Technical Documentation

Written by GT-DISC

isae
Institut Supérieur de l'Aéronautique et de l'Espace
SUPAERO

Table of Contents

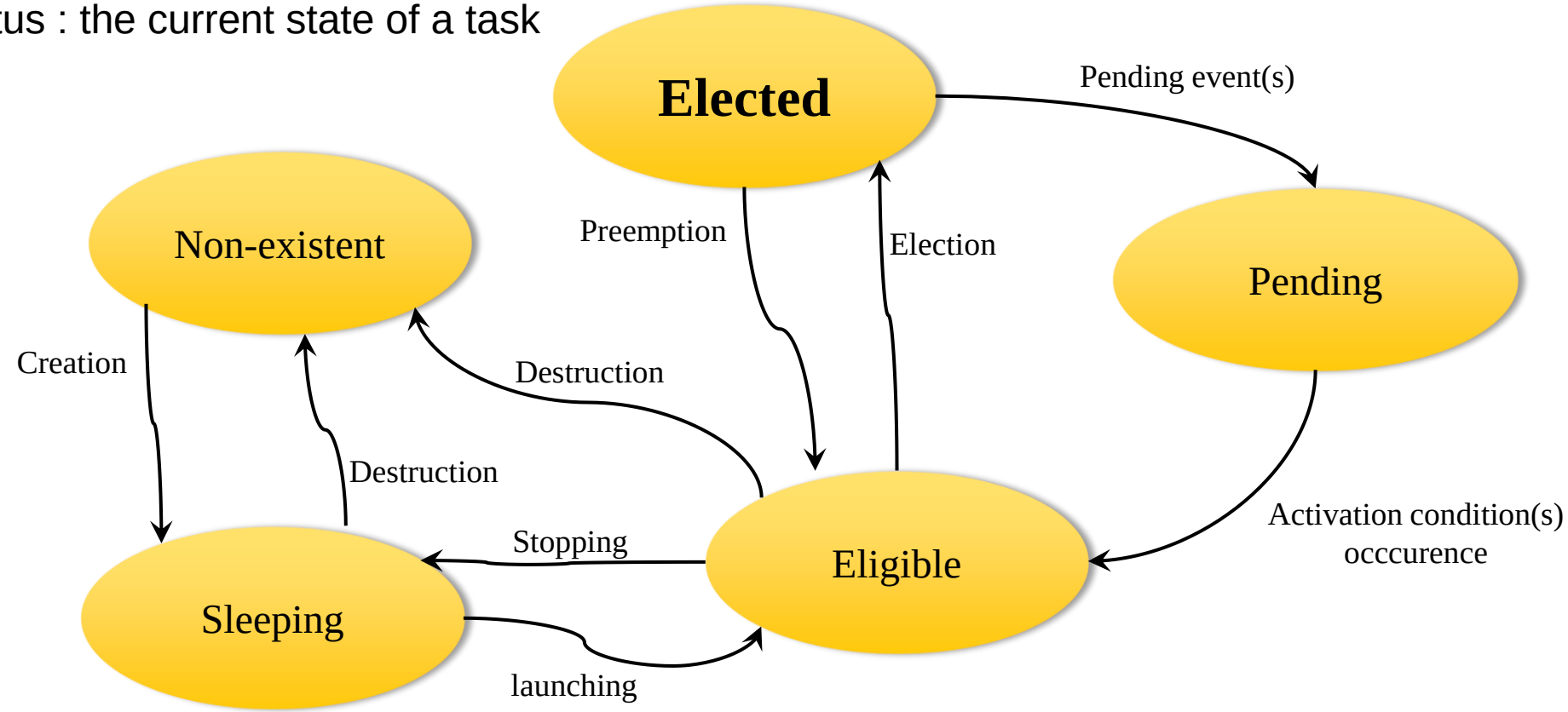
1. Real Time Reminders

- *Tasks*
- *Binary Semaphores*
- *Mutex*
- *Differences between Binary semaphore and Mutex*
- *Messages queue*
- *Signals*

Tasks (1)

» Tasks Status

- Active task : the task being served by the CPU
- Tasks status : the current state of a task



Tasks (2)

- Tasks usage :
  - ➔ May group one or several functions
  - ➔ Is defined by :
    - An entry point — *void My_Task()*
    - A relative priority with the other tasks
    - An identifier — *(RT_TASK)*
  - ➔ Usualy an infinite loop
  - ➔ Must contains at least one waiting point
  - ➔ Main services used
    - Creation : — *(rt_task_create)*
    - Starting : — *(rt_task_start)*
  - ➔ Destroyed by the RTOS — *(rt_task_delete)*

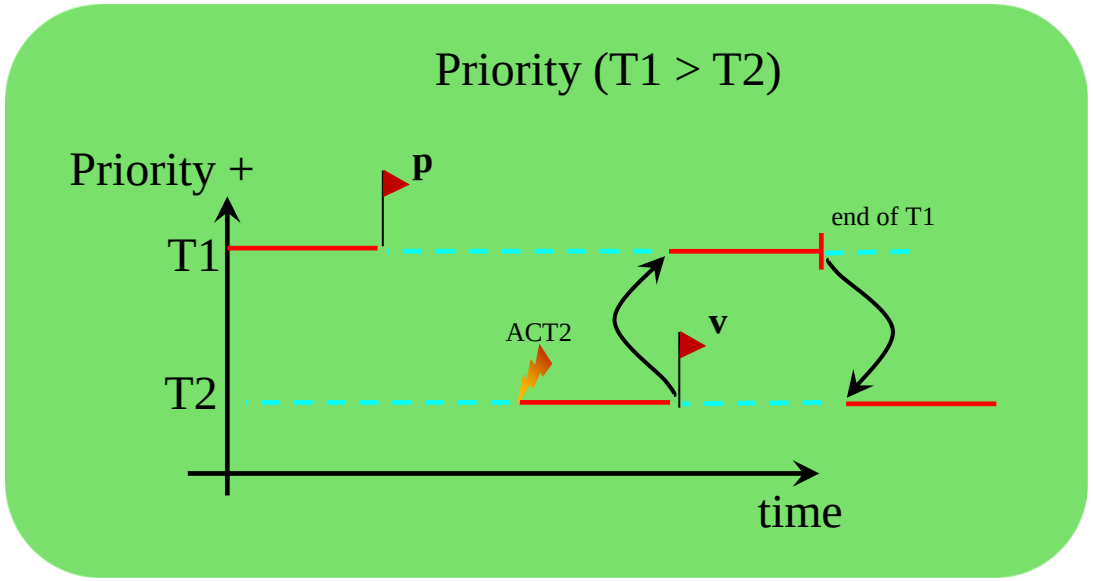
Binary Semaphores (1)

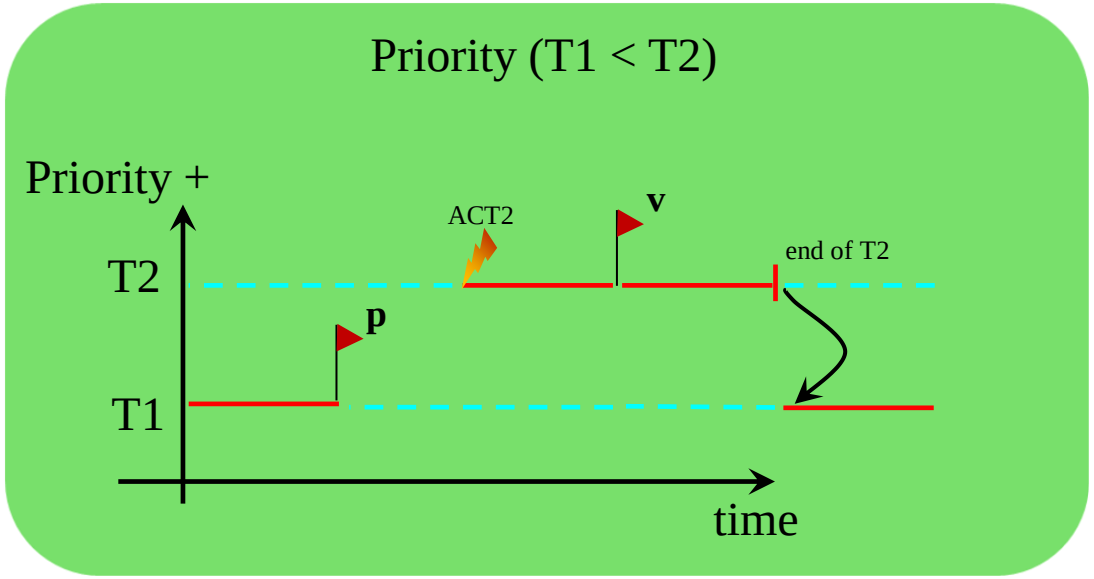
Under Xenomai a binary semaphore is a counting semaphore initialized to 0.

- Used for synchronization between tasks
- Created and initialized by the RTOS *(rt_sem_create)*
- Is defined by :
  - An identifier *(RT_SEM)*
- Using 2 primitives :
  - ‘v’ : Verhogen (NL) : to increase *(rt_sem_v)*
  - ‘p’ : Proberen (NL) : to try *(rt_sem_p)*
- Is destroyed by the RTOS *(rt_sem_ delete)*

Binary Semaphores (2)

» Binary semaphore behavior





T1 : Task #1

T2 : Task #2

ACT2 : Activation condition for task T2

p : Pending on a semaphore

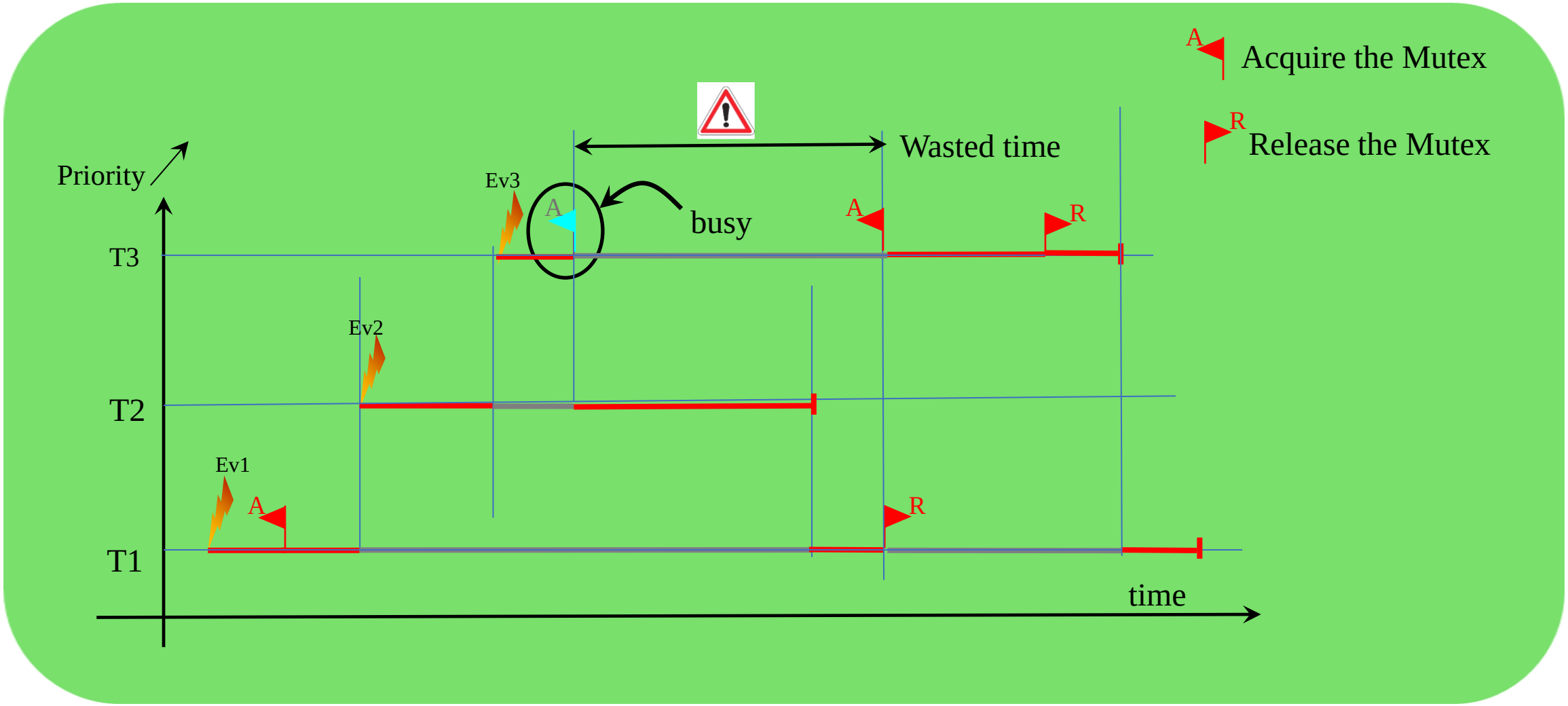
v : Semaphore donation

Mutex (1)

- **Usage** :
  - Used to protect to :
    - Simultaneous Read/Write acces to a global variable
  - Acces to a hardware ressource
  - Created and initialized by the RTOS *(rt_mutex_create)*
  - Is defined by an identifier : *(RT_MUTEX)*
  - Uses 2 RTOS services :
    - Acquire : *(rt_mutex_acquire)*
    - Release : *(rt_mutex_release)*
  - Destruction : *(rt_mutex_delete)*

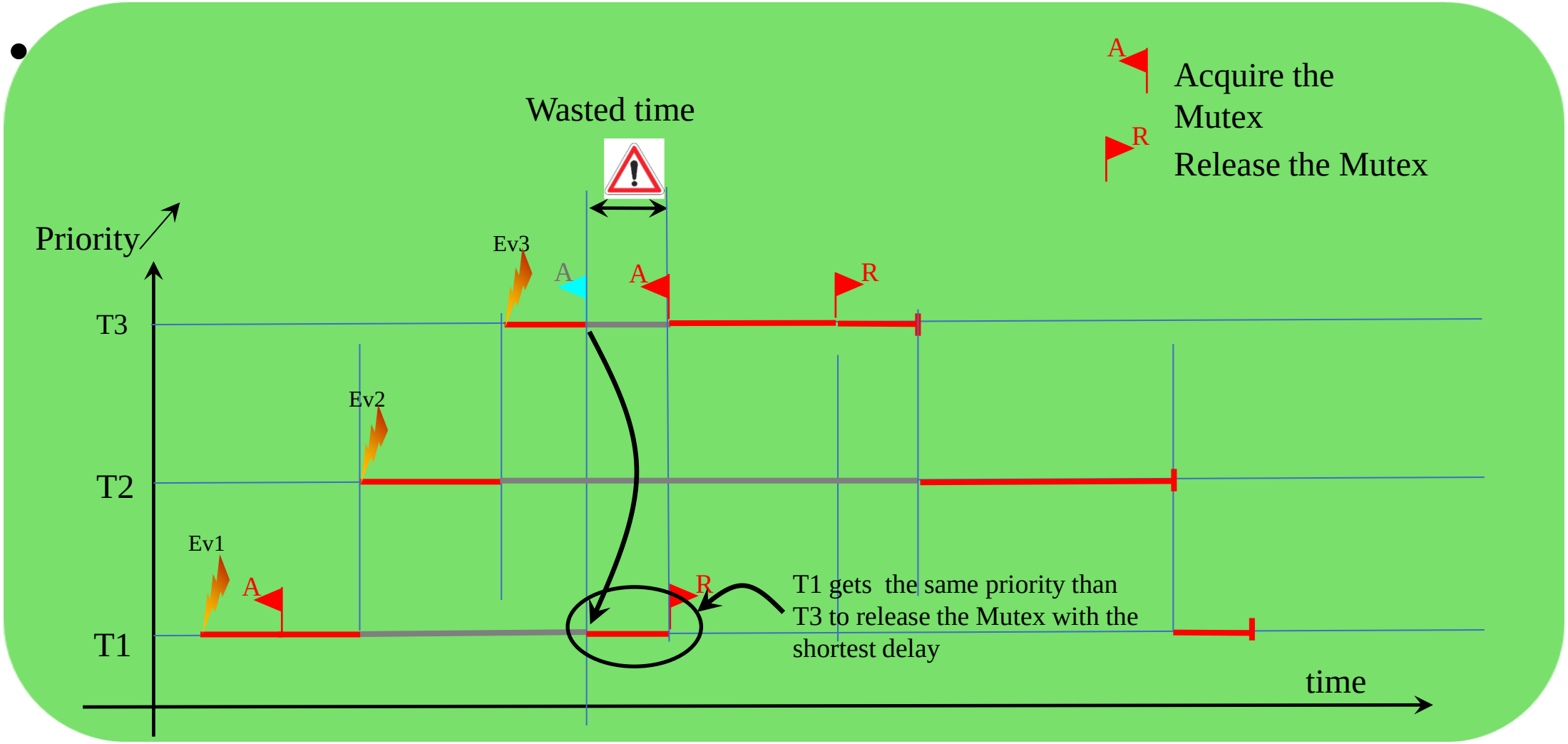
Mutex (2)

» Mutex *(priority inversion)*



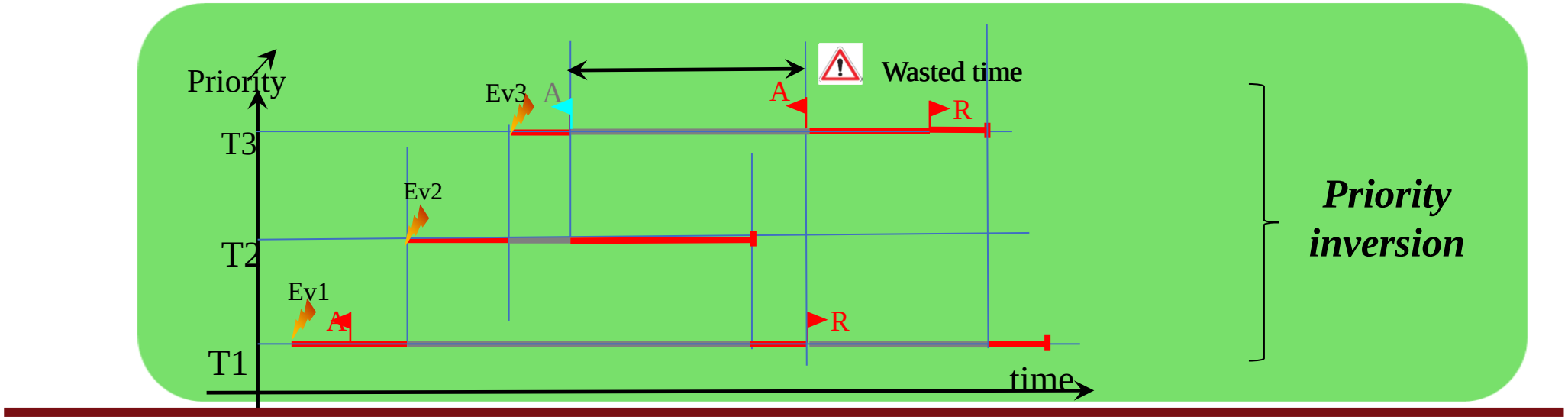
Mutex (3)

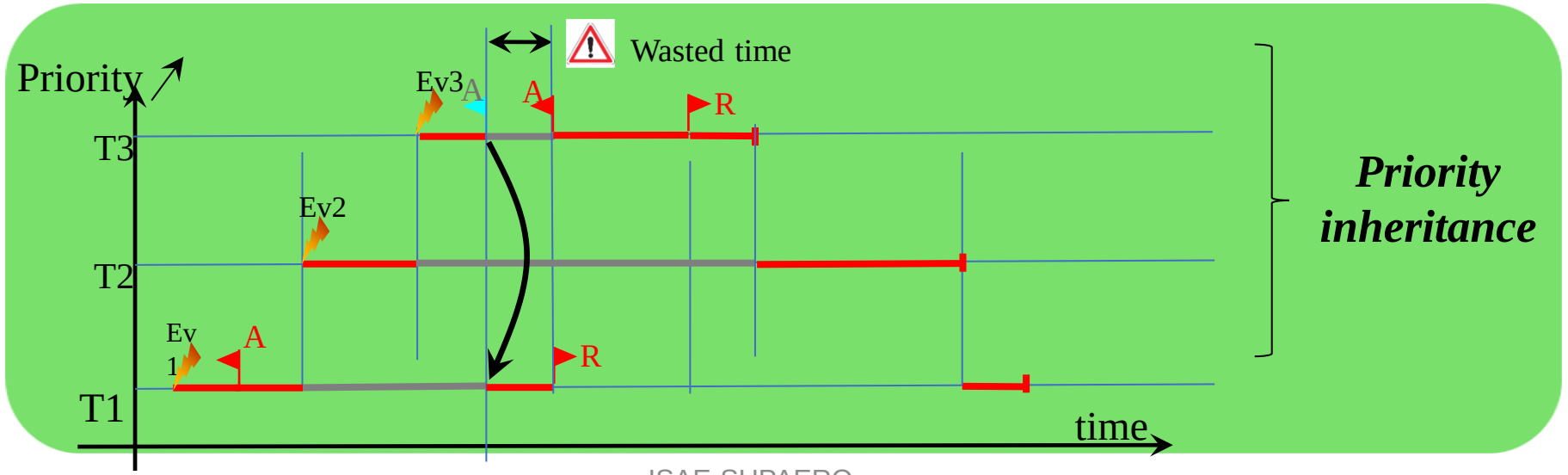
» Mutex *(priority inheritance)*



Mutex (4)

» Mutex *(management comparison)*





Differences between Binary semaphore and Mutex

Binary semaphore

Mutex

- Inheritance
- Initial condition

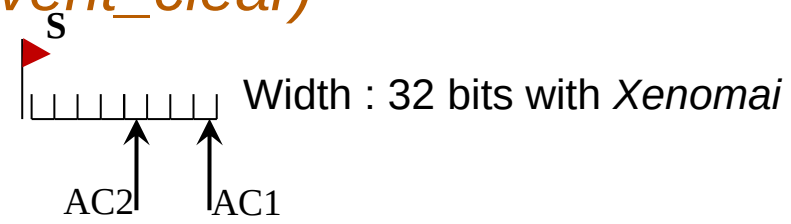
Message queue

- **Usage** :
  - ➔ For :
    - Tranfert datas between tasks
    - Synchronize tasks
  - ➔ Can contains several messages
  - ➔ Created and initialized by the RTOS *(rt_queue_create)*
  - ➔ A protection mecanism is incorporated
  - ➔ Is defined by
    - An identifier : *(RT_QUEUE)*
  - ➔ The size of 1 message
  - ➔ The number of messages
  - ➔ Uses 3 RTOS services :
    - Write a message : *(rt_queue_write)*
    - Read a message : *(rt_queue_read)*
    - Know the message queue status : (*rt_queue_inquire)*
  - ➔ Destruction : *(rt_queue_delete)*

Signals

» **Usage** :

- Used to synchronize one or sevaral tasks with one (or more) binary signal
- Created and initialized by the RTOS : *(rt_event_create)*
- Is defined by an identifier : *(RT_EVENT)*
- Uses 3 RTOS services :
  - Posting events signal : *(rt_event_signal)*
  - waits for one or more events : *(rt_event_wait)*
  - Clears an event mask : *(rt_event_clear)*
- Destruction : *(rt_event_delete)*



2. Xenomai API

- *Tasks creation and destruction*
- *Binary semaphores*
- *Mutex*
- *Message Queues*
- *Signals*

Tasks prototypes (1)

rt_task_create: Create a new real-time task

Prototype function :

```
int   rt_task_create (RT_TASK *        task,
                      const char
*                     name,
                      int
                      stksize,
                      int
                      prio,
                      int
                      mode)
```

Returned values :

0 :	Is returned upon success
ENOMEM :	The systel faild to get enough dynamic memory
EEXIST	The name is already used by some registred object
EPERM	This service was called from an asynchronous context

Parameters Description :

task :	Task descriptor
name :	ASCII string standing for the symbolic name of the task
stksize :	The size of the stack (in bytes) for the new task.. If zero is passed, a reasonable pre-defined size will be substituted
prio :	The base priority of the new task. This value must range from [1 .. 99] (inclusive) where 1 is the lowest effective priority.
mode :	T_FPU : allows the task to use the FPU T_SUSP : causes the task to start in suspended mode T_CPU : makes the new task affine to CPU T_JOINABLE : allows another task to wait on the termination of the new task.

Example :

```
RT_TASK DescrTask1;

/* Create Task1 */

rt_task_create(& DescrTask1, ``MyTask``,0,9,T_FPU);
```

Tasks prototypes (2)

rt_task_start: Start a new real-time task.

Prototype function :

```
int    rt_task_start (RT_TASK *task,
                      void(*)(void *cookie) entry,
                      void *cookie)
```

Parameters Description :

task :	Task descriptor
entry :	The address of the task's body routine. In other words, it is the task entry point. *:*
cookie :	A user-defined opaque cookie the real-time kernel will pass to the emerging task as the sole argument of its entry

Returned values :

0 :	Is returned upon success
EINVAL	Is returned if *task* is not a task descriptor
EIDRM	Is returned if *task* is a deleted task descriptor
EBUSY	Is returned if *task* is a already started
EPERM	This service was called from an asynchronous context

Example :

```
RT_TASK DescrTask1;

/* Start Task1 */

rt_task_start(& DescrTask1 ,&tTask1, NULL)
```

Tasks prototypes (3)

rt_task_delete : Delete a real-time task

Prototype function :

```
int    rt_task_delete (RT_TASK *  task)
```

Parameters Description :

task :	Task descriptor

Returned values :

0 :	Is returned upon success
EINVAL *:*	The parameter *task* is not a task descriptor
EPERM	This service was called from an asynchronous context
EINTR	The task is in a safe section
EIDRM	The parameter task is a deleted task descriptor

Example :

```
RT_TASK DescrTask1;

/* Destruction of Task1 */

rt_task_delete(& DescrTask1);
```

Tasks prototypes (4)

rt_task_spawn: Spawn a new real-time task.

Prototype function :

```
int    rt_task_spawn (RT_TASK *          task,
                      const char *       name,
                      int                stksize,
                      int                prio,
                      int                mode,
                      void(*)(void *cookie)  entry,
                      void *             cookie)
```

Returned values :

0 :	Is returned upon success
ENOMEM :	The systel faild to get enough dynamic memory
EEXIST	The name is already used by some registred object
EPERM	This service was called from an asynchronous context

Parameters Description :

task :	Task descriptor
name :	ASCII string standing for the symbolic name of the task*
stksize :	The size of the stack (in bytes) for the new task.. If zero is passed, a reasonable pre-defined size will be substituted
prio :	The base priority of the new task. This value must range from [1 .. 99] (inclusive) where 1 is the lowest effective priority.
mode :	T_FPU : allows the task to use the FPU T_SUSP : auses the task to start in suspended mode T_CPU : makes the new task affine to CPU T_JOINABLE : allows another task to wait on the termination of the new task.
entry	The address of the task's body routine. In other words, it is the task entry point. *:*
cookie :	A user-defined opaque cookie the real-time kernel will pass to the emerging task as the sole argument of its entry

Example :

```
RT_TASK DescrTask1;

/* Create and launch Task1 */

rt_task_spawn(& DescrTask1,” MyTask ‘’,0,9,T_FPU,&tTask1,NULL);
```

Tasks prototypes (5)

rt_task_sleep : Delay the execution of the calling task

Prototype function :

```
int    rt_task_sleep (RTIME delay)
```

Parameters Description :

delay :	duration of suspension of the task in nano seconds

Returned values :

0	is returned upon success.
EINTR	is returned if rt_task_unblock() has been called for the sleeping task before the sleep time has elapsed.
EWOULDBLOCK	is returned if the system timer is inactive.
EPERM	is returned if this service was called from a context which cannot sleep

Example :

```
/* Pending 1 second delay   */

rt_task_sleep(1000000000);
```

Tasks prototypes (6)

rt_task_inquire : Return various information about the status of a given task.

Prototype function :

```
int         rt_task_inquire (RT_TASK                    * task,
                             RT_TASK_INFO * info)
```

Returned values :

0 :	Is returned upon success
EINVAL :	Is returned if *task* is not a task descriptor
EPERM :	This service was called from an asynchronous context
EIDRM	Is returned if *task* is a deleted task descriptor

Parameters Description :

task : Task descriptor

RT_TASK_INFO Structure :

int *bprio*	Base priority
int *cprio*	Current priority
unsigned *status*	Task's status
RTIME relpoint	Time of next release
char *name [NAME_LEN]*	Symbolic name assigned at creation
RTIME *exectime*	Execution time in primary mode in nanoseconds
int *modeswitches*	Number of primary → secondary mode switc*hes*
int *ctwswtiches*	Number of context swithces
int *pagefaults*	Number of trigered page faults

Example :

```
RT_TASK Task1Descriptor;
RT_TASK_INFO Infos;

rt_task_inquire (&Task1Descriptor,&Infos);
```

Tasks prototypes (7)

rt_task_set_periodic : Make a real-time task periodic.

Prototype function :

```
int     rt_task_set_periodic  (RT_TASK*   task,
                                                    RTIME        idate,
                                                    RTIME
                period)
```

Returned values :

0 :	is returned upon success.
EINVAL :	is returned if *task* is not a task descriptor, or *period* is different from TM_INFINITE but shorter than the scheduling latency value for the target system, as available from /proc/xenomai/latenc..
EIDRM :	is returned if *task* is a deleted task descriptor
ETIMEDOUT :	is returned if *idate* is different from TM_INFINITE and represents a date in the past
EWOULDBLOCK :	is returned if the system timer is not active.
EPERM	is returned if *task* is NULL but not called from a task context.

Parameters Description :

task : The descriptor address of the affected task. This task is immediately delayed until the first periodic release point is reached. If *task* is NULL, the current task is set periodic

idate : The initial date of the first release point, expressed. If *idate* is equal to TM_NOW, the current system date is used, and no initial delay takes place..

Period : The period of the task, expressed in clock ticks. Passing TM_INFINITE attempts to stop the task's periodic timer;

Example :

```
rt_task_set_periodic(NULL,TM_NOW,1000000);  // activation each 1 ms
```

Tasks prototypes (8)

rt_task_wait_period : Wait for the next periodic release point.

Prototype function :

```
int        rt_task_wait_period ( unsigned long  *  overrun)
```

Returned values :

0 :	is returned upon success. Otherwise
EWOULDBLOCK :	The service rt_task_set_periodic() has not previously been called for the calling task.
EINTR :	The rt_task_unblock() has been called for the waiting task before the next periodic release point has been reached.
ETIMEDOUT :	This value is is returned if a timer overrun occurred
EPERM :	This service was called from an asynchronous context.

Parameters Description :

Overrun :	must be a pointer to a memory location which will be written with the count of pending overruns. This value is copied only when *rt_task_wait_period* returns -ETIMEDOUT or success; the memory location remains unmodified otherwise. If NULL, this count will never be copied back

Example :

Pending a periodic activation. The period is defined when the service *rt_task_set_periodic* is called

```
rt_task_wait_period(NULL)
```

Bynary semaphores prototypes (1)

rt_sem_create : Binary semaphore creation.

Prototype function :

```
int          rt_sem_create  (RT_SEM *          sem,
                             const char *
name,
                             unsigned long    icount,
                             int              mode)
```

Returned values :

0 :	Is returned upon success
ENOMEM :	The systel faild to get enough dynamic memory
EEXIST	The name is already used by some registred object
EINVAL :	is returned if the *icount* is non-zero and *mode* specifies a pulse semaphore.
EPERM	This service was called from an asynchronous context

Parameters Description :

sem :	Semaphore descriptor
name :	ASCII string standing for the symbolic name of the semaphore
icount :	The initial value of the semaphore count
mode :	S_FIFO : Makes tasks pend in FIFO order on the semaphore S_PRIO : Makes tasks pend in priority order on the semaphore. S_PULSE : Semaphore is not incremented if no waiter is pending

Example :

```
RT_SEM mySemBin;

rt_sem_create (&mySemBin, “SemX", 0, S_PULSE );
```

Bynary semaphores prototypes (2)

rt_sem_delete : Binary semaphore destruction

Prototype function :

```
int    rt_sem_delete (RT_SEM *  sem)
```

Parameters Description :

sem : Semaphore descriptor

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *sem* is not a semaphore descriptor
EIDRM	The parameter *sem* is a deleted semaphore descriptor
EPERM	This service was called from an asynchronous context

Example :

```
RT_SEM DescrSem1;

/* Destruction of Semaphore1 */

rt_sem_delete(& DescrSem1);
```

Bynary semaphores prototypes (3)

rt_sem_v : Binary semaphore donation

Prototype function :

```
int    rt_sem_v (RT_SEM *  sem)
```

Parameters Description :

sem : Semaphore descriptor

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *sem* is not a semaphore descriptor
EIDRM :	The parameter *sem* is a deleted semaphore descriptor

Example :

```
RT_SEM DescrSem1;

/* Semaphore donation*/

rt_sem_v(& DescrSem1);
```

Bynary semaphores prototypes (4)

rt_sem_p : Binary semaphore Pend on

Prototype function :

```
int    rt_sem_p (RT_SEM *  sem,
                 RTIME    timeout)
```

Parameters Description :

sem :	Semaphore descriptor
timeout:	Delay to wait for a semaphore in nano second. Passing for timeout parameter : TM_INFINITE : Pending delay infinite. TM_NONBLOCK : No waiting if no unit is available.

Returned values :

0 :	is returned upon success. Otherwise :
EINVAL	is returned if *sem* is not a semaphore descriptor
EIDRM :	is returned if *sem* is a deleted semaphore descriptor.
EWOULDBLOCK :	is returned if *timeout* is equal to TM_NONBLOCK and the semaphore value is zero
EINTR	is returned if rt_task_unblock has been called for the waiting task before a semaphore unit has become available.
ETIMEDOUT :	is returned if no unit is available within the specified amount of time
EPERM :	is returned if this service should block

Example :

```
RT_SEM DescrSem1;

/* Semaphore Pend on*/

rt_sem_p(&DescrSem1, TM_INFINITE );
```

Mutex prototypes (1)

rt_mutex_create : Mutex creation.

Prototype function :

```
int     rt_mutex_create  (RT_MUTEX *     MyMutex,
                          const char *
name)
```

Parameters Description :

MyMutex : Mutex descriptor

name : ASCII string standing for the symbolic name of the mutex

Returned values :

0 :	Is returned upon success
ENOMEM :	The system faild to get enough dynamic memory
EEXIST	The *name* is already used by some registred object
EPERM	This service was called from an asynchronous context

Example :

```
RT_MUTEX myMutex;

rt_mutex_create (&myMutex, “MutexX”);
```

Mutex prototypes (2)

rt_mutex_delete : Mutex destruction

Prototype function :

```
int    rt_mutex_delete      ( RT_MUTEX *  myMutex)
```

Parameters Description :

myMutex : Mutex descriptor

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *myMutex* is not a mutex descriptor
EIDRM	The parameter *myMutex* is a deleted mutex descriptor
EPERM	This service was called from an asynchronous context

Example :

```
RT_MUTEX  myMutex;

/* Destruction of Mutex*/

rt_mutex_delete(&myMutex);
```

Mutex prototypes (3)

rt_mutex_acquire : Acquire a mutex

Prototype function :

```
int    rt_mutex_acquire (RT_MUTEX *  myMutex,
                         RTIME       timeout)
```

Parameters Description :

myMutex :	Mutex descriptor
timeout:	Delay to wait for the mutex in nano second. Passing for timeout parameter : TM_INFINITE : Pending delay infinite. TM_NONBLOCK : No waiting if no unit is available.

Returned values :

0 :	is returned upon success. Otherwise :
EINVAL	is returned if *myMutex* is not a semaphore descriptor
EIDRM :	is returned if *myMutex* is a deleted semaphore descriptor.
EWOULDBLOCK :	*timeout* = TM_NONBLOCK and the mutex is not available
EINTR	is returned if *rt_task_unblock* has been called for the waiting task before the mutex has become available.
ETIMEDOUT :	is returned if the mutex cannot be made available to the calling task within the specified amount of time.
EPERM :	is returned if this service was called from a context which cannot be given the ownership of the mutex

Example :

```
RT_MUTEX myMutex;

/* Semaphore Pend on*/

rt_mutex_acquire (&myMutex, TM_INFINITE );
```

Mutex prototypes (4)

rt_mutex_release: Unlock mutex

Prototype function :

int rt_mutex_release (RT_MUTEX * *myMutex*)

Parameters Description :

myMutex : Mutex descriptor

Returned values :

0 : Is returned upon success

EINVAL : The parameter *muMutex* is not a semaphore descriptor

EIDRM : The parameter *myMutex* is a deleted mutex descriptor

EPERM : This service was called from an asychronous context

Example :

```
RT_MUTEX myMutex

/* Mutex release */

rt_mutex_release(&myMutex);
```

Mutex prototypes (5)

rt_mutex_inquire : Query mutex status

Prototype function :

```
int         rt_mutex_inquire (RT_MUTEX          *myMutex,
                              RT_MUTEX_INFO   * infos)
```

Returned values :

0 :	Is returned upon success
EINVAL :	Is returned if *mutex i*s not a mutex descriptor
EIDRM :	Is returned if *mutex* is a deleted mutex descriptor

Parameters Description :

task :	Task descriptor
RT_MUTEX_INFO	Structure :
int *lockent*	lock nesting level (> 0 means "locked").
int *nwaiters*	Number of pending tasks
char *name [XNOBJECT_NAME_LEN]*	Symbolic name.

Example :

```
RT_MUTEX          myMutex
RT_MUTEX_INFO   infos;

rt_mutex_inquire (&myMutex, &infos);
```

Message queues prototypes (1)

rt_queue_create: Message queue creation.

Prototype function :

```
int    rt_queue_create(RT_QUEUE *queue,
const char *           name,
                       size_t poolsize,
                       size_t qlimit,
                       int mode)
```

Returned values :

0 :	Is returned upon success
EEXIST :	The *name* is already in use by some registered object
EINVAL :	Is returned if *poolsize* is null, greater than the system limit, or *name* is null or empty for a shared queue
ENOMEM :	The name is already used by some registred object
EPERM :	This service was called from an asynchronous context
ENOSYS :	Returned ifi *mode* specifies Q_SHARED, but the real-time support in user-space is unavailable
ENOENT :	is returned if /dev/rtheap can't be opened.

Parameters Description :

queue :	Message queue descriptor
name :	ASCII string standing for the symbolic name of the task*
poolsize :	The size (in bytes) of the message buffer pool which is going to be pre-allocated to the queue
qlimit :	maximum number of messages which can be queued (max 255)
mode :	The Queue cration mode : Q_FIFO: makes tasks pend in FIFO order on the queue for consuming messages. Q_PRIORITY: makes tasks pend in priority order on the queue Q_SHARED: : causes the queue to be sharable between kernel and user-space tasks Q_DMA: : auses the buffer pool associated to the queue to be allocated in physically contiguous memory

Example :

See next slide

Message queues prototypes (2)

rt_queue_create (continued)

Example :

```
#define BUFFER_SIZE 10

typedef  struct
{
        double Elevation ;
        double Azimuth ;
} TypePosition ;

RT_QUEUE MyQueue;

rt_queue_create( &MyQueue,"queue1", BUFFER_ SIZE * sizeof(TypePosition), BUFFER_SIZE, Q_FIFO);
```

Message queues prototypes (3)

rt_queue_delete : Message queue destruction

Prototype function :

```
int    rt_queue_delete       (RT_QUEUE *  queue)
```

Parameters Description :

queue :	Message queue descriptor

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *queue* is not a queue descriptor
EIDRM	The parameter *queue* is a deleted queue descriptor
EPERM	This service was called from an asynchronous context
EBUSY	Message queue is still bound to a process

Example :

```
RT_QUEUE  myQueue;

/* Destruction of  myQueue      */

rt_queue_delete(&myQueue);
```

Message queues prototypes (4)

rt_queue_write : Write in a mesage queue**.**

Prototype function :

```
int rt_queue_write (RT_QUEUE *  Queue,
                    const void *  buf,
                    size_t  size,
                    int  mode)
```

Parameters Description :

queue :	Message queue descriptor
buf:	The message data to be written to the queue
size :	The size (in bytes) of a message data
mode :	A set of flags affecting the operation Q_NORMAL: Message added at the end of the queue Q_URGENT: Message added at the lead of the queue (LIFO) Q_BROADCAST: : send the message to all tasks currently waiting for messages

Example :

```
RT_QUEUE MyQueue;

typedef  struct
{               double Elevation ;
                double Azimuth ;
} TypePosition ;

TypePosition Position ;

rt_queue_write(&MyQueue,&Position,sizeof(TypePosition),Q_NORMAL);
```

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *queue* is not a queue descriptor
EIDRM :	The parameter *queue* is a deleted queue descriptor
ENOMEM :	Full queue

Message queues prototypes (5)

rt_queue_read : Read a message queue

Prototype function :

```
int rt_queue_read    (RT_QUEUE *  Queue,
                      const void *  buf,
                      size_t  size,
                      RTIME  timeout)
```

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *Queue* is not a queue descriptor
EIDRM :	The parameter *Queue* is a deleted queue descriptor
EWOULDBLOCK :	No message is immediately available on entry. (*timeout* is equal to TM_NONBLOCK)
EINTR	rt_task_unblock() has been called for the waiting task before any data was available.
EPERM	Is returned if this service should block, but was called from a context which cannot sleep

Parameters Description :

queue :	Message queue descriptor
buf:	The message data to be written to the queue
size :	The size (in bytes) of a message data
timeout :	Pendind delay on message queue (in nano secondes) TM_NONBLOCK : returns immediately without waiting TM_INFINITE : causes the caller to block indefinitely until some message is eventually available

Example :

```
RT_QUEUE MyQueue;

typedef  struct
{               double Elevation ;
                double Azimuth ;
} TypePosition ;

TypePosition Position ;
rt_queue_read (&MyQueue,&Position,sizeof(TypePosition), TM_NONBLOCK);
```

Message queues prototypes (6)

rt_queue_inquire : Query message queue status

Prototype function :

```
int rt_queue_inquire  (RT_QUEUE *  Queue,
                       RT_QUEUE_INFO * infos)
```

Returned values :

0 :	Is returned upon success
EINVAL :	Is returned if *Queue* is not a queue descriptor
EIDRM	Is returned if *Queue* is a deleted queue descriptor

Parameters Description :

Queue : Queue descriptor

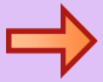
RT_QUEUE_INFO Structure

int *nwaiters*;	Number of pending tasks.
int *nmessages*;	Number of queued messages.
int *mode*	Creation mode.
size_t *qlimit*;	Queue limit
size_t *poolsize*;	Size of pool memory (in bytes).
size_t *usedmem*;	Amount of pool memory used (in bytes).
char *name[XNOBJECT_NAME_LEN]* :	Symbolic name.

Example :

```
RT_QUEUE_INFO Informations;
RT_QUEUE MyQueue;


rt_queue_inquire(&MyQueue,&Informations);
```



The field *Informations.nmessages* contains the number of queued messages

Signals prototypes (1)

rt_event_create : Creates a signal group

Prototype function :

```
int    rt_event_create (RT_EVENT *      signal,
                                        const char
*              name,
                                        unsigned
long           ivalue,
                                        int
               mode)
```

Returned values :

0 :	Is returned upon success
EEXIST :	The *name* is already in use by some registered object
EPERM :	Is returned if this service was called from an asynchronous context.
ENOMEM :	Is returned if the system fails to get enough dynamic memory from the global real-time heap in order to register the signal group.

Parameters Description :

signal :	Signal descriptor
name :	An ASCII string standing for the symbolic name of the group
ivalue :	The initial value of the group's signal mask.
mode :	The event group creation mode • EV_FIFO : makes tasks pend in FIFO order on the signal group. • EV_PRIO : makes tasks pend in priority order on the signal group.

Example :

```
RT_EVENT mySignals;

rt_event_create(&mySignals, ”Signals”, 0, EV_FIFO);
```

Signals prototypes (1)

rt_event_delete : Deletes a signal group

Prototype function :

```
int    rt_event_delete (RT_EVENT *      signal)
```

Parameters Description :

signal : Signal descriptor

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *signal* is not a signal descriptor
EIDRM :	The parameter *signal* is a deleted signal descriptor
EPERM :	This service was called from an asynchronous context

Example :

```
RT_EVENT mySignals;

/* Destruction of  mySignals */

rt_event_delete(&mySignals);
```

Signals prototypes (1)

rt_event_signal : Posts an event group

Prototype function :

```
int    rt_event_signal (RT_EVENT *      signal,
                                        unsigned
long         mask)
```

Parameters Description :

signal :	Signal descriptor
mask :	Set of signal to be posted

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *signal* is not a signal descriptor
EIDRM	The parameter *signal* is a deleted signal descriptor
EPERM	This service was called from an asynchronous context

Example :

```
RT_EVENT mySignals;

#define      BIT(n)      (1<<n)
#define      EVENT_A  BIT(4)

/* Post EVENT_A    (bit 4)*/

rt_event_signal(&mySignals, EVENT_A);
```

Signals prototypes (1)

rt_event_wait : Waits for one or more events on the specified event group

Prototype function :

```
int    rt_event_wait (RT_EVENT *          signal,
                      unsigned long       mask,
                      unsigned long *
              mask_r,
                      int
              mode,
                      RTIME
              timeout)
```

Parameters Description :

signal : Signal descriptor

mask The set of bits to wait for

mask_r : The value of the event mask at the time the task was readied

mode : The pend mode :
- EV_ANY : makes the task pend in disjunctive mode (i.e. OR)
- EV_ALL : makes the task pend in conjunctive mode (i.e. AND).

timeout : Pending delay
- TM_NONBLOCK : Returns immediately without waiting
- TM_INFINITE : Block indefinitely until the request is fulfilled

Returned values :

0 : Is returned upon success

EINVAL : The parameter *signal* is not a signal descriptor

EIDRM : The parameter *signal* is a deleted signal descriptor

EWOULDBLOCK : *timeout* is equal to TM_NONBLOCK and the current event mask value does not satisfy the request

EINTR : rt_task_unblock() has been called for the waiting task before the request has been satisfied.

ETIMEDOUT : the request has not been satisfied within the specified amount of time.

EPERM : This service was called from an asynchronous context

Example :

See next slide

Signals prototypes (1)

Example #1 :

```
RT_EVENT mySignals;

#define       BIT(n)       (1<<n)
#define       EVENT_A   BIT(4)
#define       EVENT_B   BIT(6)

unsigned long maskValue;

/*     Waits for EVENT_A  and   EVENT_B    indefinitely */
rt_event_wait(&mySignals,EVENT_A | EVENT_B, &maskValue, EV_ALL, TM_INFINITE);
```

Example #2 :

```
unsigned long maskValue

/*     Waits for EVENT_A  or   EVENT_B    indefinitely */
rt_event_wait(&mySignals, EVENT_A | EVENT_B, &maskValue, EV_ANY, TM_INFINITE);
```

The event bits are NOT cleared from the event group when a request is satisfied; rt_event_wait() will return immediately with success for the same event mask until ***rt_event_clear()*** is called to clear those bits.

```
rt_event_clear(&mySignals,EVENT_A | EVENT_B, NULL)
```

Signals prototypes (1)

rt_event_clear : Clears a set of signals from an event mask

Prototype function :

```
int    rt_event_clear (RT_EVENT *      signal,
                                       unsigned
long         mask,
                                       unsigned
long *       mask_r)
```

Returned values :

0 :	Is returned upon success
EINVAL :	The parameter *signal* is not a signal descriptor
EIDRM	The parameter *signal* is a deleted signal descriptor

Parameters Description :

signal :	Signal descriptor
mask	The set of bits to clear
mask_r :	If non-NULL, *mask_r* is the address of a memory location which will be written upon success with the previous value of the event group before the flags are cleared.

Example :

```
RT_EVENT mySignals;

#define      BIT(n)      (1<<n)
#define      EVENT_A  BIT(4)
#define      EVENT_B  BIT(6)

/*    Clears EVENT_A  and   EVENT_B    */

rt_event_clear(&mySignals, EVENT_A | EVENT_B, NULL);
```

3. Platform Documentation

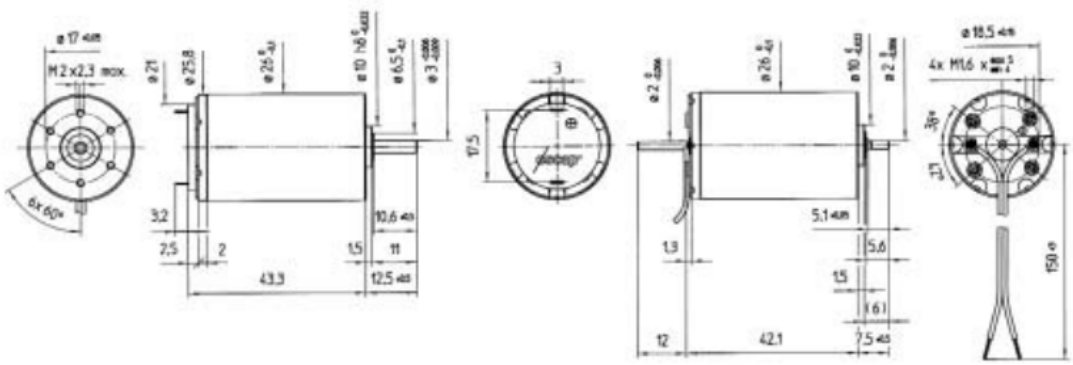
- *Motor Datasheet*
- *Current sensor Datasheet*
- *Gyroscope Datasheet*
- *Servo Datasheet*

Motor Datasheet

escap 26N58 & 26N48

Precious metal commutation system - 9 segments

D.C. Motor
5.7 Watt



scale: 2:3
dimensions in mm
mass: 114 g

26N58 ▭ • 1

26N48 ▭ • 9

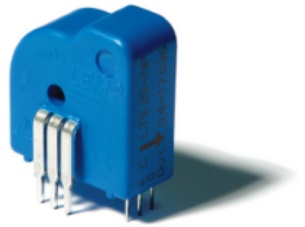
Winding types	▭	-216E	-110
Measured values			
1 Measuring voltage	V	12	24
2 No-load speed	rpm	4700	6700
3 Stall torque	mNm (oz-in)	28.6 (4.06)	25 (3.54)
4 Average no-load current	mA	16	12
5 Typical starting voltage	V	0.15	0.28
Max. recommended values			
6 Max. continuous current	A	0.86	0.34
7 Max. continuous torque	mNm (oz-in)	20 (2.8)	11 (1.56)
8 Max. angular acceleration	10^3 rad/s²	84	46
Intrinsic parameters			
9 Back-EMF constant	V/1000 rpm	2.5	3.5
10 Torque constant	mNm/A (oz-in/A)	23.9 (3.38)	33.5 (4.74)
11 Terminal resistance	ohm	10	32
12 Motor regulation R/k²	10^3/Nms	17	29
13 Rotor inductance	mH	0.8	1.7
14 Rotor inertia	kgm²·10^{-7}	8.5	5.3
15 Mechanical time constant	ms	15	19

Current Sensor Datasheet

Current Transducer LTS 6-NP

For the electronic measurement of currents: DC, AC, pulsed..., with galvanic separation between the primary circuit and the secondary circuit.

I_{PN} = 6 At



Features

- Closed loop (compensated) current transducer using the Hall effect
- Unipolar supply voltage
- Insulating plastic case recognized according to UL 94-V0
- Compact design for PCB mounting
- Incorporated measuring resistance
- Extended measuring resistance.

Electrical data

I_{PN}	Primary nominal RMS current		6	At
I_{PM}	Primary current, measuring range		0 ... ±19.2	At
I_P	Overload capability		250	At
V_{out}	Output voltage (analog) @ I_P		2.5 ±(0.625 × I_P/ I_{PN})	V
	@ I_P = 0		2.5 [1]	V
G	Sensitivity		104.16	mV/A
N_S	Number of secondary turns (±0.1 %)		2000	
R_L	Load resistance		≥ 2	kΩ
R_{IM}	Internal measuring resistance (±0.5 %)		208.33	Ω
TCR_{IM}	Temperature coefficient of R_{IM}		< 50	ppm/K
U_C	Supply voltage (±5 %)		5	V
I_C	Current consumption @ U_C = 5 V	Typical	28 + I_S [2] (V_{out}/R_L)	mA

Gyroscope Datasheet

InvenSense	MPU-9250 Product Specification	Document Number: PS-MPU-9250A-01 Revision: 1.0 Release Date: 01/17/2014

3 Electrical Characteristics

3.1 Gyroscope Specifications

Typical Operating Circuit of section 4.2, VDD = 2.5V, VDDIO = 2.5V, T_A=25°C, unless otherwise noted.

PARAMETER	CONDITIONS	MIN	TYP	MAX	UNITS
Full-Scale Range	FS_SEL=0		±250		°/s
	FS_SEL=1		±500		°/s
	FS_SEL=2		±1000		°/s
	FS_SEL=3		±2000		°/s
Gyroscope ADC Word Length			16		bits
Sensitivity Scale Factor	FS_SEL=0		131		LSB/(°/s)
	FS_SEL=1		65.5		LSB/(°/s)
	FS_SEL=2		32.8		LSB/(°/s)
	FS_SEL=3		16.4		LSB/(°/s)
Sensitivity Scale Factor Tolerance	25°C		±3		%
Sensitivity Scale Factor Variation Over Temperature	-40°C to +85°C		±4		%
Nonlinearity	Best fit straight line; 25°C		±0.1		%
Cross-Axis Sensitivity			±2		%
Initial ZRO Tolerance	25°C		±5		°/s
ZRO Variation Over Temperature	-40°C to +85°C		±30		°/s
Total RMS Noise	DLPFCFG=2 (92 Hz)		0.1		°/s-rms
Rate Noise Spectral Density			0.01		°/s/√Hz
Gyroscope Mechanical Frequencies		25	27	29	KHz
Low Pass Filter Response	Programmable Range	5		250	Hz
Gyroscope Startup Time	From Sleep mode		35		ms
Output Data Rate	Programmable, Normal mode	4		8000	Hz

98 Hz

Table 1 Gyroscope Specifications

Servo Datasheet

SERVO DUTY

SERIES E9

Dynapar™ brand

Miniature Encoder

Key Features

- Super-Compact Modular Encoder for Small Servo and Stepper Motor Feedback
- Integrated ASIC for Enhanced Reliability and Accuracy
- Up to 512 PPR Resolution

Code 1: Model	Code 2: PPR	Code 3: Hub Bore Description	Code 4: Output Description	Code 5: Mounting Description
E9	□□□□	□□□	□□	□

Ordering Information

E9	0.9" Diameter Incremental Modular Encoder	0100	2.0	2.0 mm	00	See Figure 1	0	No mounting base
		0144	2.5	2.5 mm	01	See Figure 2	A	4x M1.6 on 0.728" BC
		0200	3.0	3.0 mm	02	See Figure 3	C	2x #2-56 on 0.75" BC
		0500	4.0	4.0 mm			D	3x #0-80 on 0.823" BC
		0512	125	0.125 in			E	2x #2-56 On 1.812" BC
		Special Order Consult Factory for Lead Time & Price	156	0.156 in				
		0256	Special Order Consult Factory for Lead Time & Price					
		0300	1.5	1.5 mm				
		0360						

Worldwide Brands: NorthStar™ • Dynapar™ • Hengstler™ • Harowe™

DYNAPAR™ INNOVATION - CUSTOMIZATION - DELIVERY

WWW.DYNAPAR.COM

Headquarters: 1675 Delany Road • Gurnee, IL 60031-1282 • USA

Customer Service:
Tel.: +1.800.873.8731
Fax: +1.847.662.4150
custserv@dynapar.com

Technical Support
Tel.: +1.800.234.8731
Fax: +1.847.662.4150
dynapar.techsupport@dynapar.com

European Sales Representitive
Hengstler GmbH
Uhlandstrasse 49, 78554 Aldingen
Germany
www.hengstler.com

SPECIFICATIONS

STANDARD OPERATING CHARACTERISTICS

Code: Incremental, Optical
Resolution: 100 to 512 PPR (pulses/revolution)
Symmetry: 180° ±18° electrical
Index: 90° ±36° electrical
Quadrature Phasing: 90° ±18° electrical
Phase Sense: A leads B for CW shaft rotation
Format: See chart below (output waveform & connections)

ELECTRICAL

Input Power: 5 VDC ±10%, 10mA, typ.
Output Signals: 2.5 V min. high (V_{OH}); 0.5 V max. low (V_{OL}). 6 mA sink/source (25°C), 4 mA (100°C)
Frequency Response: 200 kHz
Termination: 5 pin header (accessory 12" wires w/connector, part no. CA0050012) or flying leads
Recommended Mating Connector: AMP part number 103675-4

MECHANICAL

Hub Diameter: 1/8", 5/32", 1.5mm ~ 4.0 mm
Hub Dia. Tolerance: +0.0004"/-0.0000" (+0.010 mm/-0.000 mm)
Mating Shaft Length: See table
Mating Shaft Runout: 0.001 TIR
Mating Shaft Endplay: >256 ppr: ±0.003" (±0.076mm); 250, 256 ppr: +0.005/-0.003" (+0.127/-0.076mm); <250 ppr: +0.007/-0.003" (+0.178/-0.076mm)
Moment of Inertia: 0.15 x 10^{-3} oz-in-sec^2 (0.11 gm-cm^2)
Housing and Cover: Plastic
Weight: 0.15 oz (4.14 g)

ENVIRONMENTAL

Operating Temperature: -20° to 100°C
Storage Temperature: -50° to 125°C
Humidity: Up to 90% (non-condensing)

OUTPUT WAVEFORMS & CONNECTIONS

Code 4= 00

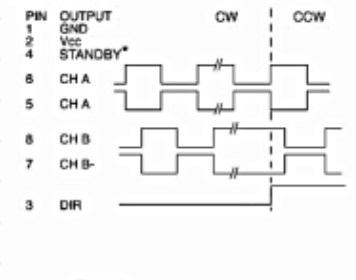


Figure 1

Code 4= 01

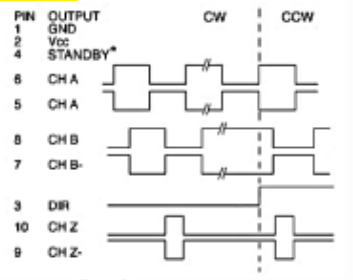


Figure 2

Code 4= 02

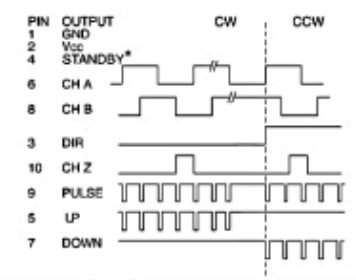


Figure 3

* For operation, connect **STANDBY (4)** to **Vcc (2)**

Institut Supérieur de l’Aéronautique et de l’Espace

10, avenue Edouard Belin – BP 54032
31055 Toulouse Cedex 4 – France
T +33 5 61 33 80 80

www.isae-supaero.fr

isae
Institut Supérieur de l’Aéronautique et de l’Espace
SUPAERO