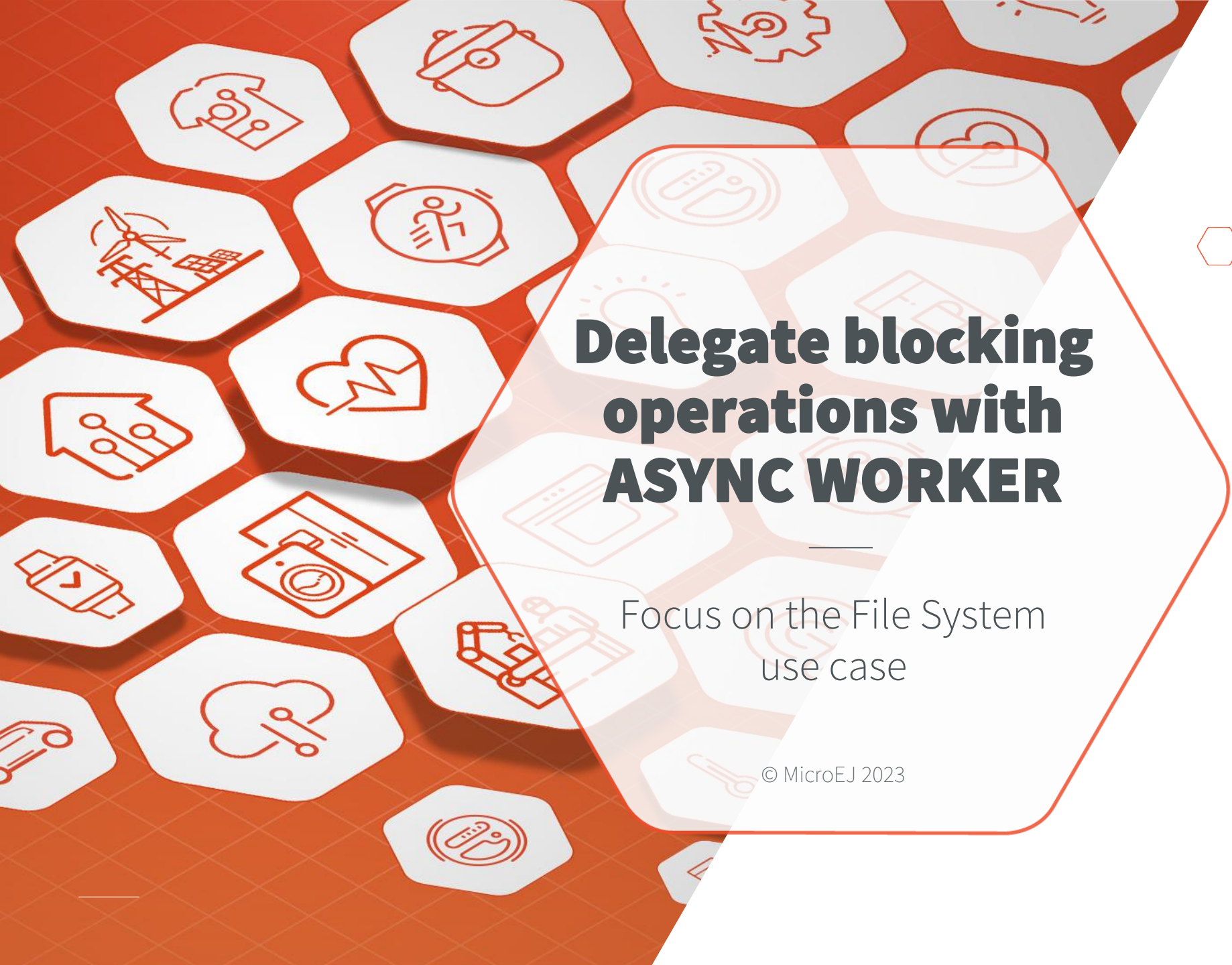


Delegate blocking operations with ASYNC WORKER

Focus on the File System use case

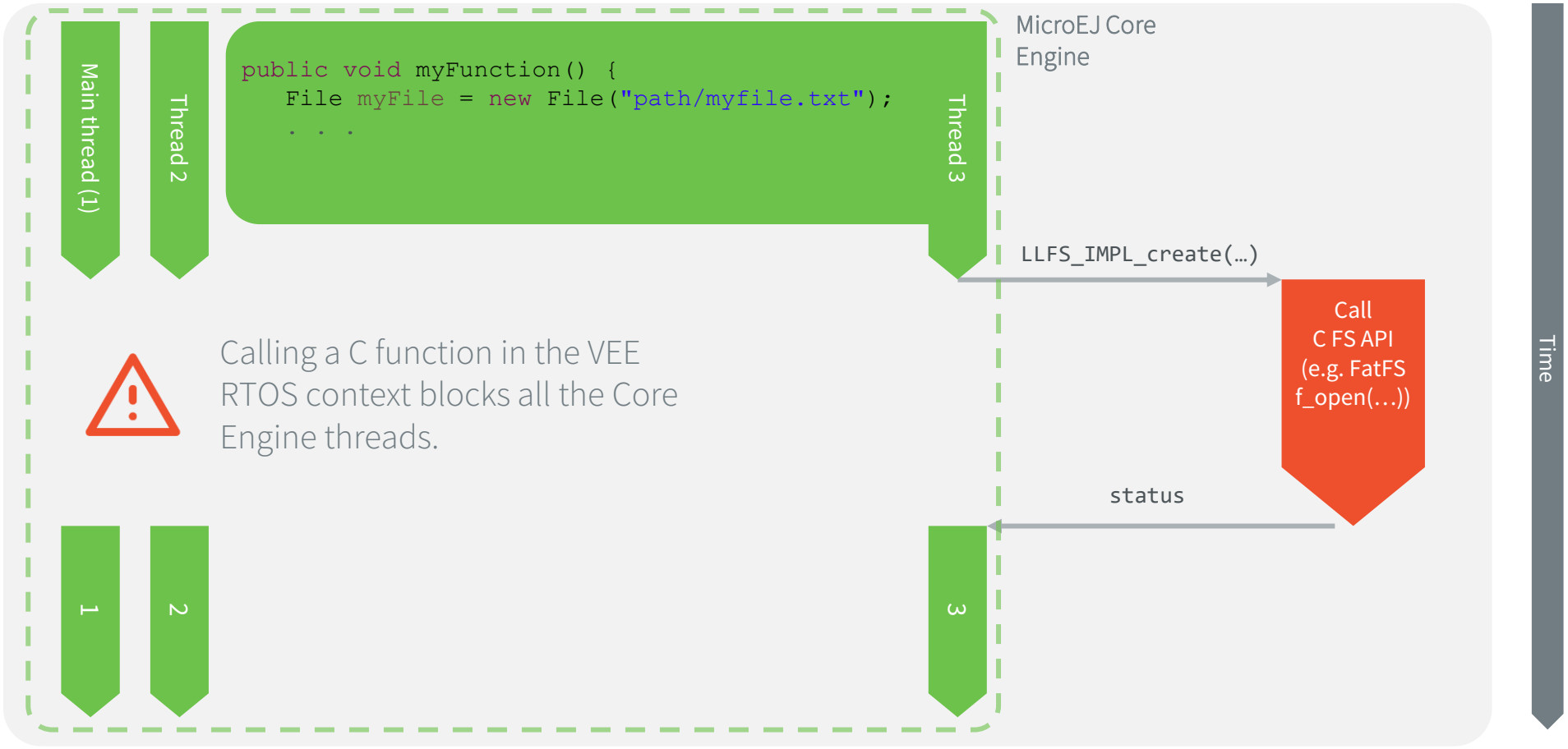
MICROEJ®

OVERVIEW

- MicroEJ Async Worker is a C Component that helps to delegate blocking operations from VEE RTOS context to another RTOS task context.
- This component relies on the SNI mechanism.
- This presentation describes the Async Worker and illustrates its use with in File System implementation.
- Async Worker is available on MicroEJ Developer Repository.

BLOCKING SCENARIO

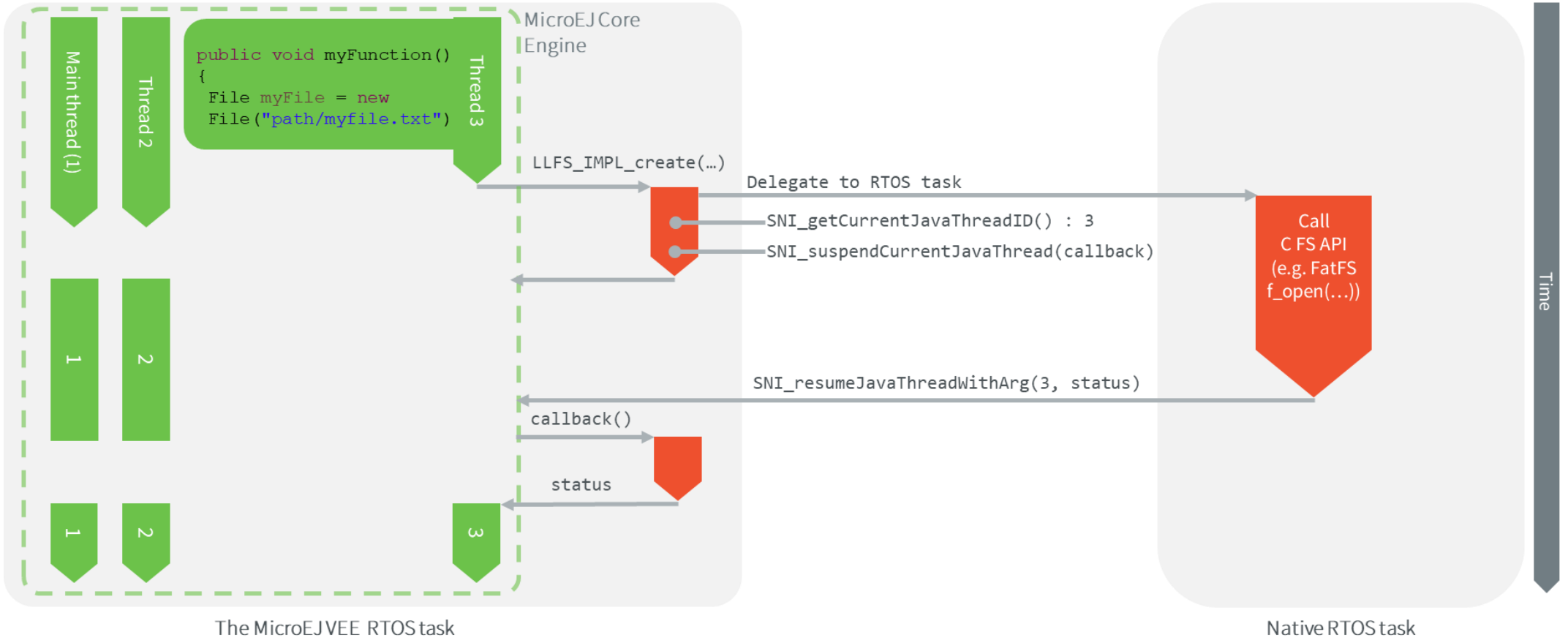
- Create a file on the filesystem partition (or SD Card):



The MicroEJ VEE RTOS task

NON-BLOCKING SCENARIO

- Create a file on the filesystem partition (or SD Card):

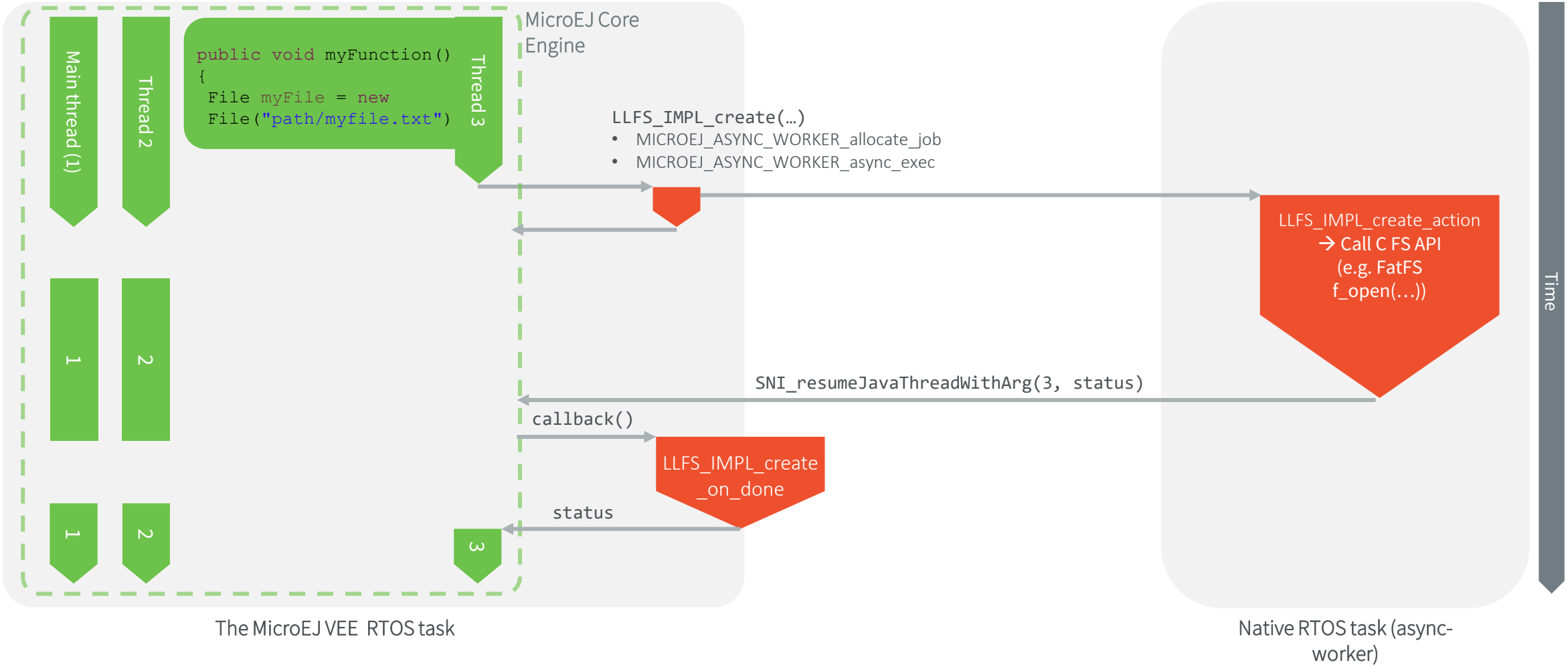


- Blocking actions are delegated to a dedicated RTOS task.
- The Async Worker can be used to ease the implementation of this mechanism.

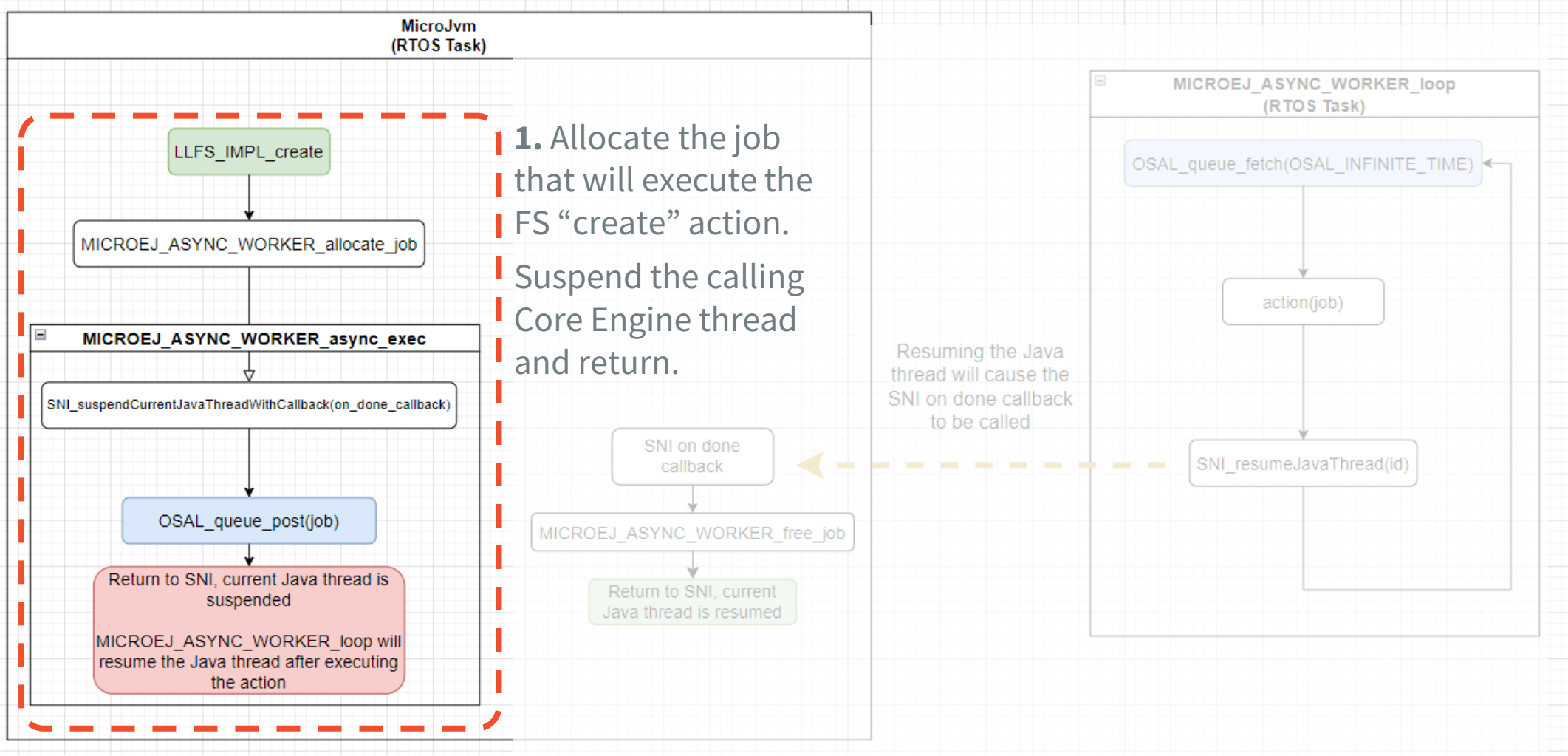
Non-Blocking Scenario Using Async-Worker

ASYNC-WORKER SCENARIO

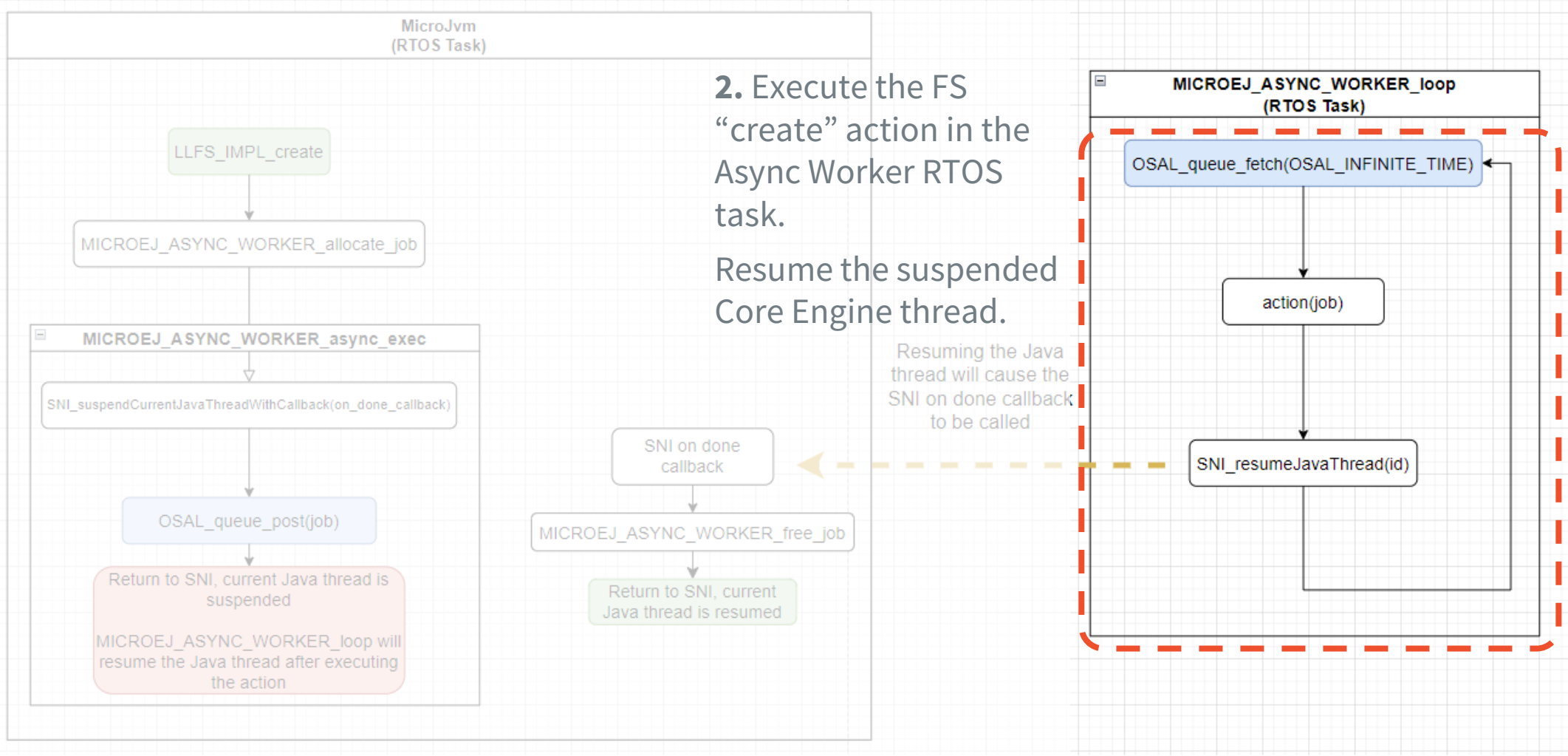
- Create a file on the filesystem partition (or SD Card):



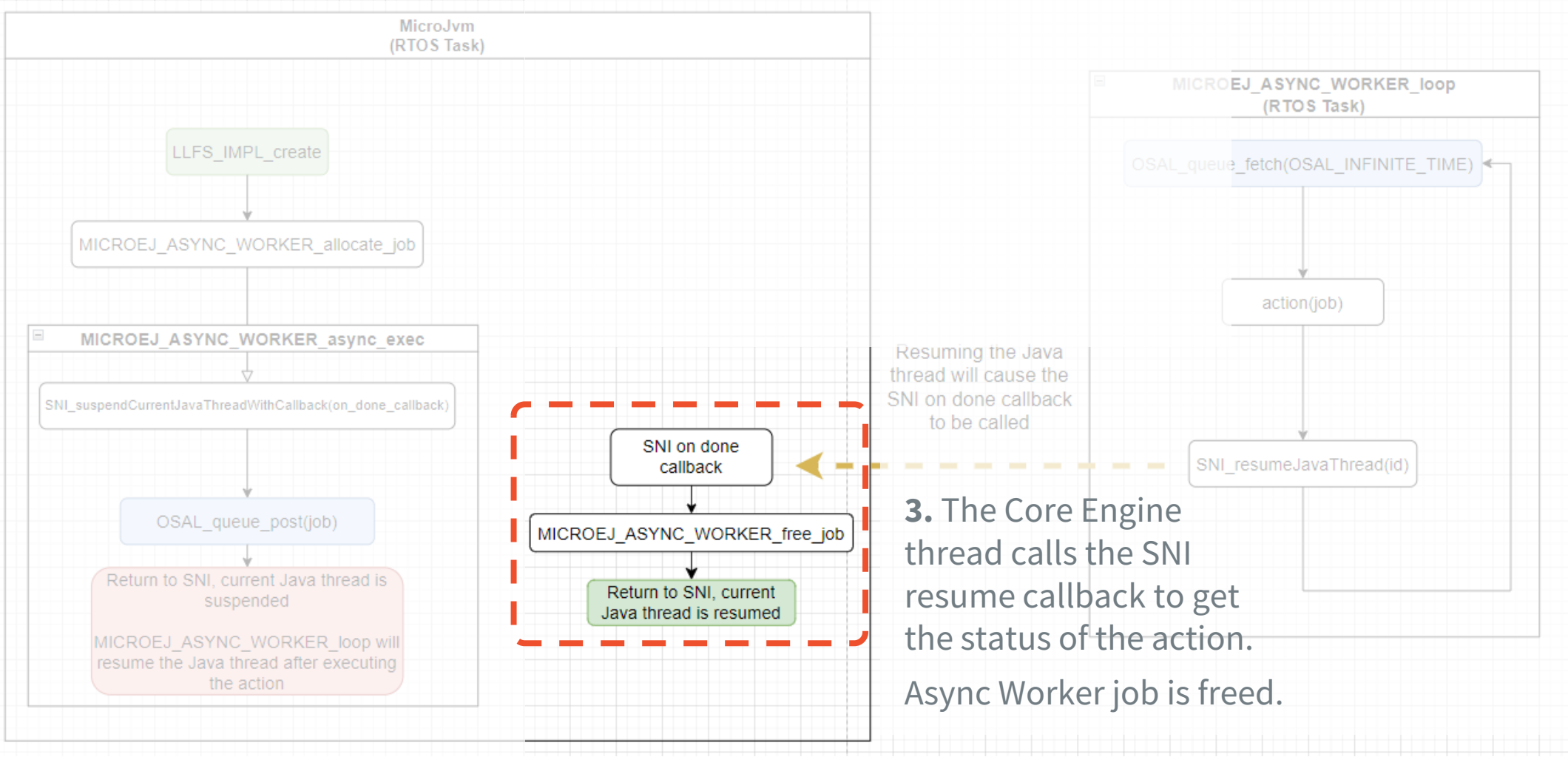
ASYNC-WORKER SIMPLIFIED FLOW DIAGRAM (1/3)



ASYNC-WORKER SIMPLIFIED STATE DIAGRAM (2/3)



ASYNC-WORKER SIMPLIFIED STATE DIAGRAM (2/3)



3. The Core Engine thread calls the SNI resume callback to get the status of the action.

Async Worker job is freed.

Implementation Example

Extracted from STM32F7508-DK
VEE PORT 1.5.0

NATIVE FUNCTION IMPLEMENTATION

```c
int32_t LLFS_IMPL_create(uint8_t* path){
    MICROEJ_ASYNC_WORKER_job_t* job = MICROEJ_ASYNC_WORKER_allocate_job(&fs_worker, (SNI_callback)LLFS_IMPL_create);
    if(job == NULL){
        // No job available, either:
        // - wait for a job to be available and this function to be executed again,
        // - or an exception is pending
        return LLFS_NOK;
    }

    FS_path_operation_t* params = (FS_path_operation_t*)job->params;
    if(LLFS_set_path_param(path, (uint8_t*)&params->path) != LLFS_OK){
        SNI_throwNativeIOException(LLFS_NOK, "Path name too long");
    }
    else{
        MICROEJ_ASYNC_WORKER_status_t status = MICROEJ_ASYNC_WORKER_async_exec(&fs_worker, job, LLFS_IMPL_create_action, (SNI_callback)LLFS_IMPL_create_on_done);
        if(status == MICROEJ_ASYNC_WORKER_OK){
            // Wait for the action to be done
            return SNI_IGNORED_RETURNED_VALUE;//returned value not used
        } // else an error occurred and MICROEJ_ASYNC_WORKER_async_exec has thrown a SNI exception
    }

    // Error
    MICROEJ_ASYNC_WORKER_free_job(&fs_worker, job);
    return LLFS_NOK;
}
```

Retry function in case the job cannot be allocated on the worker.

Set job parameters.

Return value not used as the SNI thread is suspended. It's real return value is set by the SNI callback.

Function executed asynchronously in the worker's context.

SNI Callback called once the action is done and the SNI thread is resumed.

Source code available in LLFS_impl.c

ACTION FUNCTION IMPLEMENTATION

```
void LLFS_IMPL_create_action(MICROEJ_ASYNC_WORKER_job_t* job) {

    FS_create_t* param = (FS_create_t*) job->params;
    FIL fp = {0};
    FRESULT res = FR_OK;

    uint8_t* path = (uint8_t*)&param->path;

    res = f_open(&fp, (TCHAR*)path, FA_CREATE_NEW);
    if (res == FR_OK) {
        res = f_close(&fp);
        if (res == FR_OK) {
            param->result = LLFS_OK;
        } else {
            param->result = LLFS_NOK;
            param->error_code = res;
            param->error_message = "f_close failed";
        }
    } else if (res == FR_EXIST) {
        param->result = LLFS_NOT_CREATED;
        param->error_code = res;
        param->error_message = "file exists";
    } else {
        param->result = LLFS_NOT_CREATED;
        param->error_code = res;
        param->error_message = "f_open failed";
    }

    LLFS_DEBUG_TRACE("[%s:%u] create file %s (err %d)\n", __func__, __LINE__, path, res);
}
```

Get job parameters set by the SNI native function.

Perform blocking / long API execution.

Return data / status in the same job parameters structure.

Source code available in fs_helper_FatFs.c

SNI CALLBACK IMPLEMENTATION

```
static int32_t LLFS_IMPL_create_on_done(uint8_t* path){
    MICROEJ_ASYNC_WORKER_job_t* job = MICROEJ_ASYNC_WORKER_get_job_done();
    FS_create_t* params = (FS_create_t*)job->params;

    int32_t result = params->result;
    if(result == LLFS_NOK){
        // Exception
        SNI_throwNativeIOException(params->error_code, params->error_message);
    }
    MICROEJ_ASYNC_WORKER_free_job(&fs_worker, job);

    return result;
}
```

Get job parameters that contains the result of the action.

Return to the Core Engine thread.

Source code available in LLFS_impl.c

APPENDIX

ASYNC-WORKER SPECIFICATION (1/6)

MICROEJ

Declaration

The worker is declared statically using the *MICROEJ_ASYNC_WORKER_worker_declare* macro, containing the following parameters:

- _name: Name of the async-worker variable.
- _job_count: Maximum number of jobs that can be allocated for this worker. Since a job represents an asynchronous execution requested from an SNI native, and it blocks the caller Java thread, the job count parameter can be interpreted as “Maximum number of Java threads that can call simultaneously native functions running on the same async-worker, e.g. filesystem native functions”.
- _param_type: An union type to hold all the parameters structure.
- _waiting_list_size: Waiting list size, holding a number of Java threads that are suspended when no job is available. This parameter can be interpreted as “If more Java threads than _job_count are calling native functions running on the same async-worker, e.g. filesystem native functions, they will be suspended and added to a waiting FIFO of this size, and will be resumed once a free job is available on the worker.

ASYNC-WORKER SPECIFICATION (2/6)

MICROEJ

Initialization

During a MicroEJ Adaption Layer initialization at BSP side, the async-worker is initialized using the function *MICROEJ_ASYNC_WORKER_initialize*. Initialization parameters:

- async_worker: The worker to initialize, declared previously with *MICROEJ_ASYNC_WORKER_worker_declare.*
- name: The worker name, will be also the name of all RTOS components created by the worker: task, queue, mutex.
- stack: The worker's task stack, declared previously with *OSAL_task_stack_declare.*
- priority: The worker task priority.

Note

A worker is initialized once, usually during a native initialization function, e.g. Java_com_is2t_java_io_FileSystem_initializeNative, and active for the whole lifecycle of the application.

ASYNC-WORKER SPECIFICATION (3/6)

MICROEJ

Allocate job

After initialization of the worker, it is ready to receive jobs from any native function implementation of the respective module, e.g. *Java_com_is2t_java_io_FileSystem_createNative*.

First, the native function must allocate a job on the worker, used to pass parameters to/from SNI context and the worker RTOS context. Job allocation is done by *MICROEJ_ASYNC_WORKER_allocate_job*, and requires the following parameters:

- async_worker: The worker on which to allocate jobs, declared previously with *MICROEJ_ASYNC_WORKER_worker_declare.*
- sni_retry_callback: If there is no job available, the current Java thread is suspended, added to the waiting list and NULL is returned. Then, when a job is available, the Java thread is resumed and the function sni_retry_callback is called. Usually the sni_retry_callback function argument is the current SNI function itself.

ASYNC-WORKER SPECIFICATION (4/6)

MICROEJ

Execute job

After allocating a job on the worker, e.g. *Java_com_is2t_java_io_FileSystem_createNative*, the job is ready to be executed asynchronously. There are two APIs that can be used for that, with slightly different behavior:

- *MICROEJ_ASYNC_WORKER_async_exec_no_wait:* This function executes the given job asynchronously, on the worker RTOS task. It does not block and returns immediately. When the job is finished, the job is automatically released by the async worker RTOS task. This function can be called in the scenario “fire-and-forget”, when the SNI thread does not need to wait for it to finish its execution and does not need its returned data.
- *MICROEJ_ASYNC_WORKER_async_exec:* This function executes the given job asynchronously, on the worker RTOS task. It does not block and returns immediately but it suspends the execution of the current Java thread until the job is finished. When the job is finished, the SNI callback on_done_callback is called before going back to Java. If the job is not used anymore, the callback must release it explicitly by calling MICROEJ_ASYNC_WORKER_free_job.

ASYNC-WORKER SPECIFICATION (5/6)

MICROEJ

Execute parameters

The job execution needs the following parameters:

- async_worker: The worker used to execute the given job. Must be the same that the one used to allocate the job.
- job: The job to execute, must have been allocated with *MICROEJ_ASYNC_WORKER_allocate_job*
- action: The function to execute asynchronously.
- on_done_callback: The SNI_callback called when the job is done.

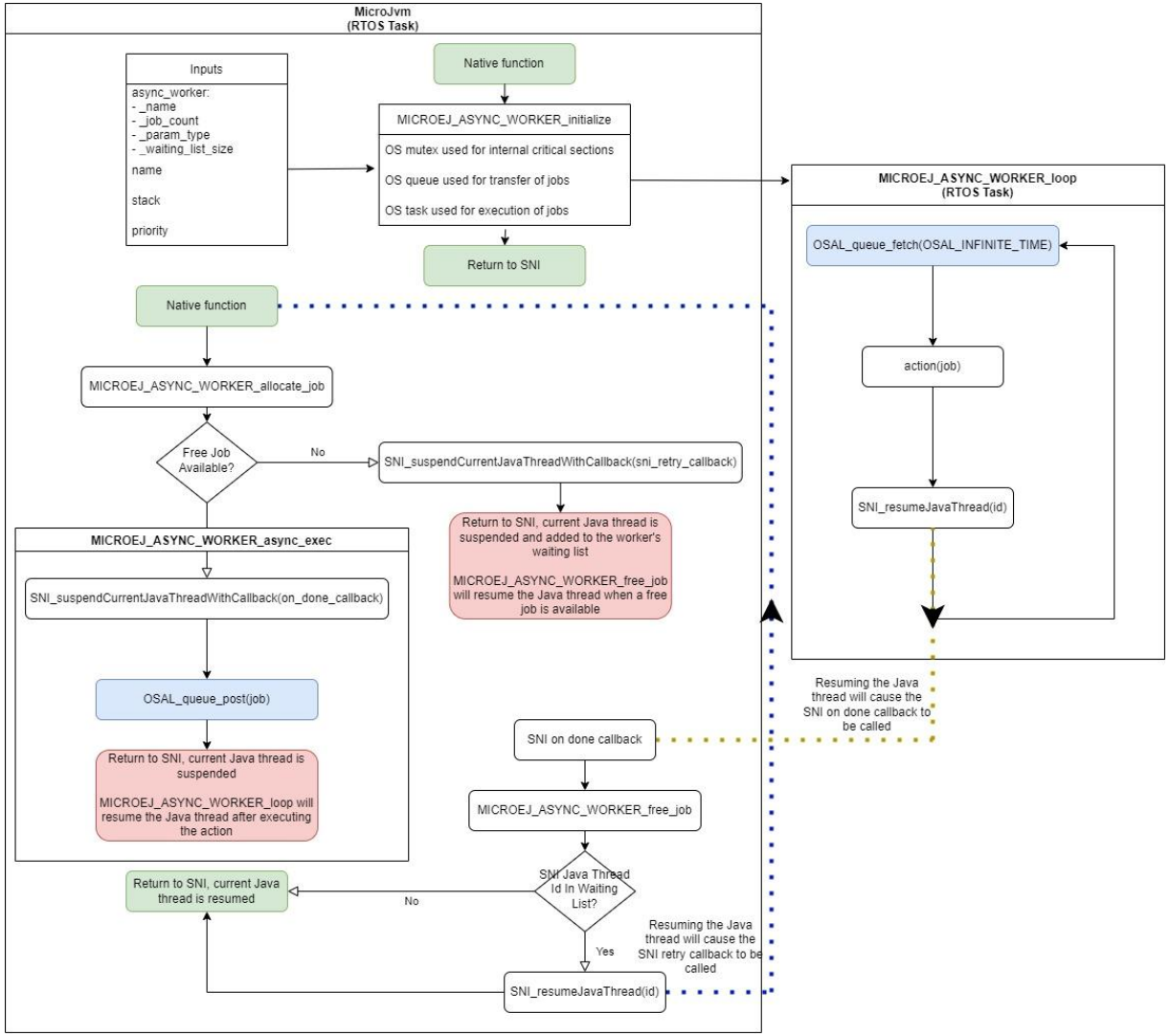
ASYNC-WORKER SPECIFICATION (6/6)

MICROEJ

Other APIs

- *MICROEJ_ASYNC_WORKER_free_job:* Frees a job previously allocated with *MICROEJ_ASYNC_WORKER_allocate_job*.
- *MICROEJ_ASYNC_WORKER_get_job_done:* Returns the job that has been executed.

ASYNC WORKER STATE DIAGRAM



THANK YOU

for your attention !

MICROEJ®