2025 IEEE/SICE International Symposium on System Integration (SII)
January 21-24, 2025, Munich, Germany

µ-RoMS-OS – An Operating System for a Robot Middleware Software for Low-Level-Microcontroller

Michael Zauner*, Roman Froschauer

Abstract— **This paper presents a multitasking system for low-level microcontrollers. The main focus in developing this operating system was to design an optimized software that does not have high requirements in terms of memory or computing time. Another requirement was to be able to port the system to different platforms very easily. Therefore, C was chosen as the programming language. Another focus was on the ease of use of the program. It should also be able to be used by less experienced programmers without a long training phase. The program is now used in a wide variety of projects, from robot systems to wide range of other applications, on different microcontrollers, such as Xmega256A3 or AVR128DB48.**

I. Introduction

Today, *ROS* [1] is a de facto standard in mobile, autonomous robotics. However, *ROS* was developed to run primarily under *Linux* on a powerful computer. In recent years, efforts have also been made to implement *ROS* on micro-controller platforms [2]. However, here too, only powerful 32-bit ARM-based processors are supported. The preferred operating system here is *FreeRTOS* from Amazon Web Services [3].

For many tasks, however, it is not necessary to use 32-bit microcontrollers. Very often, cheaper and more energy-efficient 8-bit microcontrollers can be used. This paper deals with an approach for an operating system for 8-bit micro-controllers. When using low-performance microcontrollers, however, the limited resources, such as limited memory size or clock rate, must be taken into account.

In [4] and [5], current operating systems for micro-controllers are compared and their advantages and disadvantages are discussed. The common procedures for implementing an operating system are also discussed. In principle, a distinction can be made between *super loops*, *cooperative* and *preemptive operating systems*. However, the operating systems presented do not meet the requirements in terms of memory needed for an 8-bit microcontroller. The operating system that comes closest to the requirements is *TinyOS* [6].

II. Functionality of Operating Systems for Microcontroller

A. Super-Loops

The operation of a super loop is kept very simple. In the main program, there is an endless loop that is not exited as soon as it is started for the first time. To guarantee an exact timing, a hardware-timer runs parallel to the super loop, which triggers variables for the corresponding time periods [7]. These variables are monitored in the super loop and as soon as a variable is set for the corresponding time unit, the tasks that run with this interval are started. This is illustrated in Fig. 1 and 2.

The following problems occur with this type of task processing:

- The program is very confusing and therefore difficult to maintain.
- A task is always assigned a fixed time; if other intervals are required between two calls of the task, these must be derived from the standard time in the task.
- If a task requires more time than the minimum interval time, this affects all other tasks.

Such a system can only be implemented reasonably for very simple programs.

```
/* main-function */
void main()
{
    /* initialize the µC */
    init_uC();

    /* infinity-loop -> super-loop */
    while (1)
    {
        /* time-slot 1 -> 1 ms */
        if (time1ms)
        {
            time1ms = 0;

            /* task 1 */
            task1();
            /* task 2 */
            task2();
        }
        /* time-slot 2 -> 10 ms */
        if (time10ms)
        {
            time10ms = 0;

            /* task 3 */
            task3();
            /* task 4 */
            task4();
        }
    }
}
```

Fig. 1: main-loop

Michael Zauner and Roman Froschauer are with the University of Applied Sciences Upper Austria, Stelzhamerstr. 23, 4600 Wels, Austria (phone: +43 (0)5/0804-43520; e-mail: michael.zauner@fh-wels.at, roman.froschauer@fh-wels.at

```
/* timer-ISR is executed every 1 ms */
ISR(TIMER1_OVF)
{
    /* time-slot 1 -> 1 ms */
    time1ms = 1;

    /* time-slot 2 -> 10 ms */
    if (++time10ms_T >= 10)
    {
        /* reset 10 ms - Timer */
        time10ms_T_T = 0;
        /* set time-slot 2 */
        time10ms = 1;
    }
}
```

Fig. 2: timer interrupt service routine

B. *Cooperative Operating System*

In cooperative multitasking, the current task is executed until it voluntarily hands over control to the kernel. There is no fixed time slice or timeout. The task scheduler then decides which task is to be executed next. This method is preferred in safety-critical applications as it allows strict control over CPU time and the tasks themselves are responsible for transferring control. This is illustrated in Fig. 3.

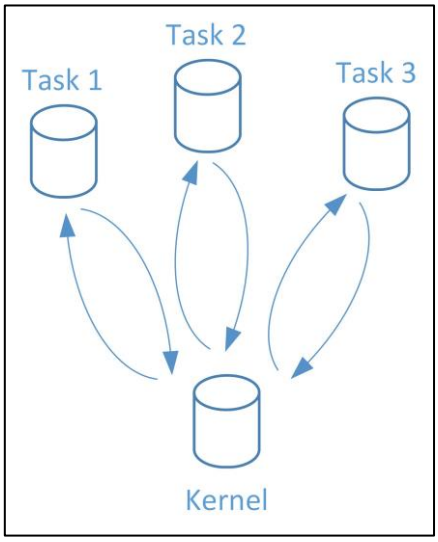

Fig. 3: task-scheduling process

This procedure leads to the following problems:

- If a task no longer frees up computing time, all other tasks are blocked
- The tasks are processed one after the other and important tasks can be delayed as a result

C. *Preemptive Operating System*

In preemptive multitasking, a task runs until the point at which either its assigned time slice expires, it is blocked, or it explicitly relinquishes control. The task planner then takes over the task of determining the next task to be executed, taking into account the priorities of the tasks. This type of multitasking ensures that tasks are automatically interrupted and continued according to their priority and time allocation, which is an advantage in more complex applications.

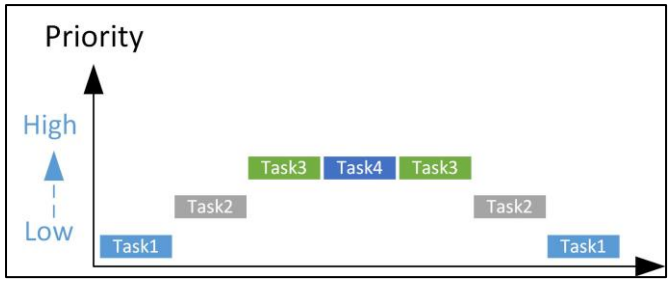

Fig. 4: preemptive scheduling with priority

Fig. 4 shows the process of preemptive multitasking. Tasks 3 and 4 have the highest priority and can therefore interrupt tasks 1 and 2. Round-robin scheduling is also shown, in which tasks of the same priority are executed alternately for a certain period of time. This ensures that all tasks with the same priority are executed equally [9].

The preemptive multitasking system naturally requires more effort to process the tasks. As each task can be interrupted at any time, it must be ensured that all internal states of a task can be saved at any time and restored at a later point in time. This requires a separate stack for each task. Furthermore, saving and restoring requires additional CPU resources. A further problem arises when several tasks access one and the same hardware resource. In this case, it must be ensured that this resource is blocked until the first task no longer requires it. Either mutex or semaphores are used for this purpose. In the cases just mentioned, however, so-called deadlocks can also occur.

III. µ-RoMS-Operating System

µ-RoMS-OS was developed for the use on low-level microcontrollers that are used in robot applications. However, the operating system can be used for any conceivable application. The main focus was placed on the economical use of computer resources such as memory requirements and computing time. To achieve this, the implementation was realized as a cooperative multitasking system.

Tasks can be created that are only executed once, but it is also possible to generate cyclical tasks that are called with a fixed or variable cycle time.

The program was written in C and consists of three files:

- rtos.c
- rtos.h
- rtos_config.h

These files can be added to any project on any platform.

The c-file contains the entire source code. This can be used out of the box. The only setting that needs to be made is the linking of the RTOS-timer function with a timer physically present on the microcontroller or with its interrupt service routine.

Specific definitions and the main structure of a task can be found in rtos.h. In rtos_config.h, the user can set the maximum number of tasks – OS_MAX_TASK – that can be used and the identifiers of the respective tasks. This identifier must be unique for each task and is a number between 0 and 255.

In order to prioritize time-critical tasks, three priority levels have been implemented. Tasks with a high priority can be set to OS_HIGH_PRIORITY. Tasks with a lower priority can be set to OS_MID_PRIORITY and tasks with the lowest priority can be set to OS_LOW_PRIORITY.

A. *µ-RoMS-OS Kernel*

The µ-RoMS-OS kernel or schedular is structured in three stages. First, all tasks with a high priority are processed. If a task with a high priority is found that also has the status OS_ENABLE, the function associated with the task is called. All tasks with a high priority are therefore processed in the first step.

After all tasks have been examined according to this pattern, the system continues with the tasks with medium priority. All tasks are checked again to see whether they belong to the group of tasks with medium priority. If a task is found that also has the status OS_ENABLE, it is executed and the search for the next task with the medium priority is aborted. If not, all tasks have been searched and a task has been found in the middle of the list, further processing is interrupted and the cycle starts again from the beginning with the high-priority tasks. If all tasks have already been searched for medium priority, the search continues with the tasks with the lower priority.

The processing of the high-priority tasks is therefore only interrupted by a task with a medium or low priority. This ensures that the high-priority tasks have more computing time available as shown in Fig. 5.

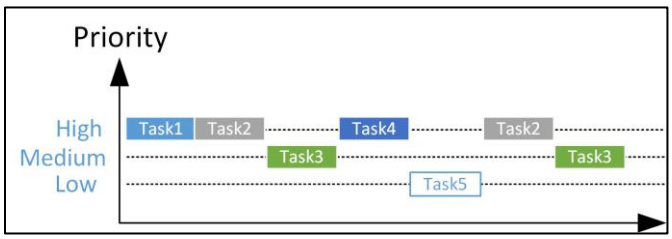

Fig. 5: µ-RoMS-OS schedular

B. *µ-RoMS-OS internal structure*

The µ-RoMS-OS kernel runs in an endless loop in the main program. In order to generate precise timing, a hardware timer of the microcontroller is required. This timer provides the system tick. In most cases, a time of *1ms* is sufficient. As already described above, this timer must be linked to the RTOS timer function.

The heart of the multitasking system is the structure of a task. This structure consists of the following member variables:

- *Status (8 bit)*: The status variable indicates the internal state of the task. There are the following bits (Fig. 6):
  - *Active (A)*: This bit is set when the task is activated/created (OS_ACTIVE)
  - *Priority (P1/P0)*: These bits can be used to set the three priority levels
  - *Cycle (C)*: this bit indicates that the task is enabled cyclically (OS_CYCLE)
  - *Enable (E)*: If this bit is set, the task is called the next time (OS_ENABLE)
- *Delay (16 bit)*: Waiting time between two calls to the task. With a timer interval of *1ms*, waiting times of up to *65.535s* can be created.
- *Time (16 bit)*: This variable is incremented from 0. When the value of the *Delay* is reached, the *Enable* bit in the status register is set to 1.
- *ID (8 bit)*: Identifier of the task - it must be a unique number between 0 and 255.
- *Caller (8 bit)*: The process that enabled the task can be entered here. When the delay time expires, the value OS_CYCLE_CALL is entered here. Timeout conditions can be generated and recognized using the caller variable. If, for example, a task is waiting for an event, a timeout can be generated using the delay variable. When the task is called, the caller can be used to recognize which event activated the task (event or timeout)
- **fct (32 bit)*: Pointer to the function associated with the task.

A	-	-	-	P1	P0	C	E

Fig. 6: status register

The structure for a task requires 11 bytes of RAM.

C. *µ-RoMS-OS system functions*

The µ-RoMS-OS provides different functions to the user applications to manipulate the tasks. There are functions for processes which allow to:

- Create a task
- Delete a task
- Set several bits in the status register
- Set the delay time
- Set the priority of the task
- Get information according to the status register, the delay time or the address of the task-structure

The timer function, an initialize function and a default task are also provided. In the initialize function the function pointers of each task is pointed to the default task. The functions will be described here in more detail.

void osInitRTOS();

The initialize function resets all values of any task. So, the status register, the delay time and the identifier are set to zero. The pointer of the task function will be set to the default task.

void osDefaultTask();

Is the default function, that is be called if no other function is set. It is an empty function, that does nothing.

void osTimerRTOS();

The timer checks cyclically (e.g. every *1ms*) which task has set the cycle bit in the status register. The waiting time for the corresponding tasks is increased by one. When the waiting time has expired, the task is enabled, i.e. the enable bit is set in the status register. This process is shown in Fig. 7.

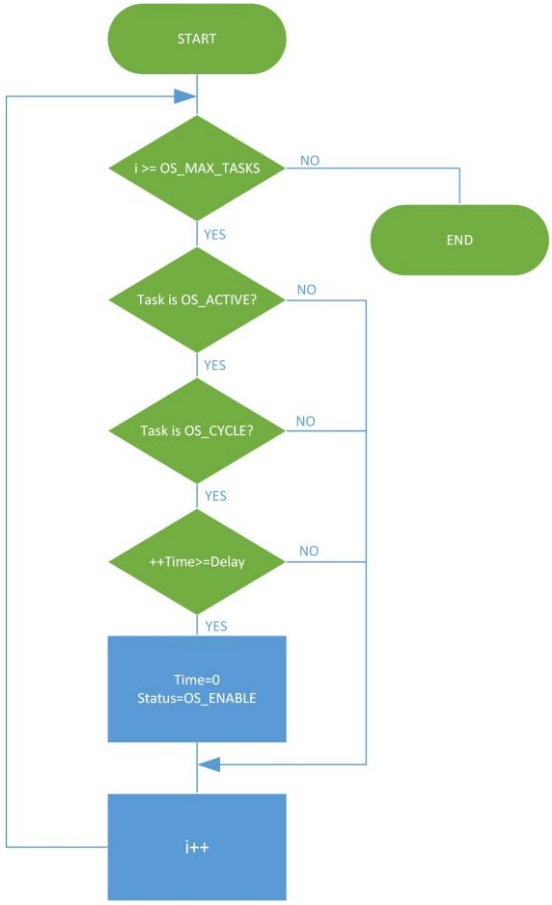

Fig. 7: osTimerRTOS

void osKernelRTOS();

The function of the kernel has already been explained in detail in section A.

*uint8_t osCreateTask(uint8_t status, uint16_t delay, uint8_t id, uint8_t prio, void(*fpt)());*

This function is used to initiate a task and make all the necessary settings. First, a free task is searched and blocked. The first parameter describes the behavior of the task in more detail. The macro OS_ENABLE can be used to enable the task and it is executed immediately. If the value OS_CYCLE is transferred in the first position, it is a cyclical task that is activated cyclically with the interval time delay (second transfer parameter). If the value OS_DISABLE is transferred, the task is created but not yet executed. The second transfer parameter is the cycle time for cyclical tasks. The third parameter is used to assign a unique identifier to the task. This identifier can be used to find the task in the task list. The fourth parameter specifies the priority of the task, which can be OS_LOW_PRIORITY, OS_MID_PRIORITY or OS_HIGH_PRIORITY. The scheduling process has already been explained above. The last parameter is the pointer to the function that is assigned to the task, i.e. this function is called when the task is enabled.

The function returns the position of the task in the task list if a free item was found. Otherwise, it returns the value OS_TASK_CREATION_ERROR. If the task has been successfully created, the active bit is set in the status register of the task. This signals that the task is already in use.

uint8_t osGetTaskIndex(uint8_t id);

This function uses the identifier to search for the position/index of the task in the task list. This index can be used to access the task and, for example, change values or delete the task.

void osDeleteTask(uint8_t idx);

This function can be used to delete a task. All entries are reset to zero at *idx* in the task list and the pointer to the function assigned to the task is reset to *osDefaultTask*.

void osSetStatus(uint8_t idx, uint8_t status);

This function can be used to change the status of a task. A task that has been initialized as OS_DISABLE, for example, can be activated by writing the value OS_ENABLE.

void osSetPriority(uint8_t idx, uint8_t prio);

This function can be used to change the priority of the task. For example, the task at the *idx* position in the task list with a low priority can be set to a higher priority.

D. Why implementing a cooperative multitasking system?

As shown at the beginning, a cooperative multitasking system has some disadvantages compared to a preemptive multitasking system. However, there are also advantages that predominate in this case. The first advantage is the low memory requirement of a task. As shown, each task only requires 11 bytes of RAM. Compared to a preemptive system, this is very efficient. The standard setting for the stack of a single task in *FreeRTOS* is often in the kB range. This would be impossible to implement for a microcontroller with only a few hundred bytes to a few kilo bytes. Furthermore, preemptive systems often have a memory requirement of several 10 kB in the program memory [4] and [5]. In contrast, the system presented here requires less than 2 kB. In a test application with the microcontroller *AVR128DB48* from Microchip, the IDE *MPLABX V6.20* delivered a code with 1848 bytes of program memory.

Table 1 compares various embedded OSs with the presented µRoMS- OS in terms of their memory requirements. A number of 5 tasks was used for the operating systems marked with an *.

Name	*Schedular*	*Flash (kB)*	*RAM (kB)*
Contiki	Cooperative	30	10
RIOT-OS	Preemptive	5	1.5
Mbed	Preemptive	16	4
TinyOS	Cooperative	4	1
LiteOS	Preemptive	26	6
FreeRTOS*	Preemptive	10	2*
µRoMS-OS*	Cooperative	1.8	0.05*

Tab. 1: memory footprint for embedded OS [4] [5]

Knowing that a cooperative system is being used, the tasks can be programmed in an optimized way. State machines are an important tool for creating such optimized programs. This allows programs to be divided into subtasks that can be interrupted again and again. This frees up computing time for other tasks and each task can be executed more efficiently.

IV. Robot Control with µ-RoMS-OS

One application is a robot that is built for uneven terrain and is equipped with an omnidirectional power engine and vision system (Fig. 1) [10]. This robot is built to get a fast overview in search and rescues missions. This is also necessary to plan and coordinate the action force like fire fighters or first-aid suppliers.

Fig. 8: rescue robot with omnidirectional power engine

The controlling system is separated in two levels, the high-level control and the low-level control. The high-level control is based on an industrial computer. This computer is set up with Linux and ROS. It deals with the cameras, RGB and thermal. It is also capable to make a 2D and 3D-Mapping using a laser range finder and an Assus Xtion Pro. The computer is connected via WLAN with the operator station. It provides also the motion commandos for the low-level control system. These commandos can be received from the operator station or can be generated by the computer itself in autonomous mode. The low-level control is responsible to control the driving system motors or the actuator motors. It deals with different sensors like CO2-sensors, ultrasonic-sensors or the IMU. It also supports interfaces for add on devices (Fig. 9).

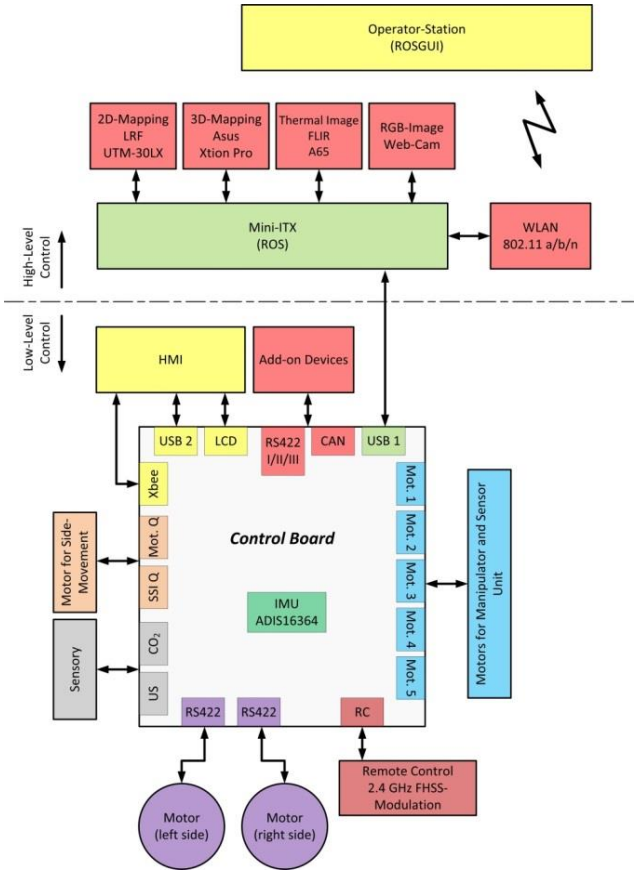

Fig. 9 overview control system

The software of the low-level control is divided into different layers. Thereby the lower layers provide the low-level data processing and the higher layers deal with the high-level data processing. All layers work with µ-RoMS-OS, which was developed for different robot projects. This multitasking system needs less space in the memory of the microcontroller and is also able to deal with very tiny controllers.

Hardware Abstraction Layer (HAL): The HAL provides methods to access direct to the hardware. These methods are e.g. subroutines to read out the ADC, deal with the PWM-controller or to initialize the whole microcontroller. Thus, it is possible to port the code to another microcontroller. The task to do is only to switch the HAL.

Control- and Evaluation Layer (CEL): The CEL provides drivers for further processing the sensor data or to control the actuators. In this layer is the protocol stack, the sensor reprocessing or the actuator control system. E.g. the sensor reprocessing method read out, with the HAL-driver, the corresponding ADC-value and filters it. After the filtering the value is recalculated to the corresponding SI-unit. This value is provided to the next higher level.

Instruction Layer (IL): The IL receives commands from the PC and execute them. Such commands are to level the LRF or stabilize the robot in a horizontal position. An additional task is the human machine interaction.

V. CONCLUSION

This paper presents a multitasking system for low-level microcontrollers. The program was originally developed to simplify the process of controlling mobile autonomous robots. In this area in particular, a wide variety of tasks must be carried out in parallel and almost simultaneously. Attention was also paid to the application of the software on different microcontrollers with different memory and computing time resources. Another important point was to make the implementation in a project as simple as possible. It should also be able to be used by less experienced programmers without a long training phase. The program is now used in a wide variety of projects, from robot applications and motor controllers to data acquisition, on different microcontrollers, such as Xmega256A3 or AVR128DB48, but also on 32-bit processors such as ATSAME70Q21.

REFERENCES

[1] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., … Ng, A. (2009). ROS: an opensource Robot Operating System. In ICRA workshop on open-source software (Bd. 3, S. 5). Kobe.

[2] Belsare, K., Rodriguez, A. C., Sánchez, P. G., Hierro, J., Kołcon, T., Lange, R., ... & von Mendel, J. (2023). Micro-ros. In Robot Operating System (ROS) The Complete Reference (Volume 7) (pp. 3-55). Cham: Springer International Publishing.

[3] Barry, R. (2008). FreeRTOS. Internet, Oct. 2008

[4] Hee, Y. H., Ishak, M. K., Asaari, M. S. M., & Seman, M. T. A. (2021). Embedded operating system and industrial applications: a review. Bulletin of Electrical Engineering and Informatics, 10(3), 1687-1700.

[5] Cekerevac, Z., Dvorak, Z., & Pecnik, T. (2020). Top seven IoT operating systems in mid-2020. MEST Journal, 8(2), 47-68.

[6] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., ... & Culler, D. (2005). TinyOS: An operating system for sensor networks. Ambient intelligence, 115-148.

[7] Nahas, Mouaaz. Implementation of highly-predictable time-triggered cooperative scheduler using simple super loop architecture. *International Journal of Electrical & Computer Sciences*, 2011, 11. Jg., Nr. 4, S. 33-38.

[8] Ibrahim, Dogan. "µRTOS: Simple multitasking with microcontrollers." *Near East University in Cyprus* (2010).

[9] Silberschatz, Abraham, and Peter Baer Galvin. "Operatings System Concepts." (2013).

[10] Edlinger, Raimund, Michael Zauner, and Walter Rokitansky. "INTELLIGENT MOBILITY-New approach of robot mobility systems for rescue scenarios." *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*. IEEE, 2013.