Specification

# Objectives

An objective of this test suite is to confirm user mode functionality of lightweight event group, lightweight message queue, and lightweight semaphore according to Usermode\_design, development phase 2.

## Reference documentation

| Document Name | Version |
| --- | --- |
| 1. MQX MFS User Guide | 2.2 |
| 1. Usermode\_design | SHEET |
| 1. Usermode\_study | SHEET |

Table 1. Reference documentation

## Test environment

|  |  |
| --- | --- |
| **Software Item Name** | **Version** |
| IAR ARM | 6.10.1 |

Table 2. Required software resources

|  |  |
| --- | --- |
| **Hardware Item Name** | **Version** |
| TWR – K60N512 | N/A |

Table 3. Required hardware resources

# API

## Lightweight event group API

|  |  |
| --- | --- |
| *API name* | *Test app containing API* |
| \_usr\_lwevent\_clear | lwevent1, lwevent2, lwevent3 |
| \_usr\_lwevent\_set | lwevent1, lwevent2, lwevent3 |
| \_usr\_lwevent\_set\_auto\_clear | lwevent1, lwevent3 |
| \_usr\_lwevent\_wait\_for | lwevent2, lwevent3 |
| \_usr\_lwevent\_wait\_ticks | lwevent1, lwevent2, lwevent3 |
| \_usr\_lwevent\_wait\_until | lwevent2, lwevent3 |
| \_usr\_lwevent\_get\_signalled | lwevent1 |
| \_usr\_lwevent\_create | lwevent1, lwevent2, lwevent3 |
| \_usr\_lwevent\_destroy | lwevent1, lwevent2, lwevent3 |

## Lightweight message queue API

|  |  |
| --- | --- |
| *API name* | *Test app containing API* |
| \_usr\_lwmsgq\_init | lwmsgq1, lwmsgq2 |
| \_usr\_lwmsgq\_send | lwmsgq1, lwmsgq2 |
| \_usr\_lwmsgq\_receive | lwmsgq1, lwmsgq2 |

## Lightwight semaphore API

|  |  |
| --- | --- |
| *API name* | *Test app containing API* |
| \_usr\_lwsem\_poll | lwsem4 |
| \_usr\_lwsem\_post | lwsem1, lwsem2, lwsem4 |
| \_usr\_lwsem\_wait | lwsem1, lwsem2, lwsem3, lwsem4 |
| \_usr\_lwsem\_wait\_for | lwsem1, lwsem2, lwsem4 |
| \_usr\_lwsem\_wait\_ticks | lwsem1, lwsem2, lwsem4 |
| \_usr\_lwsem\_wait\_until | lwsem1, lwsem2, lwsem4 |
| \_usr\_lwsem\_create | lwsem1, lwsem2, lwsem3, lwsem4 |
| \_usr\_lwsem\_destroy | lwsem1, lwsem2, lwsem4 |

## General API

|  |  |
| --- | --- |
| *API name* | *Test app containing API* |
| \_mem\_set\_pool\_access | mpu |

# specifications

User mode test are based on existing tests for lightweight event group, lightweight message queue and lightweight semaphore. In addition test applications are added to test allowed access to global variables from user tasks and synchronization between user tasks and privilege tasks.

## Test app – LWEVENT1

This application tests the basics operations with lightweight event group in user mode. Synchronization is tested only between user tasks.

### Test case #1 – Destruction of uninitialized event

Attempt to call destroy function with uninitialized (zero-filed) LWEVENT\_STRUCT as parameter

Expected result: other than MQX\_OK

API used: \_usr\_lwevent\_destroy

### Test case #2 – Create the lwevent

Standard creation of lwevent object

Expected result: MQX\_OK

API used: \_usr\_lwevent\_create

### Test case #3 – Wait in ISR

This test case tries to call lwevent waiting function in ISR. ISR is simulated with direct write into kernel data, in member IN\_ISR. Since user tasks are not allowed to do so, another privilege task is created to perform this operation (ISR\_TASK). ISR\_TASK is created prior main test task and wait until this test case unblock it. Block/unblock of ISR\_TASK is performed by using other lwevent (isr\_lwevent)

Expected result: MQX\_CANNOT\_CALL\_FUNCTION\_FROM\_ISR

API used: \_usr\_lwevent\_set, \_usr\_lwevent\_wait\_ticks, \_usr\_lwevent\_create

### Test case #4 – Destroy lwevent with tasks waiting

There are 4 tasks created (WAIT\_TASK) that are blocked using lwevent wait function. After that, lwevent is destroyed. Waiting tasks should be rescheduled and correctly ended. Every waiting task increment COUNTER.

Expected result: after wait tasks creation lwevent.WAITING\_TASKS.SIZE must equal 4. After lwevent destruction COUNTER must be 4. This indicates that all waiting tasks was rescheduled and properly finished. Destroy function must return MQX\_OK

API used: \_usr\_lwevent\_create, \_usr\_lwevent\_destroy \_usr\_lwevent\_wait\_ticks

### Test case #5 – Invalidate LWEVENT\_STRUCT

This test case tries to invalidate LWEVENT\_STRUCT with direct writes into its members. Since LWEVENT\_STRUCT must be declared as read-only for user tasks, these attempts results with exceptions. Therefore an exception handler is added into application (FAULT\_ISR). This handler increments error counter and test case evaluates this counter.

Expected result: every direct write into LWEVENT\_STRUCT must throw exception (error counter is increased)

API used: none

### Test case #6 – Normal operation

For this test scenario another test tasks are created: LWEVENT\_TASK. Number of LWEVENT\_TASK depends on NUMBER\_OF\_TASKS macro value. After creation of all lwevent\_tasks lwevent bit on position 0 is set. Test case also tests AutoClear functionality. Every even bit from lwevent is set to auto-clear.

LWEVENT\_TASK description: tasks wait for a specific bit from lwevent. Bit position depends on INDEX passed from test case. After this bit is set, task checks the auto-clear functionality. Order of execution is stored into global variable count[]. After that task set the next bit in lwevent (bit with position INDEX+1) i.e. it unblocks next waiting instance of LWEVENT\_TASK.

Expected result: every even bit from lwevent must be cleared after task waiting on this bit is unblocked. Waiting tasks are executed in order as they waits on specific bit in lwevent: task waiting on bit 0 is executed first, task waiting on bit 1 is second, and task waiting on bit 2 is third and so on.

API used: \_usr\_lwevent\_wait, \_usr\_lwevent\_wait\_ticks, \_usr\_lwevent\_clear, \_usr\_lwevent\_set\_auto\_clear, \_usr\_lwevent\_get\_signalled, \_usr\_lwevent\_set, \_usr\_lwevent\_clear

### Test case #7 – Normal operation, different priorities

Test case is basically same as test case #6, besides that LWEVENT\_TASKS are created with higher and higher priority.

Expected result: same as test case #6, task priority rising cannot have impact on order of execution

## Test app - LWEVENT2

Application contains time specific tests for lightweight event group.

### Test case # 1 – Creation of lwevent

In this test case a lwevent for next test cases is created

Expected result: MQX\_OK

API used: \_usr\_lwevent\_create

### Test case #2 – Wait ticks

Wait\_ticks function is called with specified number of ticks to wait on.

Expected result: difference between elapsed time before and after wait function call must be equal to time specified in wait function. Time is expressed in ticks. Wait function must return LWEVENT\_WAIT\_TIMEOUT

API used: \_usr\_lwevent\_wait\_ticks

### Test case #3 – Wait for

Wait\_for function is called with specified amount of time to wait on.

Expected result: time difference between time before and after wait function call must be equal to time amount specified in wait function. Wait function returns LWEVENT\_WAIT\_TIMEOUT

API used: \_usr\_lwevent\_wait\_for

### Test case #4 – Wait until

Time in future is calculated and wait\_until function is called.

Expected result: calculated time as parameter and time after wait\_until function time-outs must be same. Wait\_until function must return LWEVENT\_WAIT\_TIMEOUT

API used: \_usr\_lwevent\_wait\_until

### Test case #5 – Timeout and destroy

For this test case another user task is created: LWEVENT\_TASK. This task, perform operations based on return code from wait\_ticks function:

* MQX\_OK: lwevent\_count++
* LWEVENT\_WAIT\_TIMEOUT: lwevent\_timeout++
* MQX\_LWEVENT\_INVALID: \_task\_destroy(0)

After LWEVENT\_TASK is created, test application calls \_time\_delay\_ticks, so wait time specified in LWEVENT\_TASK expires. After that lwevent is destroyed.

Expected result: lwevent\_count must be 1, lwevent\_timeout must be greater than 0 (exact value depends on parameter used in \_time\_delay\_ticks). LWEVENT\_TASK must be destroyed after lwevent is not valid (destroyed)

API used: \_usr\_lwevent\_wait\_ticks, \_usr\_lwevent\_set, \_usr\_lwevent\_destroy

## Test app – LWEVENT3

This application tests synchronization between user and privilege tasks using lightweight event group.

### Test case # 1 – Accessing prohibided LWEVENT\_STRUCTs

Test case contains various attempts to use lwevents that are not allowed to use in user tasks. User tasks can use only lwevent declared as read-only and created with \_usr\_lwevent\_create function. Creation can be done either in user or privilege task. Test includes operations on these lwevents: no access, read-only, read-write created in privilege task and read-write created in user task. This test case tests also usage of MQX\_ENABLE\_USER\_STDAPI, which allows calling user mode API without \_usr prefix.

Expected result: other than MQX\_OK

API used: \_usr\_lwevent\_create, \_usr\_lwevent\_set, \_usr\_lwevent\_clear, \_usr\_lwevent\_set\_auto\_clear, \_usr\_lwevent\_wait\_ticks, \_usr\_lwevent\_wait\_for, \_usr\_lwevent\_wait\_until, \_usr\_lwevent\_destroy

### Test case #2 – User task/Privilege task synchronisation

For test purposes there is one privilege task created. User and privilege tasks are synchronizing each other; evaluating and modifying global variable. Privilege task is using both privilege and user API (\_usr prefix). Test case is using access to usermode lwevents from user tasks and privilege tasks. Lwevents are created in both user task and privilege task.

Expected result: global variable must be altered in correct way

API used: \_usr\_lwevent\_wait\_for(until), \_usr\_lwevent\_set,

## Test app – LWMSGQ1

Application tests usage of lightweight message queue synchronization between user tasks only.

### Test case #1 – Basic operations

This test case utilizes 6 types of tasks using one lwmsgq (3 for writing and 3 for reading) each one with different synchronization mechanism.

Synchronization mechanism tested on write tasks:

* NONE
* LWMSGQ\_SEND\_BLOCK\_ON\_FULL
* LWMSGQ\_SEND\_BLOCK\_ON\_SEND

Synchronization mechanism tested on read tasks:

* NONE
* LWMSGQ\_RECEIVE\_BLOCK\_ON\_EMPTY
* LWMSGQ\_TIMEOUT\_FOR
* LWMSGQ\_TIMEOUT\_UNTIL

Each task has its own error counter and passes counter

Expected result: each error count must be 0, pass counter must be greater than 0

API used: \_usr\_lwmsgq\_init, \_usr\_lwmsgq\_send, \_usr\_lwmsgq\_receive

## Test app – LWMSGQ2

### Test case #1 -- Accessing prohibided LWMSGQ\_STRUCTs

Test case contains various attempts to use lwmsgqs that are not allowed to use in user tasks. User tasks can use only lwmsgq declared as read-only and created with \_usr\_lwmsgq\_init function. Creation can be done either in user or privilege task. Test includes operations on these lwmsgqs: no access; read-only, read-write created in privilege task and read-write created in privilege task. This test case tests also usage of MQX\_ENABLE\_USER\_STDAPI, which allows calling user mode API without \_usr prefix.

Expected result: other than MQX\_OK

API used: \_usr\_lwmsgq\_init, \_usr\_lwmsgq\_send, \_usr\_lwmsgq\_receive

### Test case #2 – User task/Privilege task synchronisation

For test purposes there is one privilege task created. User and privilege tasks are synchronizing each other. Both tasks are evaluating and modifying message prior sending it to other task.

Expected result: message is altered as assumed

API used: \_usr\_lwmsgq\_init, \_usr\_lwmsgq\_send, \_usr\_lwmsgq\_receive

## Test app – LWSEM1

### Test case #1 – Destruction of uninitialized semaphore

Attempt to call destroy function upon uninitialized (zero-filed) LWSEM\_STRUCT

Expected results: other than MQX\_OK

API used: \_usr\_lwsem\_destroy

### Test case #2 – Creation of lwsem

Creation of lwsem for next test cases

Expected results: MQX\_OK

API used: \_usr\_lwsem\_create

### Test case #3 – Wait in ISR

This test case tries to call lwsem waiting function in ISR. ISR is simulated with direct write into kernel data, in member IN\_ISR. Since user tasks are not allowed to do so, another privilege task is created to perform this operation (ISR\_TASK). ISR\_TASK is created prior main test task and wait until this test case unblock it. Block/unblock of ISR\_TASK is performed by using other lwsem (isr\_lwsem)

Expected results: MQX\_CANNOT\_CALL\_FUNCTION\_FROM\_ISR

API used: \_usr\_lwsem\_post, \_usr\_lwsem\_wait\_for, \_usr\_lwsem\_wait\_ticks, \_usr\_lwsem\_wait\_until

### Test case #4 – Tasks waiting on semaphore

There are 4 tasks created (WAIT\_TASK) that are blocked using lwsem wait function. After that, lwsem is destroyed. Waiting tasks should be rescheduled and correctly ended. Every waiting task increment COUNTER.

Expected result: after wait tasks creation lwsem.TD\_QUEUE.SIZE must equal 4. After lwevent destruction COUNTER must be 4. This indicates that all waiting tasks was rescheduled and properly finished. Destroy function must return MQX\_OK.

API used: \_usr\_lwsem\_create, \_usr\_lwsem\_destroy, \_usr\_lwsem\_wait\_ticks

### Test case #5 – Invalidating LWSEM\_STRUCT with dirrect write

This test case tries to invalidate LWSEM\_STRUCT with direct writes into its members. Since LWSEM\_STRUCT must be declared as read-only for user tasks, these attempts results with exceptions. Therefore an exception handler is added into application (FAULT\_ISR). This handler increments error counter and test case evaluates this counter.

Expected result: every direct write into LWSEM\_STRUCT must throw exception (error counter is increased)

API used: none

### Test case #6 – Normal operation

For this test scenario another test tasks are crated: LWSEM\_TASK. Number of LWSEM\_TASK depends on NUMBER\_OF\_TASKS macro value. After creation of all lwsem\_tasks, lwsem is posted.

LWSEM\_TASK description: task waits for lwsem. After this lwsem is posted, order of execution is stored into global variable count[]. Task then posts the lwsem.

Expected result: All waiting tasks must be rescheduled in order as they were created.

API used: \_usr\_lwsem\_wait, \_usr\_lwsem\_post

### Test case #7 – Normal operation, different priorities

Same as test case #6, but LWSEM\_TASKs are created with higher and higher priority.

Expected results: All waiting tasks must be executed in order as they are waiting for semaphore. Different priority may not have impact on order of execution.

API used: \_usr\_lwsem\_wait, \_usr\_lwsem\_post

## Test app – LWSEM2

Application contains time specific tests for lightweight semaphore.

### Test case #1 – Creation of lwsem

Creation of lwsem used in next test cases.

Expected results: MQX\_OK

API used: \_usr\_lwsem\_create

### Test case #2 – Wait ticks

Wait\_ticks function is called with specified number of ticks to wait on.

Expected results: difference between elapsed time before and after wait function call must be equal to time specified in wait function. Time is expressed in ticks. Wait function must return LWSEM\_WAIT\_TIMEOUT

API used: \_usr\_lwsem\_ticks

### Test case #3 – Wait for

Wait\_for function is called with specified amount of time to wait on.

Expected results: time difference between time before and after wait function call must be equal to time amount specified in wait function. Wait function returns LWSEM\_WAIT\_TIMEOUT

API used: \_usr\_lwsem\_wait\_for

### Test case #4 – Wait until

Time in future is calculated and wait\_until function is called.

Expected results: calculated time as parameter and time after wait\_until function time-outs must be same. Wait\_until function must return LWEVENT\_WAIT\_TIMEOUT

API used: \_usr\_lwsem\_wait\_until

### Test case #5 – Time-out and destroy

For this test case another user task is created: LWSEM\_TASK. This task, perform operations based on return code from wait\_ticks function:

* MQX\_OK: lwsem\_count++
* LWSEM\_WAIT\_TIMEOUT: lwsem\_timeout++
* MQX\_LWSEM\_INVALID: \_task\_destroy(0)

After LWSEM\_TASK is created, test application post lwsem two times. Then it calls \_time\_delay\_ticks, so lwsem in LWSEM\_TASK expires. After that lwsem is destroyed.

Expected result: lwsem\_count must be 2 (lwsem is posted twice in test app), lwsem\_timeout must be greater than 0 (exact value depends on parameter used in \_time\_delay\_ticks). LWSEM\_TASK must be destroyed after lwsem is not valid (destroyed)

API used: \_usr\_lwsem\_post, \_usr\_lwsem\_wait\_ticks, \_usr\_lwsem\_destroy

## Test app – LWSEM3

### Test case #1 – Aborting waiting tasks

For test purposes another user task is created LWSEM\_TASK. This task is blocked by calling wait function. Test application creates two instances of LWSEM\_TASK and checks lwsem.TD\_QUEUE.SIZE (number of tasks waiting for semaphore)

Expected results: at the beginning lwsem.TD\_QUEUE.SIZE must be 0. With every new LWSEM\_TASK created this value must increase by 1. With every LWSEM\_TASK destroyed, this value must decrease by 1.

API used: \_usr\_lwsem\_create, \_usr\_lwsem\_wait

## Test app – LWSEM4

### Test case #1 – Accessing prohibided LWMSGQ\_STRUCTs

Test case contains various attempts to use lwsems that are not allowed to use in user tasks. User tasks can use only lwsem declared as read-only and created with \_usr\_lwsem\_create function. Creation can be done either in user or privilege task. Test includes operations on these lwsems: no access, read-only, read-write created in privilege task and read-write created in privilege task. This test case tests also usage of MQX\_ENABLE\_USER\_STDAPI, which allows calling user mode API without \_usr prefix.

Expected results: other than MQX\_OK

API used: \_usr\_lwsem\_poll, \_usr\_lwsem\_post, \_usr\_lwsem\_wait, \_usr\_lwsem\_wait\_for, \_usr\_lwsem\_wait\_ticks, \_usr\_lwsem\_wait\_until, \_usr\_lwsem\_create, \_usr\_lwsem\_destroy

### Test case #2 – User task/Privilege task synchronisation

For test purposes there is one privilege task created. User and privilege tasks are synchronizing each other; evaluating and modifying global variable. Privilege task is using both privilege and user API (\_usr prefix). Test case is using access to usermode lwsems from user tasks and privilege tasks. Lwsems are created in both user task and privilege task.

Expected result: global variable must be altered in correct way

API used: \_usr\_lwsem\_wait, \_usr\_lwsem\_post, \_usr\_lwsem\_create, \_usr\_lwsem\_wait\_for, \_usr\_lwsem\_wait\_until

## Test app – MEM\_ACCESS

This application tests accesses to global variables defined for user task as read-write, read-only and “no access”. Exceptions thrown by forbidden access is handled in application. Handler increase error counter and test application evaluate its value.

Expected results: exceptions must be thrown when:

* Writing into read-only variable
* Reading or writing in/from “no access” variable

API used: none

## Test app – MPU

### Test case #1 – Setting MPU on single memory pool

This test case is setting user mode access to on single memory pool.

Expected result: each device with MPU functionality has only limited amount of MPU records (e.g. K60 have 12 records). When all available records are used an error should be returned

API used: \_mem\_set\_pool\_access

### Test case #2 – Setting MPU on multiple memory pools

This test case is using same functionality as previous test case with one difference: new memory pools are created and MPU records are used.

Expected result: each device with MPU functionality has only limited amount of MPU records (e.g. K60 have 12 records). When all available records are used an error should be returned

API used: \_mem\_set\_pool\_access

### Test case #3 – Dynamic MPU setting

For purpose of testing, two new task are needed PRIV\_TASK, USER\_TASK. PRIV\_TASK allocate memory from memory marked as “no access” for user tasks (USER\_NO\_ACCESS). This setting is important, because MPU can only grand access rights not revoke them. In allocated memory a memory pool is created. PRIV\_TASK is setting various restrictions for usermode tasks: (NO, RO, RW) and USER\_TASK is performing read, write and alloc operations upon this memory pool. PRIV\_TASK and USER\_TASK are synchronized using lightweight event.

Expected results:

* NO\_ACCESS: read – fail, write – fail, alloc – fail
* RO\_ACCESS: read – pass, write – fail, alloc – fail
* RW\_ACCESS: read – pass, write – pass, alloc – pass

API used: \_mem\_set\_pool\_access

# known issues

Test-apps lwsem1, lwevent1 contains statements that try to invalidate LWSEM\_STRUCT and LWEVENT\_STRUCT respectively. These attempts result in exceptions, because structures are read-only for user tasks. Exceptions are handled in test-apps directly. Currently this handling is hard-wired to K60 specific platform, porting these test-apps to new platform(s) must resolve this specifics.

# possible improvements

Currently there are no tests to validate proper task manipulating (create, destroy, block …) and scheduling in user mode.

**Revision SHEET**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revision Date** | **Author** | **Description of Revision & Writer** | **Spec Coord.** |
| 21.9.2011 | Michal Starecek | Initial version |  |
| 3.10.2011 | Michal Starecek | Comments from Jaromir Jasik applied |  |
| 15.11.20011 | Michal Starecek | MPU test added |  |