Programmation temps réel

Les services

Dimitry SOLET

2020 - 2021

Plan

Plan

Tasks

Tasks are "active" elements of the application

2 categories of tasks exist in OSEK/VDX :

- basic tasks ;
- extended tasks (introduced later in the chapter).

A basic task is a sequential code that should terminate (no infinite loop).

Keyword to define a Task

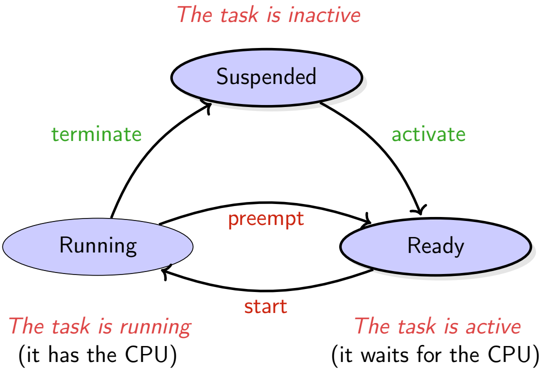
Name of the task

```
TASK(myTask)
{
  //task's code..
  ...
  TerminateTask();
}
```

task's instructions for one job

Call to the service that terminates a task

States of a basic task



- green transition : due to a system call ;
- red transition : due to the scheduler ;
- At startup, task may be either in a `Suspended` or `Ready` state.

OSEK scheduling policy

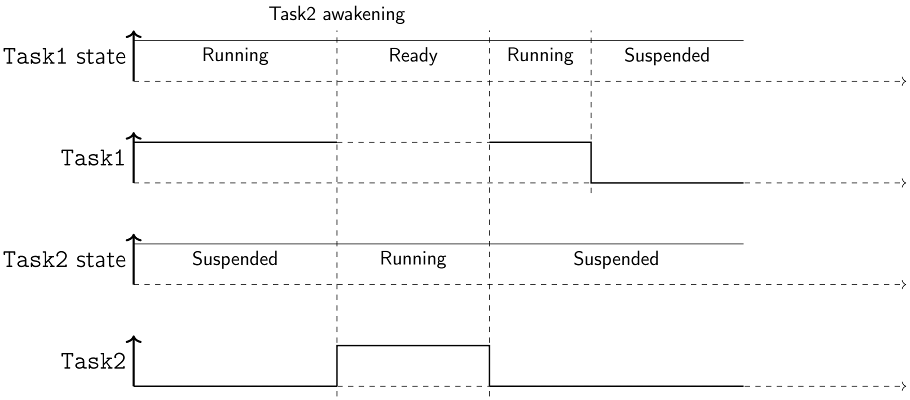
- Scheduling is done *in-line*
  - Scheduling is done dynamically during the execution of the application
- Tasks have a *fixed priority*
  - The priority of a task is given at design stage ;
  - The priority does not change (almost, taking and releasing resources may change the priority) ;
  - No round-robin. If more than one task have the same priority, tasks run one after the other. *i.e.* a task may not preempt a task having the same priority
- Tasks may be *preemptable or not* (almost)
  - This property is defined at design stage.

OSEK scheduling policy

- "*Full preemptive*" : All tasks are preemptable
  - It is the most reactive model because any task may be preempted. The highest priority Task is sure to get the CPU as soon as it is activated.
- "*Full non preemptive*" : All tasks are non-preemptable.
  - It is the most predictive model because a task which get the CPU will never be preempted. Scheduling is a straightforward and the OS memory footprint may be smaller.
- "*Mixed*" : Each task may be configured as preemptable or non-preemptable.
  - It is the most flexible model.
  - For instance, a very short task (in execution time) may be configured as non-preemptable because the context switch is longer than its execution.

Scheduling modes

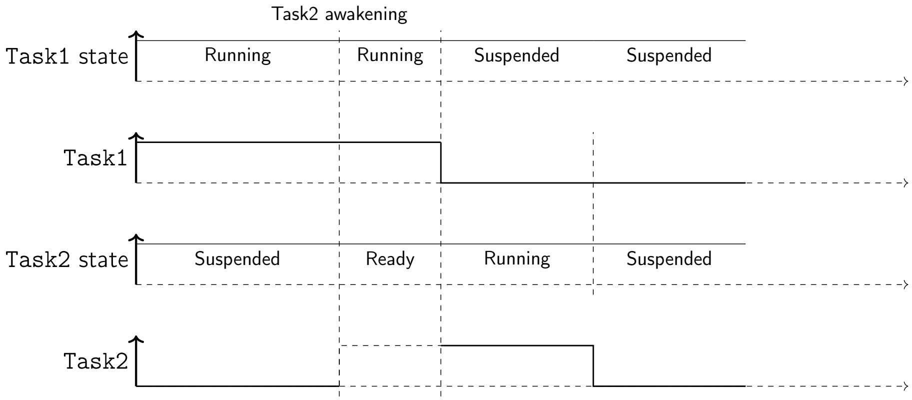
Example : 2 tasks (`Task1` and `Task2`).
At start, `Task1` runs. Then `Task2` is activated.



Prio(`Task1`) = 5 and Prio(`Task2`) = 10. *Full preemptive mode*

Scheduling modes

Example : 2 tasks (`Task1` and `Task2`).
At start, `Task1` runs. Then `Task2` is activated.



Prio(`Task1`) = 5 and Prio(`Task2`) = 10. *Full non-preemptive mode*

Tasks' services - TerminateTask

TerminateTask service :

- `StatusType TerminateTask(void);`
- StatusType is an error code :

E_OK	: no error
E_OS_RESOURCE	: the task hold a resource ;
E_OS_CALLEVEL	: the service is called from an interrupt ;

```
TASK(myTask)
{
  //task's code..
  ...
  TerminateTask()
}
```

- The service stops the calling task.
  The task goes from running state to suspended state.
- A task may not stop another task ! (like the `kill` system call on POSIX).
  - forgetting to call TerminateTask may crash the application (and maybe the OS) !

Tasks' services - ActivateTask

`ActivateTask` service :

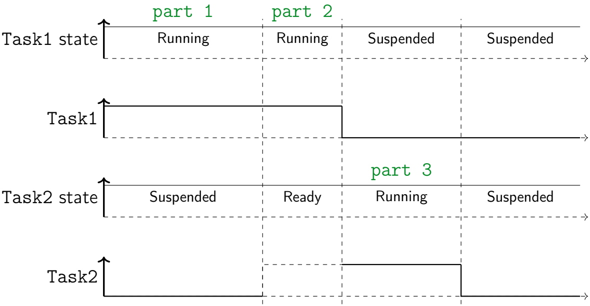
- `StatusType ActivateTask(TaskType <TaskId>);`
- The argument is the id of the task to activate
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ID`	: invalid TaskId (no task with such an id) ;
`E_OS_LIMIT`	: too many activations of the task

- *This service puts the task `<TaskId>` in ready state* :
    - If the activated task has a higher priority, the calling task is put in the ready state. The new one goes in the running state.
    - A scheduling may be done (preemptable task or not, called from an interrupt).

Tasks' services - ActivateTask

Example : 2 tasks (`Task1` and `Task2`).
`Task1` is active et start of the application (`AUTOSTART` parameter)

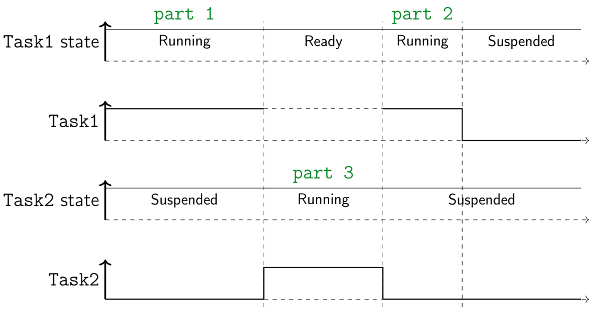



```
TASK(Task1)
{
  ... //part 1
  ActivateTask(Task2);
  ... //part 2
  TerminateTask();
}

TASK(Task2)
{
  ... //part 3
  TerminateTask();
}
```

Prio(`Task1`) $\geq$ Prio(`Task2`)

Tasks' services - ActivateTask

Example : 2 tasks (`Task1` and `Task2`).
`Task1` is active et start of the application (`AUTOSTART` parameter)

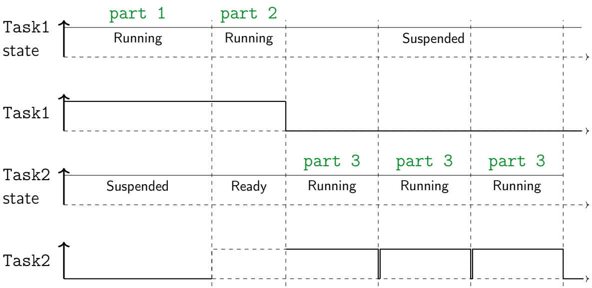



```
TASK(Task1)
{
  ... //part 1
  ActivateTask(Task2);
  ... //part 2
  TerminateTask();
}

TASK(Task2)
{
  ... //part 3
  TerminateTask();
}
```

Prio(`Task1`) $<$ Prio(`Task2`)

Tasks' services - ActivateTask

When multiple activations occur, OSEK allows to memorize them up to a value defined at design time.




```
TASK(Task1)
{
  ... //part 1
  ActivateTask(Task2);
  ActivateTask(Task2);
  ActivateTask(Task2);
  ... //part 2
  TerminateTask();
}

TASK(Task2)
{
  ... //part 3
  TerminateTask();
}
```

Tasks' services - ChainTask

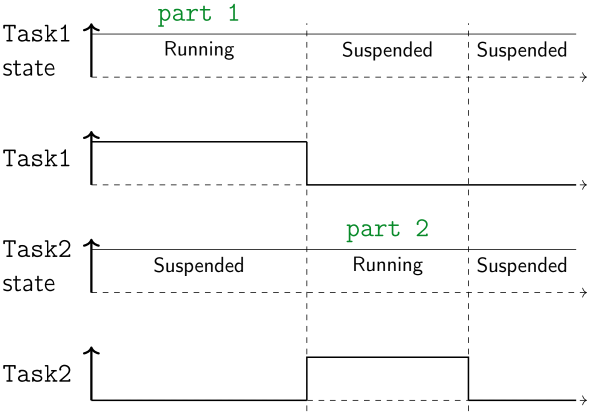
ChainTask service :

- `StatusType ChainTask(TaskType <TaskId>);`
- The argument is the id of the task to activate
- `StatusType` is an error code :

  `E_OK` : no error
  `E_OS_ID` : invalid TaskId (no task with such an id);
  `E_OS_LIMIT` : too many activations of the task
- This service *puts the task <TaskId> in ready state*, and *the calling task in the suspended state*.
  - This service replaces TerminateTask for the calling task.

Tasks' services - ChainTask

Example : 2 tasks (`Task1` and `Task2`).
`Task1` is active et start of the application (`AUTOSTART` parameter)




```
TASK(Task1)
{
  ... //part 1
  ChainTask(Task2);
}

TASK(Task2)
{
  ... //part 2
  TerminateTask();
}
```

Tasks' declaration

To declare a task in OSEK, we have to give the parameters :

- the fixed priority ;
- preemption mode (preemptive or not, mixed) ;
- task state at startup (suspended or ready) ;
- the max number of activations.

Tasks' services - OIL description

Using Trampoline, task's parameters are split in 2 structures :

- a static one in ROM ;
- a dynamic one in RAM

Declaration of the dynamic task descriptor :

```
/*
 * Dynamic descriptor of task blink
 */
VAR(tpl_proc, OS_VAR) blink_task_desc = {
  /* resources                      */  NULL,
#if WITH_OSAPPLICATION == YES
  /* if > 0 the process is trusted  */  0,
#endif /* WITH_OSAPPLICATION */
  /* activate count                 */  0,
  /* task priority                  */  1,
  /* task state                     */  SUSPENDED
};
```

Tasks' services - OIL description

Declaration of the static task descriptor :

```
/*
 * Static descriptor of task blink
 */
CONST(tpl_proc_static, OS_CONST) blink_task_stat_desc = {
  /* context                   */  blink_CONTEXT,
  /* stack                     */  blink_STACK,
  /* entry point (function)    */  blink_function,
  /* internal ressource        */  NULL,
  /* task id                   */  blink_id,
#if WITH_OSAPPLICATION == YES
  /* OS application id         */
#endif
  /* task base priority        */  1,
  /* max activation count      */  1,
  /* task type                 */  TASK_BASIC,
#if WITH_AUTOSAR_TIMING_PROTECTION == YES

  /* execution budget */        0,
  /* timeframe        */        0,
  /* pointer to the timing
     protection descriptor    */ NULL

#endif
};
```

Tasks' services - OIL description

Declaration of stack and context :

```
/*
 * Task blink stack
 */
#define APP_Task_blink_START_SEC_STACK
#include "tpl_memmap.h"
VAR(tpl_stack_word, OS_APPL_DATA) blink_stack_zone[256/sizeof(tpl_stack_word)];
#define APP_Task_blink_STOP_SEC_STACK
#include "tpl_memmap.h"

#define blink_STACK {blink_stack_zone, 256}

/*
 * Task blink context
 */
#define OS_START_SEC_VAR_NOINIT_32BIT
#include "tpl_memmap.h"
VAR(arm_core_context, OS_VAR) blink_int_context;

#define blink_CONTEXT &blink_int_context

#define OS_STOP_SEC_VAR_NOINIT_32BIT
#include "tpl_memmap.h"
```

And this is only for ONE task !

Tasks' services - OIL description

```
TASK myTask {           //ID of the Task

  PRIORITY = 2;         //Static priority of the task

  AUTOSTART = FALSE;    //State of the task a beginning:
                        //  - READY     if AUTOSTART = TRUE
                        //  - SUSPENDED if AUTOSTART = FALSE

  ACTIVATION = 1;       //maximum memorized activations

  SCHEDULE = NON;       //Scheduling mode:
                        //  - FULL: Task is preemptable
                        //  - NON:  Task is non-preemptable

  STACKSIZE = 256;      //Target specific extension.
                        //Here, the size of the stack
};
```

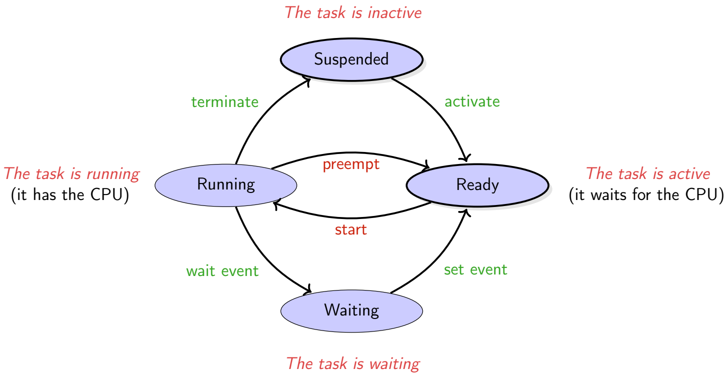
Tasks' services - OIL description

```
TASK myTask {

  PRIORITY = 2;

  AUTOSTART = TRUE {       //if the task is put in READY state
    APPMODE = AppModeStd; //at start, a sub-attribute
  };                       //corresponding to the application
                           //mode has to defined.
  ACTIVATION = 1;

  SCHEDULE = NON;



  STACKSIZE = 256;

};
```

Tasks' synchronization

- Synchronization of tasks : A task should be able to wait an external event (a verified condition).
- To implement this feature, OSEK uses events ;
- Task's model is modified to add a new state : `Waiting`.
  - The task that are able to wait for an event are called *Extended tasks* ;
  - The drawback is a more complex scheduler (a little bit slower and a little increase of code size)

States of an extended task



- green transition : due to a system call ;
- red transition : due to the scheduler ;
- At startup, task may be either in a `Suspended` or `Ready` state.

The concept of event

- An event is like a flag that is raised to signal something just happened
- An event is *private* : It is a property of an *Extended Task*. Only the owning task may wait for the event.
- It is a *N producers / 1 consumer model* :
  - Any task (extended or basic) or Interrupt Service Routines Category 2 (will be explained later) may invoke the service which sets an event.
  - One task (an only one) may get the event (i.e. invoke the service which wait for an event).
- The maximum number of event per task relies on the implementation (32 in Trampoline)

Event Mask

- An Extended Task may wait for many events simultaneously
  - The first to come wakes up the task.
- To implement this feature, an event corresponds to a *binary mask* : `0x01`, `0x02`, `0x04`,...
- An event vector is associated to 1 or more bytes. Each event is represented by one bit in this vector
- So each task owns :
  - *a vector of the events set*
  - *a vector of the events it waits for*

Event Mask

- Operation :

Event X is signaling	`ev_set |= mask_X;`
Is event X arrived ?	`ev_set & mask_X;`
Wait for event X	`ev_wait | mask_X;`
Clear event X	`ev_set &= ~mask_X;`

- In practice, these operations are done in a simpler way by using the following services. . .

but the notion of binary mask should not be ignored !

Events' services - SetEvent

SetEvent service :

- `StatusType SetEvent(TaskType <TaskID>, EventMaskType <Mask>);`
- Events of task `<TaskID>` are set according to the `<Mask>` passed as 2^{nd} argument.
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ID`	: invalid `TaskId` (no task with such an id) ;
`E_OS_ACCESS`	: `TaskID` is not an extended task (not able to manage events) ;
`E_OS_STATE`	: Events cannot be set because the target task is in the `SUSPENDED` state.

- This service is not blocking and may be called from a task or an ISR2.

Events' services - ClearEvent

ClearEvent service :

- `StatusType ClearEvent(EventMaskType <Mask>);`
- The events selected by `<Mask>` are cleared.
- May be called by the *owning* task only (as an event is private).
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ACCESS`	: `TaskID` is not an extended task (not able to manage events) ;
`E_OS_CALL_LEVEL`	: The caller is not a task.

- non-blocking service.

Events' services - GetEvent

GetEvent service :

- `StatusType GetEvent(TaskType <TaskId>, EventMaskRefType event);`
- *Beware : a* `RefType` *is in fact a pointer to a type*.
- The event mask of the task `<TaskId>` is copied to the variable `event` (As pointer to an EventMaskType is passed to the service) ;
- May be called by the *owning* task only (as an event is private).
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ID`	: invalid `TaskId` (no task with such an id) ;
`E_OS_ACCESS`	: `TaskID` is not an extended task (not able to manage events) ;
`E_OS_STATE`	: Events cannot be set because the target task is in the `SUSPENDED` state.

- This service is not blocking and may be called from a task or an ISR2.

Events' services - WaitEvent

WaitEvent service :

- `StatusType WaitEvent(EventMaskType <EventID>);`
- Put the calling task in the WAITING state until one of the events is set
- May be called by the (extended) task that owns the event ;
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ACCESS`	: the calling task is not an extended task (not able to manage events) ;
`E_OS_RESOURCE`	: The task has not released all the resources (explained later).
`E_OS_CALL_LEVEL`	: The caller is not a task.

Events' services - OIL description

```
EVENT ev1 {
   MASK = AUTO; //definition of the MASK computed
};              //by the OIL compiler.
EVENT ev2 {
   MASK = 0x4;  //litteral value which is a binary mask.
};

//myTask is automatically an extended task,
TASK myTask {
  PRIORITY = 2;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  SCHEDULE = NON;
  EVENT = ev1;  //list of events the task uses.
  EVENT = ev2;  //the task is the owner of these events
};
```

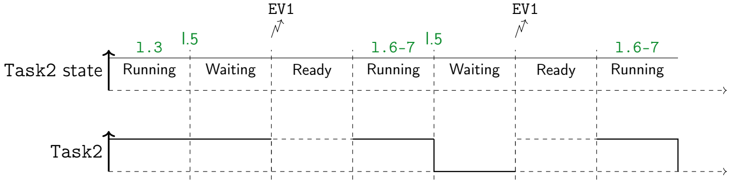
If an event is used in more than one task, only the name is share :

an event is private.

Events' services - Simple exemple with one event

```
TASK(Task1)
{
  //set EV1, which is an
  //event owned by Task2
  SetEvent(Task2, EV1);
  ...
  TerminateTask();
}
```

```
1  TASK(Task2)
2  {
3    ... //task's setup
4    while(1) {
5      WaitEvent(EV1);
6      ClearEvent(EV1);
7      ... //task's job
8    }
9    //never called
10   TerminateTask();
11 }
```



Events' services - Example with 2 events

```c
TASK(Task1)
{
  //set EV1, which is an
  //event owned by Task2
  SetEvent(Task2, EV1);
  ...
  TerminateTask();
}

TASK(Task3)
{
  ...
  SetEvent(Task2, EV2);
  ...
  TerminateTask();
}
```

```c
TASK(Task2)
{
  EventMaskType event_got;
  ...
  while(1) {
    //wait for 2 events
    //simultaneously
    WaitEvent(EV1 | EV2);
    //what event awoke me?
    GetEvent(Task2, &event_got);
    if (event_got & EV1) {
      ClearEvent(EV1);
      //manage EV1
    }
    if (event_got & EV2) {
      ClearEvent(EV2);
      //manage EV2
    }
  }
  TerminateTask();
}
```

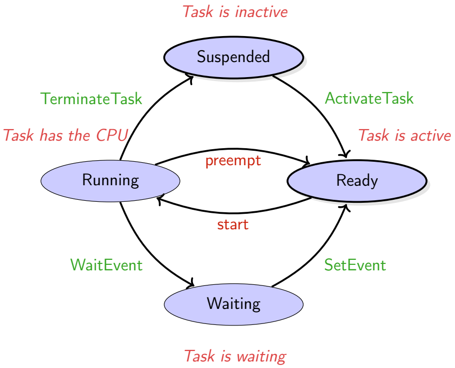
Summary - System Calls

Basic Tasks

- `TerminateTask();`
- `ActivateTask(TaskId);`
- `ChainTask(TaskId);`

Extended Tasks

- `SetEvent(TaskId, Mask);`
- `ClearEvent(Mask);`
- `GetEvent(TaskId, Mask);`
- `WaitEvent(Mask);`



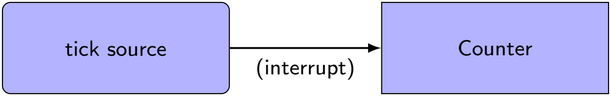
Plan

Counters and alarms

- Goal : *perform an "action" after a number of "ticks" from an hardware device* :
  - Typical case : periodic activation of a task with a hardware timer.
- The "action" may be :
  - signalization of an event (*i.e.* `SetEvent`)
  - activation of a task (*i.e.* `ActivateTask`)
  - function call (a callback since it is a user function). The function is executed on the context of the running task. These is deprecated in AUTOSAR
- The hardware device may be :
  - a timer
  - any periodic interrupt source (for instance an interrupt triggered by the low position of a piston of a motor). The frequency is not constant in this case.

Counters

- The counter is an abstraction of the hardware "tick" source (timer, interrupt source, ...)
  - The "tick" source is heavily dependent of the target platform ;
  - The counter is a standard component ;
  - Moreover, the counter has a divider.



Counters

- A counter defines 3 values :
  - The maximum value of the counter ($MaxAllowedValue$) ;
  - A division factor ($TicksPerBase$) : for instance with a `TicksPerBase` equal to 5, 5 ticks are needed to have the counter increased by 1 ;
  - The minimum number of cycles before the alarm is triggered (explained after) ;
- The counter restarts to 0 after reaching `MaxAllowedValue`

Counters - OIL description

```
COUNTER generalCounter {

  TICKSPERBASE = 10;          //number of "ticks" (from the
                              //interrupt source) needed to
                              //have the counter increased
                              //by one
  MAXALLOWEDVALUE = 65535;    //Maximum value of the counter.
                              //This value is used by the OIL
                              //compiler to generate the size
                              //of the variable used to store
                              //the value of the counter
  MINCYCLE = 128;             //minimum interval between
                              //2 triggering of the alarm
};
```

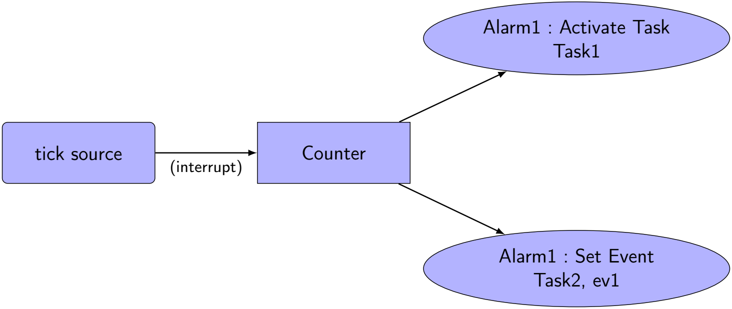
Counters

- At least one counter is available : SystemCounter
- No system call to modify the counters.
  - Their behavior are masked by the alarms.
- A hardware interrupt must be associated to a counter
  - This part is not explained in the standard and depends on the target platform and the OSEK/VDX vendor.
- Default counter for the Trampoline Cortex-M port :

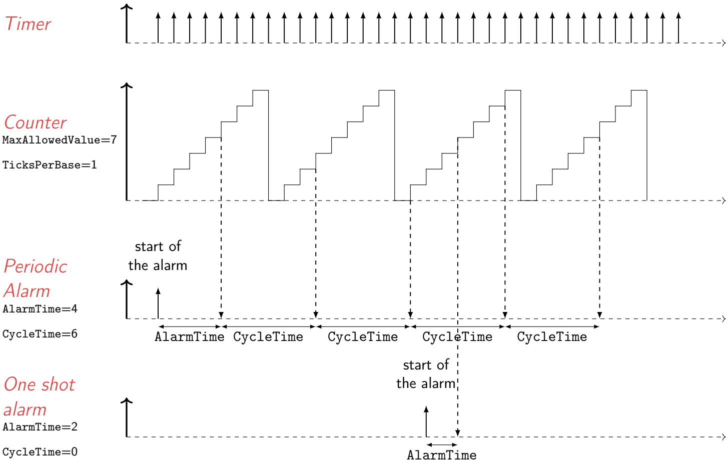
```
COUNTER SystemCounter {
  SOURCE = SysTick; //common timer for each Cortex-M
  TICKSPERBASE = 1;
  MINCYCLE = 1;
  MAXALLOWEDVALUE = 32767;
};
```

Alarms

- An alarm is connected to a counter an performs an action.
  - An alarm is associated to 1 counter ;
  - A counter may be used for several alarms ;
- When the counter reach a value of the alarm (CycleTime, AlarmTime), the alarm expires and an action is performed :
  - Activation of a task ;
  - Signalization of an event ;
  - Function call (callback) - *deprecated*



Alarms - Example



Counters/Alarms

- Counters do not have system calls.
  - They are set up statically and behave that way while the system is up and running.
  - The hardware tick source may be stopped.
- Alarms may be started and stopped dynamically, and have an initial configuration defined statically with an OIL description.

Alarm - OIL description

```
ALARM alarm_1 {
  COUNTER = generalCounter;
  ACTION = ACTIVATETASK {//action to be performed
    TASK = task_1;
  };
  AUTOSTART = TRUE {      //initial configuration
    ALARMTIME = 10;       //alarm triggered at 10 counter ticks
    CYCLETIME = 5000;     //and then each 5000 ticks
    APPMODE = AppModeStd;
  };
};
```

Counters/Alarms - SetAbsAlarm service

- ```
  StatusType SetAbsAlarm (AlarmType <AlarmID>,
                          TickType <start>,
                          TickType <cycle>);
  ```
  - `AlarmID` is the id of the alarm to start (its name in the OIL file);
  - `start` is the *absolute* date at which the alarm expire;
  - `cycle` is the relative date (counted from the start date) at which the alarm expire again. If 0, it is a one shot alarm, like the *cycleTime* in the OIL description.
- StatusType is an error code :

`E_OK`	:	no error
`E_OS_STATE`	:	The alarm is already started;
`E_OS_ID`	:	The AlarmID is invalid
`E_OS_VALUE`	:	start is <0 or >MaxAllowedValue and/or cycle is <MinCycle or >MaxAllowedValue.
  ```

Counters/Alarms - SetRelAlarm service

- ```
  StatusType SetRelAlarm (AlarmType <AlarmID>,
                          TickType <increment>,
                          TickType <cycle>);
  ```
  - AlarmID is the id of the alarm to start (its name in the OIL file);
  - start is the *relative* date at which the alarm expire, like the *alarmTime* in the OIL description.
  - cycle is the relative date (counted from the start date) at which the alarm expire again. If 0, it is a one shot alarm, like the *cycleTime* in the OIL description.
- StatusType is an error code :

E_OK	: no error
E_OS_STATE	: The alarm is already started ;
E_OS_ID	: The AlarmID is invalid
E_OS_VALUE	: start is <0 or >MaxAllowedValue and/or cycle is <MinCycle or >MaxAllowedValue.
  ```

Counters/Alarms - `CancelAlarm` service

- `StatusType CancelAlarm (AlarmType <AlarmID>);`
  - `AlarmID` is the id of the alarm to start (its name in the OIL file) ;
- it stops an alarm.
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_NOFUNC`	: The alarm is not started
`E_OS_ID`	: The AlarmID is invalid

Counters/Alarms - GetAlarm service

- ```
  StatusType GetAlarm (AlarmType <AlarmID>,
                       TickRefType <tick>);
  ```
  - `AlarmID` is the id of the alarm to start (its name in the OIL file) ;
  - `tick` is a pointer to a TickType where GetAlarm store the remaining ticks before the alarm expire.
- Get the remaining (counter) ticks before the alarm expires.
- `StatusType` is an error code :

`E_OK`	:	no error
`E_OS_NOFUNC`	:	The alarm is not started
`E_OS_ID`	:	The AlarmID is invalid
  ```

Counters/Alarms - GetAlarmBase service

- ```
  StatusType GetAlarmBase (AlarmType <AlarmID>,
                           AlarmBaseRefType <info>);
  ```
  - AlarmID is the id of the alarm to start (its name in the OIL file) ;
  - info is a pointer to an AlarmBaseType where GetAlarmBase store the parameters of the underlying counter ;
- Get the parameters of the underlying counter.
- StatusType is an error code :

E_OK	: no error
E_OS_ID	: The AlarmID is invalid
  ```

Plan

Interrupts

- 2 kinds of interrupts (Interrupt Service Routine or ISR) are defined in OSEK, according to the richness needed for the ISR ;
- Anyway, the execution time of an ISR must be short because it delays the execution of tasks.
- Level 1 interrupts
  - are very fast ;
  - stick to the hardware capabilities of the micro-controller ;
  - are *not allowed* to do a system call ;
  - usually difficult to port to another micro-controller ;
- Level 2 interrupts
  - are not as fast as level 1 interrupts
  - are allowed to do some system calls (activate a task, get a resource, ...)

ISR1

- Are not allowed to do system calls ;
- In fact, ISR1 are ignored by the operating system and defined as classical interrupts :
  - Init interrupt registers of the hardware peripheral ;
  - Init the related interrupt mask
  - Do not touch the other interrupt masks (which are managed by the operating system).

ISR2

- May (must ?) do *system calls* (activate a task, get a resource, ...)
- Roughly the same behavior as a task
  - they have a *priority* (greater than the higher priority of tasks). ISR2 priority is a logical one and may not be related to the hardware priority level.
  - they have a *context* (registers, stack, ...)
- In addition an ISR2 :
  - is associated to a hardware interrupt (triggered by an event ;

ISR2

To use an ISR2, it is necessary to

- declare it in the OIL file with the interrupt source identifier (depends on the target platform) to indicate where the interrupt handler is installed ;
- initialize the related interrupt registers of the peripheral which will trigger the interrupt.

Keyword to define an ISR

Name of the task

```
ISR(myISR)
{
  //code of ISR
  ...
}
```

task's instructions for one job

unlike for a task, there is no need for a TerminaterISR at the end

ISR2

```
ISR AppuiBouton {
  CATEGORY = 2;          //Interrupt category (ISR2)
  PRIORITY = 30;         //static priority: Should be ABOVE
                         //the higher priority of tasks
  STACKSIZE = 256;       //target specific extension
  SOURCE = EIC_IRQ {     //specific for the Atmel-SAMD21 µC
    PIN = PA15 {
      TRIGGER = FALLING;
      PULL = UP;
      FILTERING = TRUE;
    };
  };
};
```

Plan

Accessing shared resources

Hardware and software resources may be shared between tasks (an optionally between tasks and ISR2 in OSEK)

- Resource sharing implies a task which access a resource should not be preempted by a task which will access to the same resource.
- This leads to allow to modify the scheduling policy to give the CPU to a low priority task which access the resource while a high priority task which access he same resource is may not run.
    - In some cases, priority inversion may occur ;
    - Deadlocks may occur when the design is bad.
- In OSEK, the *Priority Ceiling Protocol* is used to solve this problem.

Example

Simple example with an unprotected software resource (global variable) :
Task T1 and T2 are executed only once. What is the final value of `val` ?

```
volatile int val = 0;
TASK(T1)
{
  ...
  val++;
  ...
  TerminateTask();
}

TASK(T2)
{
  ...
  val++;
  ...
  TerminateTask();
}
```

Example

Simple example with an unprotected software resource (global variable) :
Task `T1` and `T2` are executed only once. What is the final value of `val` ?

```
volatile int val = 0;

TASK(T1)
{
  ...
  val++;
  ...
  TerminateTask();
}

TASK(T2)
{
  ...
  val++;
  ...
  TerminateTask();
```

```
//l.6 of Task T1 in assembly
//r2 contains the address of 'val'
ldr     r3, [r2, #0]   ; R3 <= 'val'
adds    r3, #1         ; R3++
str     r3, [r2, #0]   ; 'val' <= R3
```

Example

Simple example with an unprotected software resource (global variable) :
Task `T1` and `T2` are executed only once. What is the final value of `val` ?

```
volatile int val = 0;

TASK(T1)
{
  ...
  val++;
  ...
  TerminateTask();
}

TASK(T2)
{
  ...
  val++;
  ...
  TerminateTask();
```

```
//l.6 of Task T1 in assembly
//r2 contains the address of 'val'
ldr     r3, [r2, #0]   ; R3 <= 'val'

//interrupt
// => rescheduling
// => start task T2 (higher priority)
// ...

// and resume to task T1
adds    r3, #1         ; R3++
str     r3, [r2, #0]   ; 'val' <= R3
```

`val` may contain either 2. . . or 1 !

There is non-determinism.

Semaphore

- Proposed by Edsger Dijkstra
- It allows to protect access to shared resources
- This mechanism, available in many OS (not OSEK) offers 3 functions :
  - Init() : initialize the semaphore ;
  - P() to test the semaphore (Probieren) ;
  - V() pour increment the semaphore (Verhogen).

Semaphore

A counter is associated to the semaphore.

- A call to `P()` is used to ask for resource access :
    - If the counter is > 0, it is decremented and the resource may be taken.
    - If the counter is $= 0$, the task which called P() is put in the waiting state until the counter became > 0. At that time the task will be awaken and the counter will be decremented again.
- A call to `V()` is used to release a resource :
    - The counter is incremented and a task which is waiting for the resource may be put in ready state.

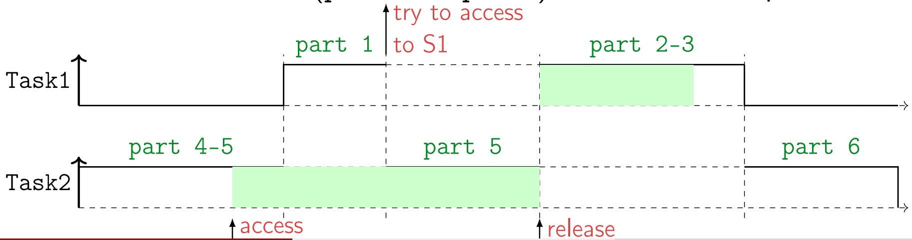
Semaphore

If there is only one resource to protect a *critical section*, it is called a mutual exclusion (*mutex*) :

```
TASK(Task1)
{
  ... //part 1
  semP(S1);
  ... //part 2
  // - critical section! -
  semV(S1);
  ... //part 3
}
```

```
TASK(Task2)
{
  ... //part 4
  semP(S1);
  ... //part 5
  // - critical section! -
  semV(S1);
  ... //part 6
}
```

Critical sections (part2 and part5) *should not* overlap !



Semaphore

The counter associated to the semaphore may have a non- binary value

example : access to a buffer :

- Here are 2 functions to read and write the buffer. These functions may be called by concurrent tasks.
- The S1 semaphore which has the initial value of its counter equal to 5 allows up to 5 writes. After that, a task which try to write is put in the waiting state until a task does a read.

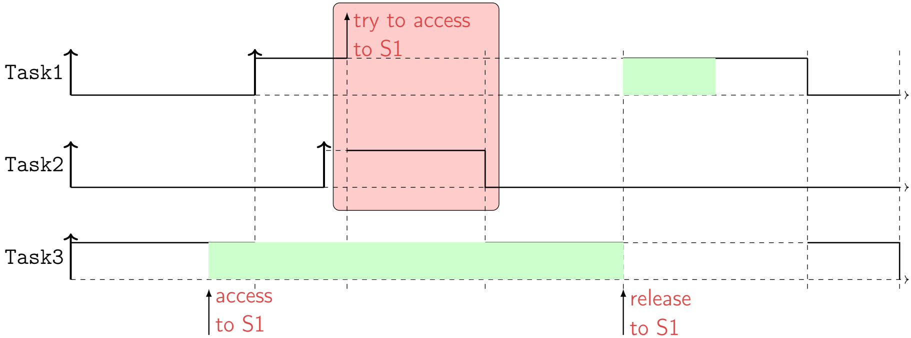
```
void init() {
    semInit(S1, 5);
}

void WriteBuffer(int data) {
  semP(S1);
  //buffer write
}

void ReadBuffer(int *data) {
  //buffer read
  semV(S1);
}
```

This is a protection against a buffer overflow. Another one is required for buffer underflows.

Semaphore - Problem of priority inversion

Classical synchronization mechanism (semaphore, mutex) may have the priority inversion problem :

- A task with a lower priority may delay a higher priority task.
- The following example shows 3 preemptable tasks, T1 has the higher priority :

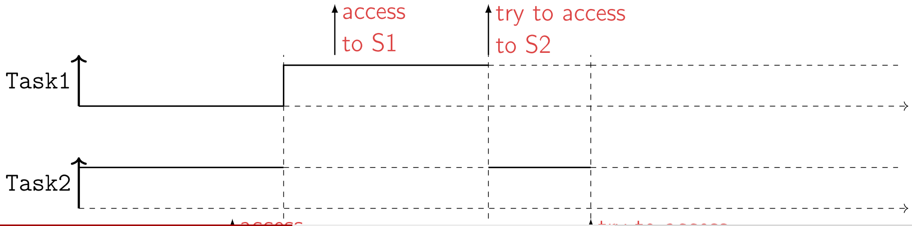


Semaphore - Deadlock

The biggest problem is the deadlock. It's a design problem

```
TASK(Task1)
{
  semP(S1);
  semP(S2);
  // - critical section! -
  semV(S2);
  semV(S1);
}
```

```
TASK(Task2)
{
  semP(S2);
  semP(S1);
  // - critical section! -
  semV(S1);
  semV(S2);
}
}
```



OSEK Resources

- OSEK resources are used to do *mutual exclusion* between several tasks (or ISR2) to *protect the access to a shared hardware or software entity*.
- Example of hardware entity :
  - LCD display ;
  - Communication network (CAN, ethernet, ...).
- Example of software entity :
  - a global variable ;
  - the scheduler access (in this case, the task may not be preempted).
- OSEK/VDX offers a RESOURCE mechanism with 2 associated system calls (to Get and Release a Resource).

A word about shared global variables

- CPU uses registers and variables should temporarily be transfered to these registers (the compiler generates such a code).
- So the memory is not always up to date
- A shared global variable is shared by using the memory. *So its value in memory should always be up to date*.
- In the C language, the `volatile` qualifier tells the compiler to always load and store a global variable from memory instead of working with a a copy that stays in registers.

It forces the compiler not to optimize memory access with the variable.

```
volatile int myGlobalVar;
```

Resources' services - GetResource

GetResource service :

- `StatusType GetResource(ResourceType <ResID>);`
- Take the resource ResID ;
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ID`	: the resource id is invalid ;
`E_OS_ACCESS`	: trying to get a resource that is already in use (it is a design error).

- A task that "owns" the resource may not be preempted by another task that will try to get the resource.
  - ⇒ What about the fixed priority scheduling ?

Resources' services - ReleaseResource

ReleaseResource service :

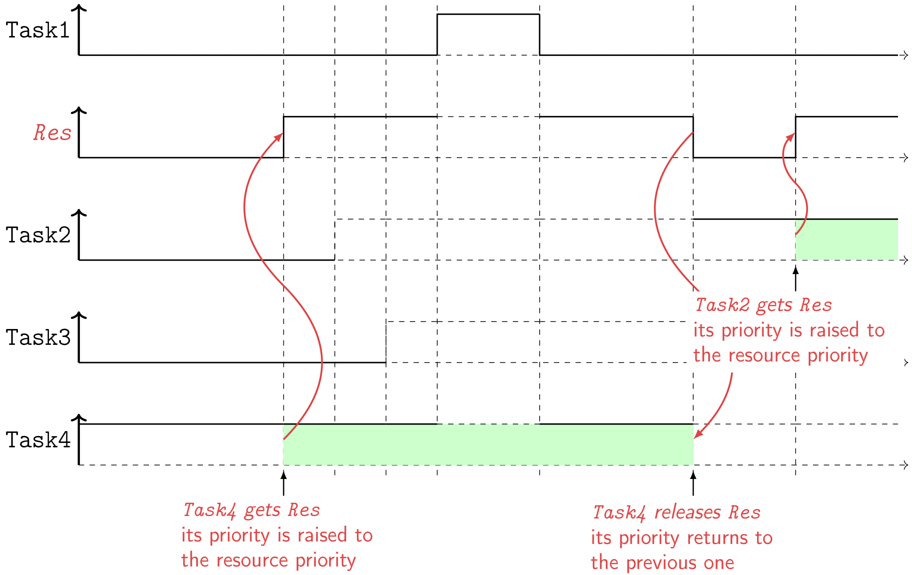
- `StatusType ReleaseResource(ResourceType <ResID>);`
- release the resource ResID ;
- `StatusType` is an error code :

`E_OK`	: no error
`E_OS_ID`	: the resource id is invalid ;
`E_OS_ACCESS`	: trying to get a resource that is already in use (it is a design error).

Resources' services - ReleaseResource

- To take resources into account in scheduling, *IPCP* (Immediate Priority Ceiling Protocol) is used.
- *Each resource has a priority* such that :
  - The priority is $\geq$ to max of priorities of tasks which may get the resource ;
  - When a task gets a resource, its priority is raised to the priority of the resource
  - When a task releases the resource, its priority is lowered to the previous one.

OSEK Resource



OSEK Resource

- `Task1` has a higher priority than the resource. Its behavior is not modified.
- `Task3` has a priority set between the priority of `Task2` and the priority of `Task4`. `Task3` is delayed while `Task4` uses the resource.
- `Task2` is delayed when `Task4` uses the resource but is never delayed by `Task3`

Major improvement

- No priority inversion
- No deadlock possible.

OSEK Resource - Some remarks

- An `ISR2` may take a resource ;
- `Res_scheduler` is a resource that disables scheduling when in use. A task which gets `Res_scheduler` becomes non-preemptable until it releases it ;
- There is no need to get a resource if a task is configured as non-preemptable in the OIL file ;
- A task should get a resource for a time as short as possible. *i.e.* only to access a shared entity because higher priority tasks may be delayed.

OSEK Resource - Exceptions

- if a shared variable is an *atomic one* (`i.e.` the CPU reads or write it with only one assembly instruction, *AND*
- the variable is *written* (and not read) *by only 1 task*, there is no need to get a resource
- if a resource is not needed, an `ISR2` may be replaced by an `ISR1` with better performance.

OSEK Resource - Task group

- This feature allows to mix non-preemptable tasks and preemptable tasks.
  - In the same group all the tasks are seen as non-preemptable by the other tasks of the group.
  - A task having a higher priority than all the tasks of the group (and not part of the group) may preempt any task of the group.
- This feature uses an internal resource for each group.

Internal resource

- The internal resource is got automatically when the task starts to run ;
- The internal resource is released automatically when the task terminates ;
- An internal resource may not be reference with `GetResource()` or `ReleaseResource()`.

OSEK Resource - RES_SCHEDULER

A default internal resource exists :

- RES_SCHEDULER internal resource has a priority equal to the max priority of the tasks.
- Any task declared as non-preemptable is in fact in a task group with the internal resource RES_SCHEDULER.

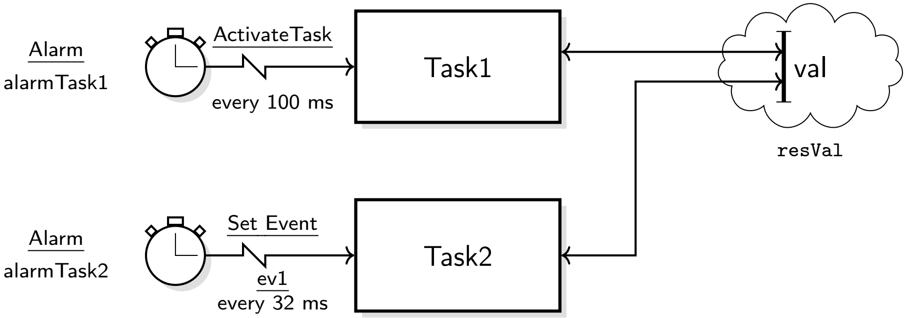
OSEK Resource - OIL description

```
RESOURCE resA {
  //RESOURCEPROPERTY may be STANDARD or INTERNAL.
  //For the latter, the resource is got automatically
  //when the task runs and released automatically
  //when it calls TerminateTask();
  RESOURCEPROPERTY = STANDARD;
};

TASK myTask {
  PRIORITY = 2;
  AUTOSTART = FALSE;
  ACTIVATION = 1;
  SCHEDULE = NON;
  RESOURCE = ResA; //mandatory!
  STACKSIZE = 256;
};
```

Important

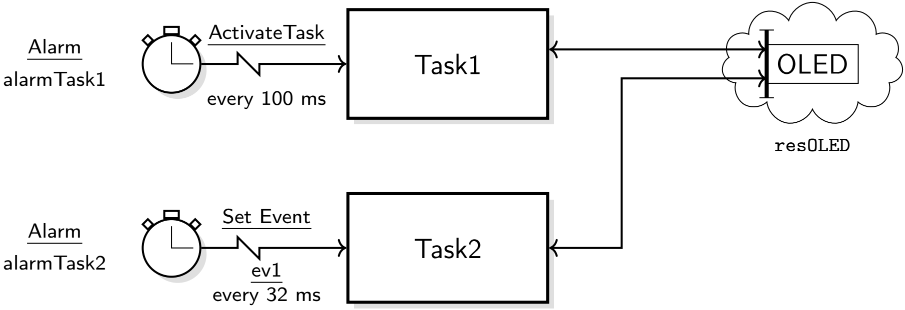
The priority of the resource is computed according to the priority of all the tasks and ISR2 that use it. So the resource must be declared. Otherwise, unpredictable behavior may occur.

OSEK Resource - Graphical representation



The access to the global variable is *protected* by the resource.

OSEK Resource - Graphical representation



The access to the peripheral `OLED` is *protected* by the resource in the same way.

OSEK Resource - Usage

The resource should be taken *as little time as possible*.

```
TASK{Task1}
{
  ...
  GetResource(resVal);
  val++;  //protected access to val
  ReleaseResource(resVal);
  ...
}
```

OSEK Resource - Usage (2)

In a conditional statement :

```
//the very bad way
TASK{Task1}
{
  ...
  GetResource(resVal);
  if(val == 5) {
    ...
  } else {
    ...
  }
  ReleaseResource(resVal);
  ...
}
```

Resource `resVal` may be taken for a *very long time*...

```
//the bad way
TASK{Task1}
{
  ...
  GetResource(resVal);
  if(val == 5) {
    ReleaseResource(resVal);
    ...
  } else {
    ReleaseResource(resVal);
    ...
  }
  ...
}
```

Resource `resVal` is taken for a short time, but a `releaseResource()` may be *forgotten in the `else` statement*.

OSEK Resource - Usage (3)

In a conditional statement, we use a temporary variable !

```
TASK{Task1}
{
  GetResource(resVal);
  const int tmp = val;
  ReleaseResource(resVal);
  if(tmp == 5) {
    ...
  } else {
    ...
  }
  ...
}
```