the tasks with different rate number must be executed within 15ms. So the multiple-rate and single-cycle scheduling is a typically non-preemptive and static scheduling algorithm.

A running task will be sized its execution by the following cases:

1. It is terminated by executing a task stop command.
2. It was running out of its time slice and was trigged by an interrupt issued by the system timer.
3. It was rescheduled by issuing a mode schedule command.

Since there are three scheduling points in the system, the scheduling algorithm will be divided into three part: `schedule1`, `schedule2` and `schedule3`.

3.1.1 Task Control Blocks (TCBs)

A task control block (TCB) is a system data structure allocated and maintained by the kernel for each task after it has been created. A TCB contains everything the kernel needs to know about a task.

A TCB is defined as a tuple $\langle \texttt{id}, \texttt{f}, \texttt{cf}, \texttt{l}, \texttt{cl}, \texttt{c} \rangle$ of six components:

- `id` is the name of task.
- `f` is frequency of task. It will be used for initialization purpose and will never be changed.
- `cf` is the current frequency of task.
- `l` is the index list of task to indicate the rate number for which the task to be active. `l` will be used for initialization purpose and will never be changed.
- `cl` is the current index list of task to the current applicable rate number for which the task to be active.
- `c` is the task code with context.

3.1.2 Mode Control Blocks (MCBs)

A MCB is allocated to a mode when it is created. This structure contains the mode's name, a list of tasks, the period for activation of the mode, the maximal frequency of task in the list of tasks and the location pointing to the current active task.

A MCB is defined as a tuple $\langle \texttt{id}, \texttt{tl}, \texttt{p}, \texttt{f}, \texttt{c} \rangle$ of five components:

- `id` is the name of mode.
- `tl` is the list of tasks contained in the mode. It will be used for initialization purpose and will never be changed.
- `p` is the period of mode.
- `f` is the maximal frequency of tasks contained in the mode.
- `c` is location pointing to the current task list in the mode.

3.1.3 Predefined Subroutines in Run-Time Environment

The following subroutines are used as pre-defined in the run-time environments:

```
procedure save_continuation(c : in out Continuation);

procedure invoke_continuation(c : in out Continuation);

function get_mcb(i : in Identifier) return Mcb;
function get_tcb(i : in Identifier) return Tcb;

procedure set_mcb(i : in Identifier; mcb : in Mcb);
procedure set_tcb(i : in Identifier; tcb : in Tcb);
```

Where we assume that the type `Continuation` stands for a piece of executable together with its run-time context. We also define the construction functions `mk_tcb` and `mk_mcb` for constructing TCBs and MCBs respectively.

3.1.4 Auxiliary Procedures

The following procedure is used for scheduling the idle task in the system.

```
procedure switch_idle_task is
...
begin
  tcb := get_tcb(idle_task_name);

  tname     := tcb.name;
  tfreq     := tcb.freq;
  tcfreq    := tcb.cfreq;
  tposlist  := tcb.poslist;
  tcposlist := tcb.cposlist;
  tcont     := tcb.cont;

  current_task_name := idle_task_name;

  invoke_continuation(tcont)
end;
```

The following procedures are used for resetting the list of tasks to its initial values.

```
procedure reset_task(id : in Identifier) is
...
begin
  tcb := get_tcb(id);

  tname     := tcb.name;
  tfreq     := tcb.freq;
  tcfreq    := tcb.cfreq;
  tposlist  := tcb.poslist;
  tcposlist := tcb.cposlist;
```

```
  tcont     := tcb.cont;

  tcb := mk_tcb(tname, tfreq, tfreq, tposlist, tposlist, tcont);

  set_tcb(id, tcb)
end;

procedure reset_task_list(idl : in List_Of_Identifier) is
...
begin
  while not is_empty_mlist(idl) loop
    tid := idl.head;
    idl := idl.tail;
    reset_task(tid)
  end loop;
end;
```

3.1.5 Scheduling After Task Terminated Normally

The first case is that a task terminates its execution by executing a task stop statement. In this case, the scheduler simply checks the current task list. If it is empty, then the all the periodic task which must be executed in the cycle have been completed and the idle task is picked up for consuming the rest of time in the cycle. If it is not empty, the task in the top of list is selected according to its index list:

1. If the index list is empty, it means that the task must be executed every cycle, then the task is selected for execution immediately.
2. If the index list is not empty, we must check if the top of current index list is coincided with the current mode frequency then the task is selected for execution immediately; if, however, the top of current index list is not coincided with the current mode frequency, then current index list is updated by its tail list and we continue to check from the beginning again.

```
procedure do_schedule1(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  while not is_empty_mlist(mcurtask) loop
    tcb       := get_tcb(mcurtask.head);
    mcurtask  := mcurtask.tail;

    mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, nmcurtask);
    set_mcb(mname, mcb);

    tname     := tcb.name;
    tfreq     := tcb.freq;
    tcfreq    := tcb.cfreq;
    tposlist  := tcb.poslist;
```

```
    tcposlist := tcb.cposlist;
    tcont     := tcb.cont;
    if is_empty_plist(tposlist) then
      tcb := mk_tcb(tname, tfreq, tcfreq - 1, tposlist, tcposlist, tcont);
      set_tcb(tname, tcb);
      current_tcb_name := tname;
      goto invoke_label
    else
      tpos := tcposlist.head;
      if tpos = current_mode_freq then
        tcposlist := tcposlist.tail;
        tcb       := mk_tcb(tname,
                            tfreq,
                            tcfreq - 1,
                            tposlist,
                            tcposlist,
                            tcont);
        set_tcb(tname, tcb);
        current_tcb := tcb;
        goto invoke_label
      end if;
    end if;
  end loop;
  switch_idle_task;
  return;
  invoke_label: invoke_continuation(tcont)
end;

procedure schedule1 is
...
begin
  mcb      := get_mcb(current_mode_name);
  mcurtask := mcb.curtask;
  if is_empty_mlist(mcurtask) then
    switch_idle_task
  else
    do_schedule1(mcb)
end;
```

3.1.6 Scheduling After a Regular Time Out

The second case is that a regular time out happens. There will be two cases:

1. The end of period for the execution of current mode is reached.
2. A cycle has been completed.

Restart for a New Period

If the period for the execution of current mode is reached, we will simply start the execution of mode for a new period. All the status of tasks contained in the mode will be reset to its initial values. The current mode frequency is reset to its maximal frequency.

```
procedure do_schedule21(mcb : in out Mcb) is
...
```

```
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  mtlist   := reset_task_list(mtlist);
  mcurtask := mtlist;

  mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
  set_mcb(mname, mcb);
  current_freq := mmaxfreq;

  schedule22(mcb)
end;

procedure schedule21(mcb : in out Mcb) is
...
begin
  do_schedule21(mcb)
end;
```

Restart for a New Cycle

If a cycle has been completed, the scheduler simply checks the current task list. If it is empty, then the all the periodic task which must be executed in the cycle have been completed and the idle task is picked up for consuming the rest of time in the cycle. If it is not empty, the task in the top of list is selected according to its index list:

1. If the index list is empty, it means that the task must be executed in every cycle, then the task is selected for execution immediately.
2. If the index list is not empty, we will check whether the current index list is empty or not.
   (a) If the current index list is empty, it means that the task to be checked has completed its execution for this specific cycle, then we will go to the beginning of the loop for the next task in the task list in the mode.
   (b) If the current index list is not empty, we must check whether or not the top of current index list is coincided with the current mode frequency. If the answer is yes, then the task is selected for execution immediately and current index list is updated by its tail list; if, however, the top of current index list is not coincided with the current mode frequency, then we continue the loop by checking the next task in the task list of mode.

```
procedure do_schedule22(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;
```

```
  if is_empty_mlist(mcurtask) then
    switch_idle_task
  else
    tcb      := get_tcb(mcurtask.head);
    mcurtask := mcurtask.tail;

    mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
    set_mcb(mname, mcb);

    current_mode_freq := current_mode_freq - 1;

    tname     := tcb.name;
    tfreq     := tcb.freq;
    tcfreq    := tcb.cfreq;
    tposlist  := tcb.poslist;
    tcposlist := tcb.cposlist;
    tcont     := tcb.cont;

    if is_empty_plist(tposlist) then
      tcb := mk_tcb(tname, tfreq, tcfreq - 1, tposlist, tcposlist, tcont);
      set_tcb(tname, tcb)
      current_task_name := tname;
      invoke_continuation(tcont)
    else
      if is_empty_plist(tcposlist) then
        do_schedule22(mcb)
      else
        tpos := tcposlist.head;
        if tpos = current_mode_freq then
          tcposlist := tcposlist.tail;
          tcb := mk_tcb(tname, tfreq, tcfreq - 1, tposlist, tcposlist, tcont);
          set_tcb(tname, tcb);
          current_task_name := tname;
          invoke_continuation(tcont)
        else
          do_schedule22(mcb)
        end if
      end if
    end if
end;

procedure schedule22(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  mcurtask := mtlist;

  mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
  set_mcb(mname, mcb);
```

```
  current_mode_freq := current_mode_freq - 1;
  do_schedule22(mcb)
end;
```

The Scheduling Procedure

If the current mode frequency is equal to 0, then we need to reset the MCB of the current mode.

```
procedure do_schedule2(mcb : in out Mcb) is
...
begin
  if current_mode_freq = 0 then
    schedule21(mcb)
  else
    schedule22(mcb)
  end if
end;
```

If the current running task is the idle task, then call `do_scheduler2`; otherwise an exception is issued for running out of time.

```
procedure schedule2 is
...
begin
  mcb := current_mcb;
  if current_task_name = idle_task_name then
    do_schedule2(mcb)
  else
    raise run_out_of_time
  end if
end;
```

3.1.7 Scheduling for Mode Switching

The third case is that the system was rescheduled by issuing a mode schedule command. In this case the new mode specified will be set as the active mode and the tasks contained in the mode will be reset to its initial status. Meanwhile the current mode frequency, the current mode name and the timer will be set to new values.

```
procedure do_schedule3(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  mtlist   := reset_task_list(mtlist);
  mcurtask := mtlist;

  mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
  set_mcb(mname, mcb);
```

```
  current_mode_freq := mmaxfreq;
  current_mode_name := mname;
  set_timer(mperiod / mmaxfreq);
  do_schedule22(mcb)
end;

procedure schedule3(mcb : in out Mcb) is
...
begin
  do_schedule3(mcb)
end;
```

3.2 Multiple-Rate and Multiple-Cycle Scheduling

The multiple-rate and multiple-cycle scheduling makes no assumption on the tasks with arbitrary rate terminating within a cycle. For instance if a mode `A` contains eight tasks and its execution period is 120ms, then the cycle of `A` is 15ms. The rate of each task contained in `A` may be 8, 4, 2, and 1. Only the tasks with rate number 8 must be executed within 15ms. The tasks with rate number 4 must be executed within two concatenated cycles - 30ms; the tasks with rate number 2 must be executed within 4 concatenated cycles and the tasks with rate number 1 can be executed within 8 concatenated cycles. So the multiple-rate and multiple-cycle scheduling is a preemptive and static scheduling algorithm.

3.2.1 Task Control Blocks (TCBs)

The TCBs for multiple-rate and multiple-cycle scheduling is different from its counterpart in multiple-rate and single-cycle. A TCB is defined as a tuple $\langle \texttt{id}, \texttt{f}, \texttt{cf}, \texttt{e}, \texttt{c}, \texttt{cc} \rangle$ of six components:

- `id` is the name of task.
- `f` is frequency of task. It will be used for initialization purpose and will never be changed.
- `cf` is the current frequency of task.
- `e` is the cycle number for which the task to be active.
- `c` is the initial task code with context.
- `cc` is the current task code with context.

3.2.2 Mode Control Blocks (MCBs)

Again a MCB is defined as a tuple $\langle \texttt{id}, \texttt{tl}, \texttt{p}, \texttt{f}, \texttt{c} \rangle$ of five components:

- `id` is the name of mode.
- `tl` is the list of tasks contained in the mode. It will be used for initialization purpose and will never be changed.
- `p` is the period of mode.
- `f` is the maximal frequency of tasks contained in the mode.
- `c` is location pointing to the current task list in the mode.

The MCBs defined here is completely as same as its counterparts in multiple-rate and single-cycle scheduling.

3.2.3 Auxiliary Procedures

The following procedures are used for resetting the list of tasks to its initial values.

```
procedure reset_task(id : in Identifier) is
...
begin
  mcb := get_tcb(current_mode_name);

  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  tcb := get_tcb(id);

  tname  := tcb.name;
  tfreq  := tcb.freq;
  tcfreq := tcb.cfreq;
  tcycle := tcb.cycle;
  tcont  := tcb.cont;
  tccont := tcb.ccont;

  tcb := mk_tcb(tname, tfreq, tfreq, mmaxfreq / tfreq, tcont, tcont);

  set_tcb(id, tcb)
end;

procedure reset_task_list(idl : in List_Of_Identifier) is
...
begin
  while not is_empty_mlist(idl) loop
    tid := idl.head;
    idl := idl.tail;
    reset_task(tid)
  end loop;
end;
```

3.2.4 Scheduling After Task Terminated Normally

A task is terminated by executing a task stop command. This the normal termination of a task.

Saving of Current Context

The semantic subroutine `save_conrrent1` is invoked for saving the current context. If the current frequency of task is 0, it means that the task has complete its execution, then the cycle number of task is set to 0; otherwise the cycle number is reset to the maximal frequency of mode divided by the frequency of task.

```
procedure save_current1 is
...
begin
```

```
  mcb := get_tcb(current_mode_name);

  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  tcb := get_tcb(current_task_name);

  tname  := tcb.name;
  tfreq  := tcb.freq;
  tcfreq := tcb.cfreq;
  tcycle := tcb.cycle;
  tcont  := tcb.cont;
  tccont := tcb.ccont;

  tcfreq := tcfreq - 1;
  if tcfreq = 0 then
    tcb := mk_tcb(tname, tfreq, tcfreq, 0, tcont, tcont)
  else
    tcb := mk_tcb(tname,
                  tfreq,
                  tcfreq,
                  mmaxfreq / tfreq,
                  tcont,
                  tcont)
  end if;
  set_tcb(current_task_name, tcb)
end;
```

Checking of Schedulability

A task is schedulable if `cur_mode_freq` $=$ `mcycle` $\cdot$ (`tcfreq` $-$ 1) $+$ `tcycle` $\cdot$ 1, where `mcycle` is defined as `mmaxfreq`/`tfreq` and 1 for one execution.

```
function schedulable1(tcb : Tcb) return boolean is
...
begin
  mcb := get_tcb(current_mode_name);

  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  tname  := tcb.name;
  tfreq  := tcb.freq;
  tcfreq := tcb.cfreq;
  tcycle := tcb.cycle;
  tcont  := tcb.cont;
  tccont := tcb.ccont;
```

```
  mcycle := mmaxfreq / tfreq;

  return current_mode_freq = (tcfreq - 1) * mcycle + tcycle
end;
```

Scheduling Next Ready Task

The semantic subroutine `schedule_next1` is used for scheduling the designated task which satisfying the scheduling condition as the next running task.

```
procedure schedula_next1(tcb : in out Tcb) is
...
begin
  tname  := tcb.name;
  tfreq  := tcb.freq;
  tcfreq := tcb.cfreq;
  tcycle := tcb.cycle;
  tcont  := tcb.cont;
  tccont := tcb.ccont;

  tcycle := current_mode_freq / tfreq;

  tcb := mk_tcb(tname, tfreq, tcfreq - 1, tcycle, tcont, tccont);
  set_tcb(current_task_name, tcb);
  current_task_name := tname;

  invoke_continuation(tccont)
end;
```

The Scheduling Procedure

The semantic subroutine `do_schedule1` is iteratively defined on the current task list. If it is empty then the idle task will be scheduled for consuming the rest of running time. If it is not empty then we will check if the task on the top of current task list is schedulable or not, if it is schedulable then the task will be scheduled, otherwise `do_schedule1` will continue to check the rest in the current task list.

```
procedure do_schedule1(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  while not is_empty_mlist(mcurtask) loop
    save_current1;
    tcb      := mcurtask.head;
    mcurtask := mcurtask.tail;

    mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
    set_mcb(mname, mcb);

    if schedulable1(tcb) then
```

```
      schedule_next1(tcb);
      return
    end if
  end loop;
  save_current1;
  switch_idle_task
end;

procedure schedule1 is
...
begin
  mcb := get_mcb(current_mode_name);
  do_schedule1(mcb)
end;
```

3.2.5 Scheduling After a Regular Time Out

The system was running out of its time slice. When an interrupt issued by the system timer, `schedule2` will be invoked. Again there will be two cases to be considered:

1. The period for the execution of current mode is reached.
2. A cycle has been completed.

Restart for a New Period

For the first case the semantic subroutine `schedule21` will be invoked.

Checking of Schedulability

Again a task is schedulable if $\texttt{cur\_mode\_freq} = \texttt{mcycle} \cdot (\texttt{tcfreq} - 1) + \texttt{tcycle} \cdot 1$, where `mcycle` is defined as `mmaxfreq`/`tfreq` and 1 for one execution.

```
function schedulable2(tcb : Tcb) return boolean is
...
begin
  mcb := get_tcb(current_mode_name);

  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  tname  := tcb.name;
  tfreq  := tcb.freq;
  tcfreq := tcb.cfreq;
  tcycle := tcb.cycle;
  tcont  := tcb.cont;
  tccont := tcb.ccont;

  mcycle := mmaxfreq / tfreq;
```

```
  return current_mode_freq = (tcfreq - 1) * mcycle + tcycle
end;
```

The definition of `schedulable2` is completely the same as `schedulable1`. But we will still give it a different name since it is used for different purpose of scheduling.

Scheduling Next Ready Task

The function `schedule_next2` is used for selecting the next schedulable task as the running task from the current task list contain in the active mode.

```
procedure schedule_next2(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  while not is_empty_mlist(mcurtask) loop
    tcb      := mcurtask.head;
    mcurtask := mcurtask.tail;

    mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
    set_mcb(mname, mcb);

    current_mode_freq := current_mode_freq - 1;

    tname  := tcb.name;
    tfreq  := tcb.freq;
    tcfreq := tcb.cfreq;
    tcycle := tcb.cycle;
    tcont  := tcb.cont;
    tccont := tcb.ccont;

    tcycle := current_mode_freq / tfreq;

    if schedulable2(tcb) then
      tcb := mk_tcb(tname, tfreq, tcfreq - 1, tcycle, tcont, tccont);
      set_tcb(tname, tcb);
      current_task_name := tname;
      goto invoke_label
    end if
  end loop;
  switch_idle_task;
  return;
  invoke_label: invoke_continuation(tccont)
end;
```

The Scheduling Procedure `schedule21`

The scheduling procedure `schedule21` is then defined as follows:

```
procedure schedule21(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  mtlist   := reset_task_list(mtlist);
  mcurtask := mtlist;
  mcb      := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
  set_mcb(mname, mcb);

  current_mode_freq := mmaxfreq;
  schedule_next2(mcb)
end;
```

Restart for a New Cycle

For the second case the semantic subroutine `schedule22` will be invoked.

Saving of Current Context

If the current cycle of task is equal to 0, then it means that the task has run out of time so an exception will be raised. Otherwise the current run-time context (rest of program) will be saved and the current cycle of task is decreased by 1.

```
procedure save_current2 is
...
begin
  tcb := get_tcb(current_task_name);

  tname  := tcb.name;
  tfreq  := tcb.freq;
  tcfreq := tcb.cfreq;
  tcycle := tcb.cycle;
  tcont  := tcb.cont;
  tccont := tcb.ccont;

  if tcycle = 0 then
    raise run_out_of_time
  else
    save_continuation(c);
    tcb := mk_tcb(tname, tfreq, tcfreq, tcycle - 1, tcont, c);
    set_tcb(current_task_name, tcb)
  end if
end;
```

The Scheduling Procedure `schedule22`

The scheduling procedure `schedule22` is then defined as follows:

```
procedure schedule22(mcb : in out Mcb) is
```

```
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  mcurtask := mtlist;

  mcb := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
  set_mcb(mname, mcb);
  current_mode_freq := current_mode_freq - 1;

  save_current2;
  schedule_next2(mcb)
end;
```

The Scheduling Procedure

If the current mode frequency is equal to 0 then we will need to check the following: if the current task is the idle task, it means that the mode has completed its execution for the whole period, we then will invoke `schedule21`, otherwise the exception of system running out of time will be raised. If the current mode frequency is not equal to 0 then `schedule22` will be invoked.

```
procedure schedule2 is
...
begin
  mcb := get_mcb(current_mode_name);
  if (current_mode_freq = 0) then
    if current_task_name = idle_task_name then
      schedule21(mcb)
    else
      raise run_out_of_time
    end if
  else
    schedule22(mcb)
  end if
end;
```

3.2.6 Scheduling for Mode Switching

The mode is rescheduled by issuing a mode schedule command. In this case the selected mode is retrieved and with the following actions:

1. All the tasks in the mode is reset to its initialized values.
2. The selected mode become the active mode.
3. The current mode name and maximal frequency are reset to its current mode values.
4. The timer is reset.
5. A new task is scheduled for execution.

```
procedure do_schedule3(mcb : in out Mcb) is
...
begin
  mname    := mcb.name;
  mtlist   := mcb.tlist;
  mperiod  := mcb.period;
  mmaxfreq := mcb.maxfreq;
  mcurtask := mcb.curtask;

  mtlist   := reset_task_list(mtlist);
  mcurtask := mtlist in
  mcb      := mk_mcb(mname, mtlist, mperiod, mmaxfreq, mcurtask);
  set_mcb(mname, mcb);

  current_mode_freq := mmaxfreq;
  current_mode_name := mname;

  set_timer(mperiod / mmaxfreq);
  schedule_next2(mcb)
end;

procedure schedule3(mode_name : Identifier) is
...
begin
  mcb := get_mcb(mode_name);
  do_schedule3(mcb)
end;
```

4 Conclusions

We have presented a real-time operating system designed specifically for flight control systems. The system service calls provided in kernel are easy to associate with any programming languages such as **C**, **C++** and **Ada** to form a real-time operating system for flight control applications.

4.1 Other Possible System Services

4.1.1 Start Mode from Selected Task

Other possible system services may be defined such as

```
procedure MODEScheduleFrom(
                           mode_id : in Identifier;
                           task_id : in Identifier
                          );
```

In stead of starting a task from the beginning of the list of tasks contained in the mode `mode_id`, we may start the specific task `task_id` in the list of tasks contained in the mode. So `MODESchedule` is a special case of `MODEScheduleFrom`.

4.1.2 Period Changing in Mode

We can adjust the period defined in a mode by replacing a new one using the following system service call.

```
procedure MODEPeriodChange(
                           mode_id : in Identifier;
                           period  : in Nat
                          );
```

4.1.3 Task Set Changing in Mode

We can replace the task set defined in a mode by a new one using the following system service call.

```
procedure MODETaskSetChange(
                            mode_id   : in Identifier;
                            task_list : in List_Of_Identifier
                           );
```

This can be used for updating the task list by inserting a task in the head of the task list, the tail of the task list, or the task with name `t` in the task list; or by deleting a task in the head of the task list, the tail of the task list, or the task with name `t` in the task list; or by shifting the task list from left to right or from right to left; or by reordering the task list.

4.1.4 Changing Frequency of Task

We can replace the frequency defined in a task by a new one using the following system service call.

```
procedure TASKFreqChange(
                         task_id   : in Identifier;
                         frequency : in Nat
                        );
```

4.1.5 Changing Index List of Task

For the multiple-rate and single-cycle scheduling we also need a system service for changing the cycle index list of task.

```
procedure TASKIndexChange(
                          task_id          : in Identifier;
                          cycle_index_list : in List_Of_Nat
                         );
```

The system services `TASKFreqChange` and `TASKIndexChange` are required in conjunction with `MODESchedule`. If two modes with different periods use the same task then we have to modify the frequency and cycle index list of the task when mode `m1` switch to `m2`. For instance, if mode `m1` with period 120ms and mode `m2` with period 60ms use the same task `t` in the task list and the frequency of `t` for `m1` is 8, then the frequency of `t` for `m2` must be 4. Since `t` has the different frequency for `m1` and `m2`, it must have different cycle index list too.

An alternative way to handle this dynamic situation is to define the rate number for each task instead of frequency, the frequency can be calculated by using the period of mode and the rate number of task. The cycle index list should be defined in the mode creation instead of the task creation.

4.1.6 Changing Task Body

We can replace the task body defined in a task by a new one using the following system service call.

```
procedure TASKBodyChange(
                          task_id : in Identifier;
                          procedure task_body(formal_parameter_list)
                         );
```

4.1.7 System Services for Memory and Storage Management

In addition there will be a set of system services required for memory and storage management used for flight control applications.

4.1.8 Background Tasks

Typical System Services for Background Tasks

In the scheduling algorithms described in Section 3, we have used idle task as the backstage task when there is a need for consuming the rest of time in a cycle. In addition we may declare a set of tasks as the backstage tasks.

```
procedure BackgroundTASKCreate(
                               task_id : in Identifier;
                               procedure task_body(formal_parameter_list)
                              );

procedure BackgroundTASKStart(task_id : in Identifier);

procedure BackgroundTASKStop(task_id : in Identifier);

procedure BackgroundTASKDelete(task_id : in Identifier);
```

There will be two options about scheduling the backstage tasks:

- There should be only one backstage task as the active task if we allow the task body to be non-terminating (loop forever). Otherwise we will need a scheduler for scheduling the backstage tasks.
- There should be a list of backstage tasks. Each of them will terminate within a finite steps. In this case, we can select (schedule) the task inside of the list one by one.

Background Tasks as Operating System Kernel

An extreme case is that to make all the backstage tasks as an ordinarily general purpose real-time operating system kernel such as **DeltaOS** with priority and round-robin scheduling. In this case we will form a two-level scheduling real-time operating system.

It is interesting to notice that when integrating this operating system kernel with a partition-based operating system[8], we will naturally obtain a three-level scheduling real-time operating system.

4.2 Considerations for Implementation

As it has been pointed out in the last section, it will be relatively easy to implement the model in programming language **C** to form a new real-time operating system specially for flight control applications. Let us call it **DeltaFliCOS**. **DeltaFliCOS** will be completely different from other kernels in the **DeltaOS** family.

4.3 The Interface between Giotto and DeltaFliCOS

There will be a natural connection in between **Giotto**[2, 3] and this real-time operating system kernel in the sense that it will be easy to translate a **Giotto** program into the program written in **C** and the system service calls provided in **DeltaFliCOS**.

4.4 Schedulability Analysis

Let us consider the case of periodic tasks. For periodic scheduling, the best that we can do is to design an algorithm which will always find a schedule if one exists. A scheduler is defined to be optimal iff it will find a schedule if one exists.

Rate-monotonic (RM) scheduling[7, 4, 5] is probably the most well-known scheduling algorithm for independent periodic tasks. It is our conjecture that the multiple-rate and single-cycle schedulability as well as the multiple-rate and multiple-cycle schedulability are equivalent to the schedulability of rate-monotonic scheduling.

4.5 Untouched Topics

4.5.1 An Integrated Scheduling Algorithm

The two scheduling algorithm may be integrated together to form a new scheduling algorithm.

4.5.2 I/O Processing

We don't know at the moment if we need a special language treatment for I/O processing in the area of flight control. The I/O processing units related with flight control applications are sensors, actuators and other communication devices.

4.5.3 Fault Tolerance

The fault tolerance is also an important topics. But again we don't know if we need a special language treatment and how we do it. It seems that there is a lot of works to be done in the area.

4.5.4 Prediction of Execution Times

Scheduling of tasks requires some knowledge about the duration of task executions, especially if meeting time constraints has to be guaranteed, as is in real-time systems. The worst-case execution time[1, 6, 10] is the basis for most scheduling algorithms. Computing such a bound is undecidable in the general case. This is obvious from the fact that it is undecidable whether or not a program terminates. Hence, the WCET can only be computed for certain programs/tasks. For example, for programs without recursion and while-loops and with constant iteration counts, the WCET can be computed. We suspect that the calculating of WCET may become easier when we restrict ourselves in certain specific areas such as the flight controls.

References

[1] C. Healy, R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding pipeline and instructions caches performance. *IEEE Transactions on Computers*, 1999.

[2] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto: A Time-Triggered Language for Embedded Programming. In *Lecture Notes in Computer Scienece 2211*. Proceedings of EMSOFT, Springer-Verlag, 2001.

[3] C. M. Kirsch, M. A. A. Sanvido, T. A. Henzinger, and W. Pree. A Giotto Based Helicopter Control System. In *Lecture Notes in Computer Science 2491*. Proceedings of Second International Workshop on Embedded Software (EMSOFT), Springer Verlag, 2002.

[4] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of IEEE Real-Time Systems Symposium*, page 166C171. IEEE Computer Society Press, December 1989.

[5] J. P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proceedings of IEEE Real-Time Systems Symposium*, pages 201–209. IEEE Computer Society Press, December 1990.

[6] S. Lim, J. H. Han, J. Kim, and S. L. Min. A worst case timing analysis technique for multiple-issue machines. In *Proceedings of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[7] C. Liu and J. Layland. Scheduling algorithm for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[8] Lei Luo and Ming-Yuan Zhu. Partitioning based operating system: A formal model. *ACM Operating Systems Review*, 37(3), 2003.

[9] Mauro Marinoni, Tulliao Facchinetti, Giorgio Buttazzo, and Gianluca Franchino. An embedded real-time system for autonomous flight control. In *Proceedings of 50th International Congress of ANIPLA on Methodologies for Emerging Technologies in Automation (ANIPLA 2006)*, Rome, November 2006.

[10] C. Y. Park. *Predicting Deterministic Execution Times of Real-Time Programs*. PhD thesis, University of Washington, August 1992.

[11] B. L. Stevens and F. L. Lewis. *Aircraft Control and Simulation*. John Wiley & Son, 1992.

[12] M.-Y. Zhu, Lei Luo, and Guang-Ze Xiong. A provably correct operating system: δ-**CORE**. *ACM Operating Systems Review*, 35(1), 2001.