GOS2022

Mutex Service Description

GOS2022

Mutex Service Description

for GOS2022 version 0.6

Description of the mutex service

List of API functions

© Ahmed Gazar, 2023

Version history

Version	Date	Author	Change	Released
1.0	2023-11-09	Ahmed Gazar	Initial version of the document.	2023-11-09

Content

1. Introduction

Mutex (**Mut**ual **Ex**clusion) is a service that ensures safe access to shared resources. A mutex guarantees that only one task has access to the object at a time that is protected by it, and other tasks wait for the resource until it has been released by the owner of the mutex in a non-blocking way.

2. Mutex states

A mutex has two definite states: *unlocked* and *locked*. Without initialization a mutex is in an indefinite state.

```
typedef enum
{
    GOS_MUTEX_UNLOCKED = 0b11010010, //!< Mutex unlocked.
    GOS_MUTEX_LOCKED   = 0b01101011  //!< Mutex locked.
}gos_mutexState_t;
```

3. Mutex instance

The mutex instance is a structure that contains a *state* and an *owner* field. The state field describes the current state of the mutex, while the owner is the task ID of the task that locks the mutex.

```
typedef struct
{
    gos_mutexState_t mutexState; //!< Mutex state.
    gos_tid_t        owner;      //!< Mutex owner task.
}gos_mutex_t;
```

4. Timeout

When locking a mutex, the user has to define a timeout value, after which the locking function returns regardless of whether the mutex has been locked or not.

The timeout value is passed in a 32-bit variable, and its unit is milliseconds.

There are two dedicated timeout values, for which it is recommended to use the pre-defined macros:

```
#define GOS_MUTEX_ENDLESS_TMO  ( 0xFFFFFFFFu )
#define GOS_MUTEX_NO_TMO       ( 0x00000000u )
```

With these macros the user can either “wait forever” (until the resource is available) or not wait at all. Any other value is a valid, “normal” timeout value, meaning that the lock function will return after the timeout value has elapsed regardless of the success of the locking.

5. Usage

Mutexes are allocated outside of the OS (in the user space), therefore there is no explicit limitation in the number of mutex instances in the system. Mutexes are passed to the OS by their pointer for operations such as: initialization, locking, and unlocking.

Mutexes shall be initialized before the first use by calling **gos_mutexInit**. A mutex can be obtained by calling **gos_mutexLock** and released by calling **gos_mutexUnlock**.

It is important to note that only the owner task has the right to release a mutex.

Here is a full example on how to create and use a mutex:

```
GOS_STATIC gos_mutex_t i2cMutex;

gos_result_t i2c_driver_init (void_t)
{
        /*
         * Local variables.
         */
        gos_result_t i2cDriverInitResult = GOS_ERROR;

        /*
         * Function code.
         */
        // TODO: initialize periphery.

        // Initialize mutex.
        i2cDriverInitResult = gos_mutexInit(&i2cMutex);

        return i2cDriverInitResult;
}
```

```
GOS_INLINE gos_result_t i2c_driver_transmit (u16_t address, u8_t* data, u16_t size)
{
        /*
         * Local variables.
         */
        gos_result_t i2cDriverTransmitResult = GOS_ERROR;


        /*
         * Function code.
         */
        if (gos_mutexLock(&i2cMutex, 1000u) == GOS_SUCCESS)
        {
                // TODO: Implement I2C transmission.
        }

        (void_t) gos_mutexUnlock(&i2cMutex);

        return i2cDriverTransmitResult;
}
```

6. API documentation

6.1. gos_mutexInit()

```
/**
 * @brief   Initializes the mutex instance.
 * @details Sets the mutex state to unlocked.
 *
 * @param   pMutex : Pointer to the mutex to be initialized.
 *
 * @return  Result of initialization.
 *
 * @retval  GOS_SUCCESS : Mutex initialized successfully.
 * @retval  GOS_ERROR   : Mutex pointer is NULL.
 */
gos_result_t gos_mutexInit (
        gos_mutex_t* pMutex
        );
```

6.2. gos_mutexLock()

```
/**
 * @brief   Tries to lock the given mutex with the given timeout.
 * @details Waits in an unblocking way until the mutex is unlocked or
 *          the timeout value is reached. If the mutex becomes unlocked
 *          within the timeout value, it locks it.
 *
 * @param   pMutex  : Pointer to the mutex to be locked.
 * @param   timeout : Timeout value.
 *
 * @return  Result of mutex locking.
 *
 * @retval  GOS_SUCCESS : Mutex locked successfully.
 * @retval  GOS_ERROR   : Mutex could not be locked within the timeout value.
 */
gos_result_t gos_mutexLock (
        gos_mutex_t* pMutex,
        u32_t        timeout
        );
```

6.3. gos_mutexUnlock()

```
/**
 * @brief   Unlocks the mutex instance.
 * @details Sets the mutex state to unlocked.
 *
 * @param   pMutex : Pointer to the mutex to be unlocked.
 *
 * @return  Result of mutex unlocking.
 *
 * @retval  GOS_SUCCESS : Unlocking successful.
 * @retval  GOS_ERROR   : Mutex is NULL or caller is not the owner of the mutex.
 */
gos_result_t gos_mutexUnlock (
        gos_mutex_t* pMutex
        );
```