RT*uin*OS
– A Real Time Operating System for Arduino –
Version 1.0
User Guide

Peter Vranken

July 2013

Contents

1 Introduction **7**

2 How does RT*uin*OS work? **9**
2.1 Implementation of RT*uin*OS' Scheduler . . . 10
2.2 Time based Events . . . 11
2.3 Explicitly posted Events . . . 12
2.3.1 Events of Kind Mutex . . . 12
2.3.2 Events of Kind Semaphore . . . 13
2.4 Application Interrupts . . . 13
2.5 Return from a Suspend Command . . . 14
2.6 Summarizing the Scheduler Actions . . . 14
2.7 Task Switches . . . 16
2.7.1 Interrupt versus API Function . . . 18

3 The API of RT*uin*OS **21**
3.1 Configuration of RT*uin*OS: rtos.config.template.h . . . 21
3.2 Initialization of RT*uin*OS: *setup* . . . 21
3.3 Specification of Tasks: *rtos_initializeTask* . . . 21
3.4 Initialization of Application Interrupts: *rtos_enableIRQUser<nn>* . . . 22
3.5 The idle Task: *loop* . . . 23
3.6 Suspend a Task: *rtos_waitForEvent* . . . 23
3.6.1 Events of Kind Mutex . . . 24
3.6.2 Events of Kind Semaphore . . . 24
3.6.3 Notes on waiting for Events of Kind Mutex or Semaphore . . . 25
3.6.4 *rtos_suspendTaskTillTime* . . . 25
3.6.5 *rtos_delay* . . . 25
3.7 Awake suspended Tasks: *rtos_sendEvent* . . . 26
3.8 Data access: *rtos_enter/leaveCriticalSection* . . . 26
3.8.1 Mutex versus Critical Section . . . 27
3.9 Diagnosis: *rtos_getTaskOverrunCounter* . . . 28
3.10 Diagnosis: *rtos_getStackReserve* . . . 28
3.11 Diagnosis: *gsl_getSystemLoad* . . . 29

4 Writing an RT*uin*OS Application **30**
4.1 Short Recipe . . . 30
4.2 The Makefile . . . 31
4.2.1 Prerequisites . . . 32
4.2.2 Concept of Makefile . . . 32
Folder Structure . . . 33
4.2.3 Compilation Configurations . . . 33

4.2.4 Selecting the Arduino Board . . . 34
4.2.5 Selecting the USB Port . . . 34
4.2.6 Weaknesses of the Makefile . . . 34
Unsafe Recognition of Dependencies . . . 34
g++ versus gcc . . . 34
4.3 Configuring the System Time . . . 35
4.3.1 The Unit of the System Time . . . 35
4.3.2 Recommended Timer Tic for regular Tasks . . . 35
4.3.3 Upper Limit of the Task Period Time . . . 35
4.3.4 Unrecognized Task Overruns . . . 36
4.3.5 Summary . . . 36
4.4 Configuring the System Timer Interrupt . . . 37
4.5 Using Application Interrupts . . . 38
4.6 Usage of Arduino Libraries . . . 39
4.6.1 Changes of Arduino's *main* Function . . . 40
4.7 Support of different Arduino Boards . . . 40

5 Outlook **42**

GNU Free Documentation License **44**
1. APPLICABILITY AND DEFINITIONS . . . 44
2. VERBATIM COPYING . . . 45
3. COPYING IN QUANTITY . . . 45
4. MODIFICATIONS . . . 46
5. COMBINING DOCUMENTS . . . 47
6. COLLECTIONS OF DOCUMENTS . . . 47
7. AGGREGATION WITH INDEPENDENT WORKS . . . 48
8. TRANSLATION . . . 48
9. TERMINATION . . . 48
10. FUTURE REVISIONS OF THIS LICENSE . . . 49
11. RELICENSING . . . 49

References

	Document	Explanation
[1]	code/RTOS/rtos.c	C source code file of RT*uin*OS
[2]	code/RTOS/rtos.h	Header file of RT*uin*OS, declaring the API
[3]	code/RTOS/rtos.config.template.h	Compile time configuration of RT*uin*OS (template file)
[4]	rtos.config.h	Application owned compile time configuration of RT*uin*OS, derived from [3]
[5]	code/RTOS/rtos_assert.h	Macro definition supporting code self-diagnosis
[6]	doc2549.pdf, e.g. at www.atmel.-com/Images/doc2549.pdf	User manual of CPU ATmega2560 and others
[7]	code/RTOS/gsl_systemLoad.c	System load estimation
[8]	code/RTOS/gsl_systemLoad.h	Definition of global interface of system load estimation

Table 1: References

Abbreviations

Abbreviation	Explanation
API	Application Programming Interface
APPSW	Application Software
AVR	Name of a popular micro controller family by Atmel
CPU	Central Processing Unit
EEPROM	Electrically erasable programmable read only Memory
I/O	Input/Output
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
NVM	None Volatile Memory
RAM	Random Access Memory
ROM	Read only Memory
RTOS	Real time operating system
SW	Software
SWI	Software Interrupt
TBC	To be Checked
TBD	To be Defined

Table 2: Abbreviations

Chapter 1

Introduction

Arduino[1] is a popular open source and open hardware micro controller platform for various purposes, mainly located in leisure time projects. Arduino comes along with a simple to use integrated development environment, which contains the complete tool chain to write source code, to browse through samples and libraries, to compile and link the software and to upload it to the board and flash it. The RT*uin*OS project adds the programming paradigm of a real time operating system to the Arduino world.

Real time operating systems, or RTOS, strongly simplify the implementation of technical applications which typically do things in a quite regular way, like checking inputs and setting outputs accordingly every (fixed) fraction of a second. For example, the temperature controller for a heating installation could be designed this way. Temperature sensors, which report the room temperatures are evaluated and the burner and maybe some valves are controlled to yield the desired target temperature. Furthermore, using a real time system the program could coincidentally and regularly update a display to provide feedback – at the same or any other rate. Regular, time based programming can be done without the need of CPU consuming waiting loops as used in the implementation of Arduino's library functions *delay* and *delayMicroseconds*. Real time operating systems characterize the professional use of micro controllers.

RT*uin*OS is a small real time operating system (RTOS) for the Arduino environment. It is simple to use and fits well into the existing Arduino code environment. It adds the concept of pseudo-parallel execution threads to the sketches.

The traditional Arduino sketch has two entry points; the function *setup*, which is the place to put the initialization code required to run the sketch and function *loop*, which is periodically called. The frequency of looping is not deterministic but depends on the execution time of the code inside the loop.

Using RT*uin*OS, the two mentioned functions continue to exist and continue to have the same meaning. However, as part of the code initialization in *setup* you may define a number of tasks having individual properties. The most relevant property of a task is a C code function[2], which becomes the so called task function. Once entering the traditional Arduino loop, all of these task functions are executed in parallel to one another and to the repeated execution of function *loop*. We say, *loop* becomes the idle task of the RTOS.

A characteristic of RT*uin*OS is that the behavior of a task is not fully predetermined at compile time. RT*uin*OS supports regular, time-controlled tasks as well as purely event controlled ones, where events can be broadcasted or behave as mutex or semaphore. Tasks can be preemptive or interact cooperatively. Task scheduling can be done using time slices and a round robin pattern. Moreover, many of these modes can be mixed. A task is not per se regular, its implementing code decides what happens and this can be decided context or situation dependent. This flexibility is achieved by the basic idea of having an event controlled scheduler, where typical RTOS use cases are supported by providing according events,

[1] See www.arduino.cc

[2] The GNU C compiler is quite uncomplicated in mixing C and C++ files. Although RT*uin*OS is written in C it's no matter do implement task functions in C++ if only the general rules of combining C and C++ and the considerations about using library functions (particularly *new*) in a multi-tasking environment are obeyed. Non-static class member functions are obviously no candidates for a task function.

e.g. absolute-point-in-time-reached. If the task's code decides to always wait for the same absolute-point-in-time-reached event, then it becomes a regular task. However, situation dependent the same task could decide to wait for an application sent event – and give up its regular behavior. In many RTOS implementations the basic characteristic of a task is determined at compile time, in RT*uin*OS this is done partly at compile time and partly at runtime.

RT*uin*OS is provided as a single source code file which is compiled together with your other code, which now becomes an RT*uin*OS application. In the most simple case, if you do not define any task, your application will strongly resemble a traditional sketch: You implement your *setup* and your *loop* function; the former will be run once at the beginning and the latter repeatedly.

RT*uin*OS on its own can't be compiled, there need to be an application. RT*uin*OS is organized as a package which combines the RT*uin*OS source file with some sample applications which are the test cases at the same time. The source code of each sample application is held in a separate folder, named tc<nn>. Any of these can be selected for compilation. You may add more folders, holding the source code of your RT*uin*OS applications. A starting point of your application folder can be a copy of any of the folders tc<nn>. The compilation always is the same. Run the makefile, where the name of the folder (which doesn't need to be tc<nn>) is an option on the command line. See section 4.2 for more.

This document introduces the basic concept of RT*uin*OS and gives an overview of its features and limitations:

Chapter 2 explains the basic principles of operation. Some core considerations of the implementation are highlighted, but the relevant documentation of the implementation is the code itself. It is commented using doxygen[3] tags; the compiled doxygen documentation is part of this project. It contains only the documentation of the global objects of RT*uin*OS; to fully understand the implementation you will have to inspect the source code itself, please refer to [1], [2], [3].

Chapter 3 lists and documents all elements of RT*uin*OS' API.

Chapter 4 explains how to write a well-working RT*uin*OS-application. The chapter starts with a short recipe, which guarantees soon success. Here is where you may start reading if you are already familiar with the concept of an RTOS.

The manual closes with chapter 5, which gives an overview of possible improvements and still missing and maybe later released features.

[3]See http://www.doxygen.org

Chapter 2

How does RT*uin*OS work?

In a traditional sketch, the function *loop* defines all actions which have to be executed. The execution of the code is strictly sequential. *loop* may of course call any sub-routines, which may call others, but this does not change the strictly sequential character of the execution of the statements. Therefore it is difficult to execute specific actions at specific points in time. For example toggle the state of an LED every seconds. This might still be easy to do if your sketch does nothing else – see Arduino's standard example *blink* –, but if you're in the middle of a sketch, which e.g. transfers some data via the USB port, it becomes ugly: You will have to merge some specific, the LED serving statements, into your USB handling code. And the accuracy of the yielded timing will not be perfect.

Imagine, you could simply write two sketches. The USB communication stays as it is but a second sketch, e.g. the sample sketch *blink*, is defined at the same time and will be executed, too. Now the USB code is no longer spoiled with double-checking the state of the LED but nonetheless the LED will blink as desired.

This actually is what RT*uin*OS offers. It's however not a complete sketch but just a function – which can of course be stored in a separate C source file –, which is executed in parallel. Write two such functions, make them so called tasks and you get what you want.

How would this work? The Arduino board continues to have a single CPU, which is available for code execution. The trick is to use it alternatingly to proceed with the one task and then with the other one. If this switching between the tasks happens fast enough, than it's just the same as if both would run at the same time – only with limited execution speed.

Does alternating between the tasks happen regularly? It depends. Different patterns of alternating between tasks are possible. The most simple pattern is to share the CPU in fixed portions between the tasks. The ratio can be chosen. If we assume in our example that serving the USB port is more challenging than flashing an LED, it would be reasonable to share the CPU by 95:5 rather than by 50:50. This pattern is called round robin and fits well if the tasks are completely independent of each other (as in our example) and if all of them continuously require the CPU – which is not the case in our example!

Toggling the LED state can be done by permanently observing a watch and switching the LED output when it reaches the next mark. A traditional Arduino implementation of this strategy requires the CPU indeed permanently and this is exactly how the sample sketch *blink* is implemented. In an RTOS you can do it better. Tell the system when you want to toggle the LED state the next time and do nothing until. Your task is inactive, does not require the CPU any more and is nonetheless executed again exactly at the desired point in time. Two advantages arise. Your task does barely consume CPU power and the regularity of the execution is very good. This pattern is the appropriate solution for our example. The LED is blinking very regularly, and nearly all of the CPU performance can still be spend on the USB task, which therefore behaves the same way as if there was no blinking LED at all.

The next pattern of sharing the CPU between different tasks is direct coupling of tasks. By means of so called events a task can indicate a specific situation to any other task, which will then react on this situation. For example, the LED should not constantly blink. It is now used to indicate the state of the

USB communication by flashing a number of times if a significant state change occurs. The number of flashes will notify what happened. The LED task will now use an event to become active and subsequently it'll use the execute-at-time pattern a number of times to realize the sequence of flashes. The LED stays dark as long as the event is not posted to the task. After the blink sequence the task will again start to wait for the next occurrence of the event.[1]. The active time of this task is still close to zero: Every time it is executed it'll just use a few statements to toggle the LED state and to inform the RTOS about the next condition under which to become active again. The main task, which implements the communication can proceed nearly as if there was no LED task. There's however an extension to its implementation. In case of a significant status change it has to indicate the number of according flashes and to post the event. The former can be done by a write to a global variable and the latter is a simple call of an RT*uin*OS API function. The global variable is shared by both tasks, the LED task will read it when being activated by the event.

If there are more tasks the scenario becomes more complicated and we need a new term. If two tasks tell the RTOS the time they want to be activated again, there is a certain chance, that it is the same time. In this situation, if this point in time is reached, RT*uin*OS decides that both tasks are due – but only one of them can get the CPU, i.e. can be activated. Which one is decided by priority. The priority of a task is a static, predetermined property of a task. At compilation time, you will decide which of the tasks has the higher priority and which is the one to get the CPU in the mentioned situation.

Similar: The tasks do not specify the same point in time but nearly the same. Obviously, the task becoming due earlier will get the CPU. And when the other one becomes due a bit later it might still desire to have the CPU. Will it continue to have it? It depends again on the priority of the tasks. If the later due task has the higher priority, it'll take over the CPU from the earlier task. The earlier task is still due (as it didn't return the CPU voluntarily so far) but no longer active. The later task is both, due (it's point in time has reached) and active (it got the CPU).

When the active, later task tells RT*uin*OS to no longer need the CPU, the earlier still due task will again get the CPU and continue to execute.

If a task tells the system to momentarily no longer require the CPU (by notifying: "Continue my execution at/when") we say it is suspended. If it becomes due again, we say it is resumed. If it is executed again, we say it is activated.

There's always one and only one active task. What if all tasks in the system did suspend themselves because they are waiting for a point in time or an event? Now, there is a single task which must never suspend itself. This task is called the idle task and RT*uin*OS can't be compiled without such a task being present.[2] The idle task doesn't need to be defined in the code. RT*uin*OS uses the function *loop* as idle task. All the code in function *loop* makes up task idle. If you don't need an idle task (as all your code is time or event controlled) just implement *loop* as an empty function body.

Not implementing the idle task is however a waste of CPU time. RT*uin*OS spends all time in *loop* when none of the other tasks is due. So if there are any operations in your application which can or should be done occasionally it's good practice to put them into the idle task. There's no drawback, if any task needs the CPU, idle is just waiting until the task has finished. The execution speed of task idle is directly determined by the CPU consumption of the other tasks. Idle will never slow down the tasks, but the tasks will slow down idle. Idle is a task of priority lower than all other priorities in the system. Idle is a task which is never suspended but always due and sometimes active.

2.1 Implementation of RT*uin*OS' Scheduler

The set of rules how to share the CPU between the different tasks is called the scheduler. Actually, RT-*uin*OS is nothing else than the implementation of a scheduler.[3] Understanding the details of the decision

[1] An implementation of this pattern can be found in test application *tc08*

[2] Caution, if the idle task would ever try to suspend itself, a crash of the system would result.

[3] From the world of personal computers you will associate much more with the term operating system than just a task scheduler. In fact, in this environment, the scheduler is just the most important part of the operating system, therefore referred to as kernel, but surrounded by tons of utilities, mostly to support various I/O operations. In the embedded world

rules implemented in RT*uin*OS is essential for writing applications that behave as desired. (An RTOS can easily show effects which are neither expected nor desired.) The rules of RT*uin*OS' scheduler will become clear in the following documentation of its implementation.

In RT*uin*OS, a task is represented by a task object. All task objects are statically allocated, there's no dynamic creation or deletion.[4] The objects are configured in Arduino's function *setup* and stay unchanged from now (besides the continuous update of the runtime information by the scheduler, see below).

RT*uin*OS manages all tasks in lists. Please refer to figure 2.1 on page 15. There's one list per priority. All tasks having a specific priority form a priority class and this class is managed with the associated list. The number of different priorities is determined at compile time by the application.

An additional list holds all tasks (of any priority) which are currently suspended.

Within a task object there are a few members which express a condition under which a suspended task is resumed. (These members are empty or in a don't care state when a task is due or active.) More concrete, RT*uin*OS knows about a limited number of distinct events[5] and the mentioned condition is a sub-set of these plus the following Boolean choice: will the task be resumed as soon as any event of the sub-set is seen or does the task stay suspended until all events have been seen? Moreover, three of the known events have a specific meaning; they are all timing related and have a parameter of kind *when*.

At system initialization time all tasks are put in state suspended. Consequently, the initial resume condition is part of the task initialization. Typically, this condition is weak, like "resume immediately". However, a delayed resume is possible. Starting some regular tasks with differing delays may be advantageous in order to avoid having too many tasks becoming due all at the same time. Furthermore, a task may specify the initial resume by a broadcasted application event. And if a task implements the handler of an application interrupt it will probably be initially resumed by this interrupt.

Please note, that events of kind mutex or semaphore must not be used as part of a task's initial resume condition. In many cases such a condition simply doesn't define a suspended state; if the mutex or semaphore (or better to say a counter value of it) is currently available the requesting task would not be suspended – which is a must at the time of initializing the task objects as the RT*uin*OS kernel is not yet running at this point in time.

2.2 Time based Events

The most relevant events of RT*uin*OS are absolute and relative time events. An absolute time event means "resume task at given time". A relative time event means "resume task after a given time span counted from now". You see, the latter – probably with time span zero – is the most typical initial resume condition for a task. It'll start immediately.

In the implementation a time event results from a system timer tic. The core of the scheduler is a clock based interrupt. In RT*uin*OS this clock has a default clock rate of about 2 ms, but this can be altered by the application code.[6] In the interrupt service routine (ISR), a variable holding the absolute, current system time is incremented. (The rate of the interrupt is the unit of all timing operations regardless of the actual physical value.) The new value of the system time is compared against the *when* parameter of all suspended tasks which suspended with a resume-at condition. If there's equality, the absolute time event is notified to this task. For all tasks which had suspended themselves with a resume-after condition, the *when* parameter of this condition is decremented. If it reaches null, the desired suspend time has elapsed and the relative time event is notified to the task.

The third time based event is available only if the system is compiled with round robin feature. Now a task may have a time slice defined. The time slice is the maximum time the task may continuously be

a real time operating system typically doesn't offer much more than a scheduler and so does RT*uin*OS.

[4] Please, refer to section 5 for more detailed considerations about this.

[5] For good reasons the events are implemented by bits in an unsigned integer word. Currently, type *uint16_t* is used as it is a good trade off between number of events and performance. A change to *uint8_t* or *uint32_t* is possible but not trivial as assembler code is affected. We thus have 16 such events.

[6] RT*uin*OS uses timer 2 as source for clocking its system time. In Arduino this timer – intended for PWM output – cycles with about 2 ms. RT*uin*OSdoesn't change Arduino's timer configuration in its standard configuration.

active. This event is implemented directly, not as a bit, not as notification to the task. A round robin task has a counter which is loaded with the time slice duration at activation time. In each system timer tic it is decremented for the one and only active task (if it only is a round robin task). When the counter reaches null it is reloaded and the task is immediately taken from the head of its due list and put at the end of this list. This means the task stays due but will become inactive. Another task, the new head of the list, is a more promising candidate for the new, active task.

The next step is to check the conditions of all suspended tasks. For each such task it is checked if its resume condition is fulfilled, i.e. if all events it is waiting for have been posted to it meanwhile. If so, it is taken out of the list of suspended tasks and placed at the end of the due list of the priority class the task belongs to.

Now, after reordering the tasks in the several lists, the ISR finishes with looking for the new, active task. The decision is easy. It loops over all due lists, beginning with the highest priority. The head of the first found non-empty list is the new, active task. If all due lists are empty, the idle task is chosen. The selected task is made active and the ISR ends.

2.3 Explicitly posted Events

Besides the system timer tic, the scheduler becomes active in two other situations. The first one is the event explicitly posted by an application task. RT*uin*OS knows a predefined number of general purpose events, which can be posted by one task and which another task can wait for. The latter task suspends itself and specifies the event as resume condition. Under application defined conditions, the former task calls the RT*uin*OS API function *rtos_sendEvent* and the latter task will resume.

In this situation it depends on the priorities of the two tasks how *rtos_sendEvent* returns. If the event-setting task has the same or higher priority *rtos_sendEvent* will immediately return like an ordinary sub-routine. The other task becomes due but not active. If the event-receiving task has the higher priority, *rtos_sendEvent* leads to temporary inactivation of the calling task and will return only when it is activated again.

More in general, *rtos_sendEvent* is implemented as a software interrupt (SWI) and behaves similar to the system timer ISR. It notifies an ordinary, broadcasted event to all currently suspended tasks, which are waiting for it and passes events of kind mutex or semaphore to the very task of highest priority, which demanded to acquire it. The rest is done exactly as the system timer ISR does. *rtos_sendEvent* checks the resume condition of all suspended tasks. Those tasks the condition of which is fulfilled are moved to the end of their due list. Then the new active task is selected. This might be the same or another task. The SWI ends with continuing the new, active task.

As a matter of fact, a call of *rtos_sendEvent* will never make the calling task undue (i.e. suspended), outermost inactive. This is the reason, why *rtos_sendEvent* may even be called by the idle task.

Side note: There is a crosswise relationship between *rtos_sendEvent* and the suspend commands. From the bird's eye view on the system task code switching appears as follows: The suspend function is invoked but it returns out of a call of *rtos_sendEvent* of another task or out of the suspend command of a task which became due meanwhile. If a task calls *rtos_sendEvent* it doesn't need to return but could for example return out of the initially mentioned suspend command.

2.3.1 Events of Kind Mutex

An event can be a mutex, a synchronization object for mutual exclusion of tasks from a resource. The kind of an event is specified at compile time, it's part of the configuration of RT*uin*OS.

If a mutex event is in the set of explicitly posted events its handling differs from ordinary events. While these are notified to all currently suspended tasks a mutex is notified to outermost one task. If one or more suspended tasks are currently waiting for the mutex the one belonging to the highest priority class will get it. If there are several suspended tasks of this class the one will get it, which waits for it the longest. If there's no suspended task waiting for the mutex it is stored inside RT*uin*OS and can be

acquired later by any task. Saving a posted event for future use is different to ordinary events: They have no effect if nobody is currently waiting for them.

Under all conditions a posted mutex event is recorded somewhere and the calling task looses the ownership of the mutex. Either the ownership is immediately passed to a waiting task or the mutex is stored in the kernel for later acquisition. The application needs to keep track of this. Normally, owning a mutex means to a task to have access to a related, managed resource. In RT*uin*OS it can't be queried by an API call who currently is the owner of a mutex; actually RT*uin*OS isn't even aware of it. The task code needs to keep track about, e.g. by implementing a related Boolean variable.[7]

2.3.2 Events of Kind Semaphore

An event can also be a semaphore, a synchronization object for managing access to a pool of resources shared between different tasks. The kind of an event is specified at compile time, it's part of the configuration of RT*uin*OS.

If a semaphore event is in the set of explicitly posted events its handling differs from ordinary events. While these are notified to all currently suspended tasks a semaphore is notified to outermost one task. If one or more suspended tasks are currently waiting for the semaphore the one belonging to the highest priority class will get it. If there are several suspended tasks of this class the one will get it, which waits for it the longest. If there's no suspended task waiting for the semaphore the counter of the semaphore is incremented (the counter is implemented inside RT*uin*OS), indicating a released instance which is not required by any task at the time being but which can be acquired in the future. Saving a posted event for future use is different to ordinary events: They have no effect if nobody is currently waiting for them.

Under all conditions a posted semaphore event is recorded somewhere, it's either notified to a waiting task or it's counted in the kernel, and the application needs to keep track of this. Normally, the acquired number of instances of a semaphore (or its counter values respectively) corresponds with the number of instances from a managed pool of resources the task was granted access to. The number of instances a task has accumulated across a series of calls of *rtos_waitForEvent* and *rtos_sendEvent* can't be queried by an API call; actually RT*uin*OS isn't even aware of it. The application code needs to keep track of, e.g. by implementing a related counter variable.[8]

2.4 Application Interrupts

The last situation where the scheduler gets active is an application interrupt. By compile switch, you can bind any AVR interrupt to the *rtos_sendEvent* function. The ISR of the interrupt source will call *rtos_sendEvent*. The event which is posted is no longer a general purpose event but dedicated to this interrupt.

The actions are exactly the same as described for *rtos_sendEvent* in section 2.3. Obviously, the ISR is asynchronous to the task execution. If the posted event makes a task due which has a higher priority than the interrupted task, the interrupted task is made inactive (but it remains still due) and the other task will become active.

There's only one use case for this kind of scheduler invocation. Typically, an application will define a task of high priority waiting for the event. The interrupt triggers this dedicated task – indirectly via the hidden call of *rtos_sendEvent* –, which actually serves as interrupt handler, doing all sort of things which are needed by the interrupt. This tasks will be implemented as an infinite *while*-loop, where the

[7] Such a Boolean variable could be hidden in a C++ class embedding all calls of *rtos_waitForEvent* and *rtos_sendEvent* into member functions like *acquireMutex* and *releaseMutex*. Each task would have its own, local object of this class. However, this concept is simple only if waiting for combinations of events is excluded – which won't be a painful restriction for most applications.

[8] An appropriate way of doing could be a C++ class embedding all calls of *rtos_waitForEvent* and *rtos_sendEvent* into member functions like *acquireSemaphore* and *releaseSemaphore*. However, this concept is simple only if waiting for combinations of events is excluded – which is no painful restriction for most applications.

while-condition is the suspend command that waits for the interrupt event and where the body of the loop is the actual handler of the interrupt.

2.5 Return from a Suspend Command

An important detail of the implementation of RT*uin*OS is the way information is passed back from the scheduler to the application code. The direct interaction of the application code with the scheduler is done with the suspend commands and function *rtos_sendEvent* in the RT*uin*OS API.

rtos_sendEvent takes a parameter – the set of events to be posted to the suspended tasks – but doesn't return anything. Here, the only complexity to understand is, that it won't immediately return. It triggers a scheduler act and won't return until the calling task is the due task with highest priority. Which can be the case between immediately and never in case of starvation.

The suspend commands take some parameters which specify the condition under which the task will become due again, e.g. an absolute-time-event. While the task is suspended, the different scheduler acts repeatedly double check whether a sufficient set of events has been posted to the task (see above). Each posted event is stored in the task object. As soon as the posted set suffices, the task is moved from the suspended list to the end of the due list of given priority class. From now on, since the task is no longer suspended, no further events will be posted to this task and nor will they be stored in the task object. Thus, the very set of events, which made the task due is now frozen in the task object. When the task, which is due now, becomes the active task again the code flow of the task returns from the suspend command it had initially invoked. At this occasion, the stored, frozen set of events, which had made the task due is returned as return value of the suspend command.

By simply evaluating the return code a task can react dependent on which events it had made due. This is of particular interest if a task suspends waiting for a combination of events. In practice this will be most often the combination of an application event and a relative-time event, which this way gets the meaning of a timeout. Obviously, the task needs to behave differently whether it received the expected event or if a timeout occurred.

By the way, what has been said for the return from a suspend command also holds true for the initial entry into a task function. Any task is initialized in suspended state. The first time it is released the code flow enters the task function. What's otherwise the return code of a suspend command is now passed to the task function as function parameter. This way the task knows by which condition it has been initially activated.

2.6 Summarizing the Scheduler Actions

The different actions of the scheduler are depicted in figure 2.1. The solid arrows indicate how task objects are moved within and across the lists in different situations. The dashed arrows represent pointers to particular task objects.

Under all circumstances, the active task – highlighted in green – is the head of the top most non-empty list, i.e. the first due task in the order of falling priority. Because of the particular importance of this task the scheduler permanently holds a pointer to this task object. One could say, that maintaining this pointer actually is all the scheduler has to do.

The idle task object can be considered the only member of the due list of lowest possible priority. This task is needed as fall back if no due task is found in any of the priority classes and the scheduler has a constant pointer to this specific task object. If RT*uin*OS is idle both pointers have the same value; they point to the idle task object.

Arrow 1 depicts the round robin action. Round robin activities can only apply to the currently active task. If its time slice is elapsed, it is taken from the head of the due list and placed at the end of this list. Naturally – and indicated by arrow 2 – the next object in the list becomes the new head of the list

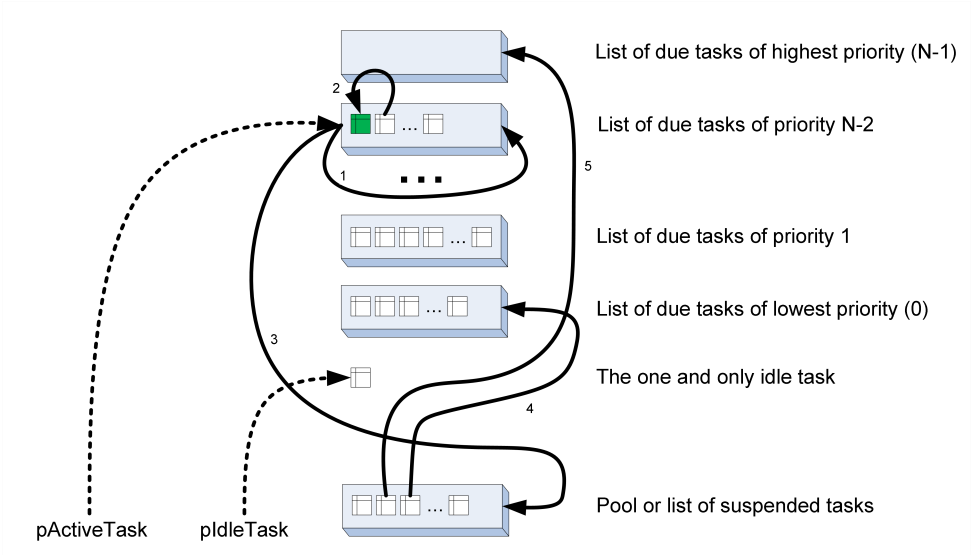

Figure 2.1: Scheduler of RT*uin*OS

and will therefore be the new active task.[9] If all tasks in this list are configured to have time slices (and if there were no other resume conditions), the list is cycled and all tasks get the CPU for a predefined amount of time.

Arrow 3 shows the effect of a suspend command. If the active task issues such a command, it is moved from the head of its due list to the end of the list of suspended tasks. Again, arrow 2 shows how the next task in the due list will become the new head and active task. However, if there was no second task in this list, the head of a due list of lower priority would become the new active task. In our figure, this could then be the head of priority list 1.

The explanation of arrow 3 needs some refinement. If RT*uin*OS is compiled without support for events of kind mutex and semaphore we indeed place the suspended task at the end of the list – but just for sake of simplicity of the implementation. The list of suspended tasks is not ordered; actually it is not a list but a pool of task objects. The situation differs if we have either mutex or semaphore events. Now we really have an ordered list of suspended tasks. The list is ordered by task priority and the suspended task is not put at the end but behind the last task of same or higher priority. This list order will strongly facilitate the distribution of posted events of kind mutex or semaphore to the right receiver in a later call of *rtos_sendEvent*.

Arrows 4 and 5 show how a suspended task becomes due again. In any call of either the system timer ISR or *rtos_sendEvent* all suspended tasks are checked if their resume condition became true. If so, the task object is moved from the list of suspended tasks to the end of their due list. The due list a task belongs to is predetermined at compile time, when the task priority is chosen. Actions 4 and 5 might appear in the same timer tic or call of *rtos_sendEvent* or in different ones.

Arrow 5 shows the resume of a task of the highest known priority: Here, we have an example where an event causes the interruption of a running task. The due list of the resumed task was empty before this action. Thus the resume creates a due task object of new maximum priority – higher than that of the task which was active so far. This less prior task is inactivated. From the perspective of the task execution its code flow is interrupted. Note, that the inactivated task object is not moved! It remains the head of its due list. The task is inactive but still due and still the first candidate for reactivation within its priority class.

[9] This is not fully correct: If the round robin action takes place in the due list which has not the highest priority, it can occasionally happen that a task of higher priority becomes due – and active – in the same timer tic.

2.7 Task Switches

The basic principle of the scheduler is to switch between different tasks. The chapters before explained the rules the scheduler applies to decide when and why to switch to which task. This section explains how a task switch is done.

The CPU doesn't know about tasks. It is a state machine, which has a set of registers, which determine the next action in the next tic of the CPU clock. The most relevant register in this context is the program counter (PC). It is a pointer to one specific word in the program memory (flash ROM). The word it points to is the next CPU command, which will be executed. This next command will probably have an effect on one or more of the CPU registers; it might for example command the CPU to add two data registers, to add a constant to one of the data registers, to read a memory cell into a register or to write a register into a memory cell. While the command is being executed the PC is incremented so that it points to the next word in the program flash. The next command is selected. And so on. A program is build from hundreds and thousands of these elementary commands. The sequence of commands form the program.

Since they are not endless it's possible to have several such command sequences one after another in the program memory. These different sequences now form the program code of the different tasks. The main thing to do when switching between tasks is to alter the value of the program counter from the one sequence to one of the others. There are dedicated CPU command to do so, like *jump*, which directly loads the PC with the desired target address or *rts* and *reti*, which load the PC from the stack.

Obviously, a scheduler needs to be able to later continue the left command sequence, or task respectively. The continuation needs to be seamless, to be done as if there had not been a switch to another sequence at all. This means in the first place, that we remind the value the PC had before we forced it to point to the other program sequence. But it also means that all the CPU's data registers have exactly the same value they used to have before the switch. This condition can only be fulfilled if we save the contents of all registers prior to switching to another command sequence.

The AVR CPU has 32 general purpose data registers plus a status register. The scheduler holds a task object for each task. It would be straight forward to have an array *uint8_t saveReg[32+1]* as member of the task object and to place the register contents here. Yes, this would basically work well but there is a much simpler and cheaper way to do: All the registers are pushed onto the stack. The ease of doing starts with saving the PC: As we saw, a task switch is always initiated by either an interrupt or the call of an API function (a suspend function or *rtos_sendEvent*). Both machine operations start execution with pushing the current PC onto the stack. As we intend to use the stack as storage location of all our registers it's fine to already have the PC here, where we want it to have. Saving the other registers on the stack is as easy: The command set of any CPU contains a push command which directly stores the contents of a data register onto the stack. For an AVR CPU this command is called *push*. Consequently, all interrupt service routines and all API functions, which could initiate a task switch, start with 32 push operations to save the general purpose data registers. (An important exception is mentioned in section 2.7.1.) This code sequence is completed by two simple CPU commands which also push the CPU's status register onto the stack.

The register pushing entry-commands of an ISR or a task switch causing API function basically enable this function to change the PC to point to the code of another task (and thus to continue with that task). Figuring out whether this is necessary will be the next step of these functions. Quite often this will not be the case. In which case the function terminates by doing all in reverse order. All registers are read and taken from the stack again. (They are "popped" from the stack.) This includes the PC as very last register – and the code execution continues where the interrupt had interrupted the task code, or behind the API function respectively. This is like any ordinary interrupt service routine.

Saving the registers on the stack is easy but only half the battle. Different tasks can't operate on the same stack. A stack is the implementation of the paradigm of strictly hierarchical function calls and nested sub-function calls and all the local data of these nested function invocations. The paradigm thus of many high level languages, among which C/C++. This paradigm is not applicable to multi-tasking. Tasks can be switched disregarding function entry and exit points; the activated task is not a sub-function

of the left other task.[10] Therefore, each task has its own, dedicated stack. Before switching from one task to another we need to save the stack of the left task. At runtime, i.e. out of scope of memory allocation and stack initialization considerations, the stack actually is completely represented by the pointer to its current top – and this "stack pointer" (the last CPU register to mention) is what we need to save also. Here, we use an absolute storage location. The task object has a member to hold the stack pointer of this task. The value of this object member is meaningful always and only while the task is inactive.

Before we proceed to the perhaps most difficult to understand detail, we will shortly summarize what we saw so far:

All code, which can initiate a task switch is a function. Either an ISR or a dedicated API function. All of these functions are well prepared to do a task switch as they push the PC, all data registers and the status register onto the stack of the active task. We say, they save the "CPU context" onto the task's stack. And in case a task switch should become necessary these functions know about a well-defined memory location where to also save the stack pointer of the currently active task.

Now we reach the maybe most tricky detail, easy on the one hand but nonetheless a bit difficult to see on the other hand. Let's put it into a question: How could we ever get here to the point where we are about to leave the currently active task? The answer: By the same kind of task switch from another task into this task! Consequently, the other task will have done the same. In this instance, there will be its dedicated stack and this stack will have all the saved registers of that task on its top (including the PC). So, if we load the stack pointer of that task, we just have to do the same as if we'd return from the task switch causing function back to the same task (see above): pop all registers and go ahead with that task.[11]

Important to see: All tasks which are not currently active have been deactivated the same way; all have left a stack with all registers including the PC on its top and all their stack pointers have been saved at the well-defined, known location. A picture of a task switch could be as follows: Pushing the registers on the stack is like climbing up a hill, from its top we jump to the top of any of the other hills around and then we slide it down (i.e. restore the registers with the contents of this other hill).

Doing the good preparation of storing the CPU context onto the stack the complete task switch reduces to nothing else than exchanging the value of the CPU's stack pointer. The value of the still active task is stored and the value of the newly activated task is loaded – that's all. Then we leave the task switch causing function and its simple register popping exit code loads the earlier saved context of the new task just like that into the CPU.

The general story is complete if we mention the initialization. Before the first task becomes active the still un-threaded initialization code of RT*uin*OS prepares the stack areas of all tasks as if these tasks had been active before and as if they had then been suspended. This is done by the simple, assembler code free C function *prepareTaskStack*. It places 33 bytes at the beginning of this memory area[12] – and these bytes will become the initial values of the data registers of the CPU when the task is activated the very first time (general purpose and status registers). And in front of these 33 bytes it places the two or three bytes of the wanted PC (depending on the type of AVR micro controller). At runtime this is the program memory location where to continue a task; now at initialization time it's obviously the start address of the task or – in C – the pointer to the task function. The address where the last byte has been placed is the value of the stack pointer that has to be stored in the task objekt.

When all stack areas are prepared the system timer interrupt is enabled and the first task activation a little bit later can in no way be distinguished from any later task switch.

[10] Prove: To continue the left task it is not a prerequisite that the activated task terminates – in order to "return" to the left task.

[11] The program counter is popped as the last register with a command *reti*, "return from interrupt".

[12] The logical beginning is meant. This is the last memory address in the stack area. The AVR architecture lets the stack grow from higher to lower memory addresses.

2.7.1 Interrupt versus API Function

Basically all task switch causing functions behave as described in the previous section. Task switch causing functions are either interrupt service routines or API functions. Interrupts can be the system timer tic or an application interrupt. The API functions are either the suspend command *rtos_waitForEvent*[13] or function *rtos_sendEvent* to post an event.

There's a significant difference between interrupt service routines and API functions. The API functions can return a result to the calling code but an ISR won't. When the ISR returns to a task (the one it had interrupted or another one in case of a task switch) all data registers need to be in the exactly same state as at the point of inactivation of this task. When API functions return, some registers may be altered and some others need to be altered. Data registers which pass parameter information to the function could be changed in the function without confusing the calling code after return. These "may be changed" conditions are disregarded by RT*uin*OS; the registers will be restored even if this is not necessary. Some other data registers, which are selected by the compiler to pass the function result back to the calling code however need to be altered; just restoring them like the other registers would mean not to return a meaningful function result. We only have the case of the suspend command *uint16_t rtos_waitForEvent*; the GNU compiler lets such a function return its 16 Bit integer result in register pair r24/r25.

Consequently, if a task switch activates a task, which had been deactivated by a suspend command, the context restore operation differs. All registers but the register pair r24/r25 are restored and r24/r25 is loaded with the function result. The final command *reti* then pops the PC from the stack and the task is continued (and will probably evaluate the function result in r24/r25 as one of its first operations).

Some details still need explanation. First the stack balance; please refer to figure 2.2. Since we save the CPU context on the task stack we must always push and pop the same number of bytes at each task switch. When restoring the context on return to an originally suspended task we will not restore r24/r25 from the stack (but load it with the function result) and therefore we must not push these registers when saving the context. The context save operation on entry into a suspend command will save all registers but r24/r25 - anticipating that these registers will not be restored later.

The task switch causing API function *rtos_sendEvent* is a void function. It doesn't return a result and all registers are thus saved on entry into *rtos_sendEvent* and restored on return to the calling task. (Which might be immediately or later because of releasing another, intermediately executed task of higher priority.)

A second, less important detail is about how the register pair r24/r25 is loaded with the function result. If the task switching code recognizes that the task to be activated had been suspended before it pushes the function result onto the stack as a last action before leaving the function. The code sequence to leave the task switch causing function (be it an ISR or an API function) can now be identical under all conditions: It pops *all* data registers, it pops the status register, it finally pops the PC and it'll continue with the new active task. This simple implementation idea requires that the save-context code pushes the registers not in their natural order, r0...r31, but the registers r24/r25 need to be pushed as last registers.

The last and important detail is about how to know whether an activated task had originally been suspended or not. The consideration of this detail also answers the question where the function result of the suspend commands comes from. The answers are depicted in figure 2.3.

A task object owns a member *postedEventVec*, which is null initially and when the task is active. It is still null on entry in state suspended. While a task is in state suspended it listens to all broadcasted events and it might wait for specific mutexes or semaphores. Whenever another task posts an event it is double-checked if this event is relevant to the suspended task. If so, it is added to member *postedEventVec* of the suspended task. This variable holds the set of events, which have been received by the task since it became suspended. As soon as the set of received events suffices to make the suspended task due again this variable is no longer changed.

A task can become due also on another path through the state chart. At any time the active task can

[13] This includes the derived suspend commands *rtos_delay* and *rtos_suspendTaskTillTime*.

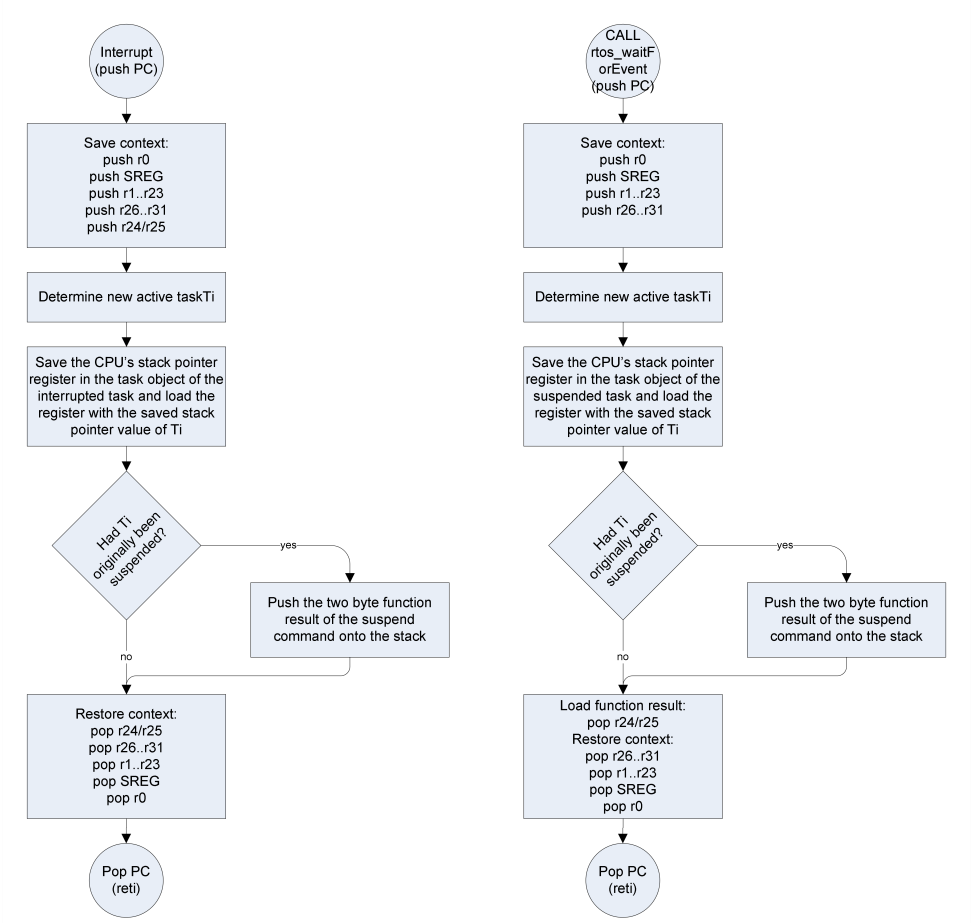

Figure 2.2: Implementation of task switches. On the left-hand side for interrupt service routines. On the right-hand side for the suspend commands: The only difference is to push or not to push the register pair r25/r25 on function entry. The API function *rtos_sendEvent* is not shown; it is implemented like an ISR, it pushes the register pair

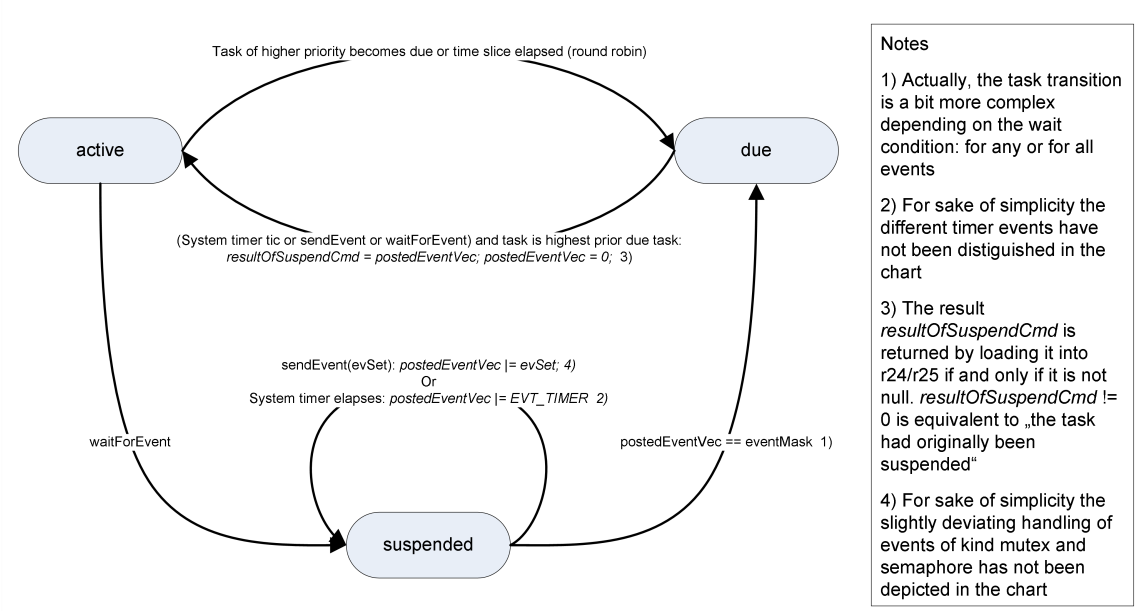

Figure 2.3: State model of a single task. The transition conditions are shown in default font, whereas the transition actions are shown in italics. The task variable *postedEventVec* is used as flag that indicates whether the task is going to return from a suspend command or not.

be inactivated because of any other task of higher priority becoming due (and active at the same time). If it is a round robin task a task can be inactivated when its time slice is elapsed. Inactivation means to transit from state active to state due.

The variable *postedEventVec* is always null in state active. If the task transits from active to due the variable is still null in state due and won't be changed here. If and only if a task transits from suspended to due the variable is not null. This way, the variable has a double meaning once we reach state due: In the first place it holds the set of received events that made the task due. This is the return value of the suspend command which originally made the task inactive. In the second place the variable holds the Boolean information whether the task will return from either a suspend command (which expects the result in r24/r25) or from an ISR or *rtos_sendEvent* (which must return with restored contents of r24/r25). This information is exploited by the task switching code.

The variable *postedEventVec* is cleared to null when activating the task, i.e. when doing the task switch to this task. Now it is prepared for the next cycle active → due → active, or active → suspended → due → active respectively.

Chapter 3

The API of RT*uin*OS

This chapter introduces the different functions of RT*uin*OS' application programming interface (API). The intention is to explain the meaning and use cases of the different API function, not all the details of the function signatures. Please refer to the doxygen documentation which directly builds on the C source code for details on function parameters, return values, side effects, etc.

3.1 Configuration of RT*uin*OS: rtos.config.template.h

Building an RT*uin*OS application starts with the configuration of the system. All elements of RT*uin*OS, which require application dependent configuration have been placed in the file *rtos.config.template.h*. This file is not part of the build, it's just a template for the file which will actually by used. Copy the template to your application source code and rename it to *rtos.config.h*. This is the name it has to have as this is the name used in the *#include* statements in the code. Open the renamed file in a text editor and read the comments. You'll find a number of *#define*'s which you need to adjust according to the demands of your application. The number of tasks (besides the idle task) you're going to use, the number of priority classes they belong to and the size of these classes are the most evident settings. You may enable the round robin strategy (by default it's turned off), you may configure your application interrupts and you may chose the resolution of the system timer. Please find a more in detail discussion of some of these topics below and in the comments in the source code.

3.2 Initialization of RT*uin*OS: *setup*

The RT*uin*OS application starts with a call of *setup*. This function is a callback from the Arduino startup code (and the RT*uin*OS initialization code at the same time) into your application. If you do not provide this function the linker will report a problem and refuse to build the executable. You may put all the initialization code of your application here. And you need to place the specification of the tasks of your application here. No other code location is possible to do this.

3.3 Specification of Tasks: *rtos_initializeTask*

From within *setup* you have to call *rtos_initializeTask* once per task. This function specifies the properties of a task. It's important to know, that the specific RT*uin*OS interrupts have not been started while *setup* is executed. You may thus interfere with any data objects owned by your tasks without consideration of race conditions and access synchronization.

The most important thing to specify is the executable code of the task, i.e. the task function. A specific function pointer type, *rtos_taskFunction_t*, has been defined for this purpose. A task function is a

void function with a single parameter. This parameter will pass the set of events to the task code, which made the task due the very first time after system startup; in the most typical use cases this will be just one of the two system timer events and may be ignored by the task code.

Typically, task functions cycle around (similar to the function *loop* in a traditional Arduino sketch). Some other RTOS offer a task termination but RT*uin*OS doesn't. In RT*uin*OS a task must never end. If it ever did it would "return" to the reset vector of the Arduino board and your application would start up again – and probably stay in a loop of repeatedly doing so.

The stack area of the task is specified by a pointer to a reserved address space in RAM and by the length of this reserved area. *Reserved* means that your application has to allocate the necessary memory. Since the stack area can never be changed at runtime it doesn't make sense to consider some dynamic allocation operations. Just define an array of *uint8_t* and pass its address and size.

Caution: Reserved also means that the specified stack area must never be touched by your application code.

If you configured RT*uin*OS to support the round robin strategy you will specify the duration of the time slices the task gets. Not all the tasks need to be subject to the round robin scheduling. If the duration of the time slice is set to 0 there is no such behavior for this task; with other words its time slice is of unlimited duration.

Another important part of the task specification is the initial resume condition. It is specified like at runtime when using the suspend command *rtos_waitForEvent*; please refer to section 3.6 for details. Your task will only start up if the condition specified now becomes true.

rtos_initializeTask uses an index to identify which task object is meant by the function call. The index has no particular meaning besides being a kind of handle to the same task object when later using the diagnostic functions *rtos_getTaskOverrunCounter* and *rtos_getStackReserve*. The index needs to be lower than the number of tasks but will not have an impact on the priority or order of execution of the tasks.

3.4 Initialization of Application Interrupts: *rtos_enableIRQUser<nn>*

Your application may configure RT*uin*OS to use its own interrupts which trigger dedicated tasks. If you configure an interrupt a callback into your application code is made as part of the system startup. The function *rtos_enableIRQUser<nn>* is intended to release your interrupt <nn>. This typically means to configure some hardware registers of a peripheral device and to finally set the so called interrupt mask bit.

You must never try to do this as part of the general initialization code in *setup*: At this point in time the task, which is coupled to the interrupt is not ready to run and releasing the interrupt now could cause unpredicted behavior of the service routine of your interrupt.

On the other hand, when *rtos_enableIRQUser<nn>* is invoked, the RT*uin*OS system is up and running (besides your interrupt) and all implications with respect to race conditions and data access synchronization need to be considered.

Your application needs to enable the interrupt source but it doesn't have to implement a service routine. This routine is part of the RT*uin*OS implementation. Its standard action – which can not be changed – is to post a dedicated event. Your application will surely specify a task of high priority which cyclically waits for this event and implements the actual interrupt service code.

The specification of the interrupt source is one detail of the RT*uin*OS configuration made in *rtos.config.h*.

Currently, RT*uin*OS implements up to two application interrupts (i.e. <nn> is either 00 or 01), but it's simple and straight forward to extend the implementation to more interrupts if required.

After return from the last callback *rtos_enableIRQUser<nn>* your application is completely up and running.

3.5 The idle Task: *loop*

Once the system is started it cyclically calls the void function *loop* which has to be implemented by your application. If you do not provide this function the linker will report a problem and refuse to build the executable.

The repeated call of *loop* is the idle task of RT*uin*OS. This means the execution of this code is done only if no other task requests the CPU. The execution speed of *loop* is directly dependent on the activities of your tasks. Therefore it typically contains code which has no real time constraints.

A typical use case of the idle task is to put some diagnostic code here. For example, RT*uin*OS permits to double-check the usage of the stack areas. (If a task would ever exceed its reserved stack area an immediate crash is probable; which is a hard to find bug in the code.) This code is quite expensive but when located in the idle task it doesn't matter at all. The only impact of expensive code in the idle task is that the results will be available somewhat later or less frequently. For a diagnostic function this is typically uncritical.

RT*uin*OS supports typical use cases where always at least one task is due. An example is a couple of tasks, all continuously running, and scheduled by the round robin strategy. They are cyclically activated but never suspended. In this situation *loop* will never be called. However, as a prerequisite of successful code linkage it's nonetheless required to have it.

If no idle task is required or if there's no idle time left simply implement *loop* as an empty function.

3.6 Suspend a Task: *rtos_waitForEvent*

A task in RT*uin*OS stays due as long as it desires. If it has finished or if it becomes dependent on the work result of some other task or on an external event (reported by an application interrupt) it will suspend itself voluntarily.

In a technical system, a task is often applied to do a regular operation, e.g. read and process the input value from an analog-digital converter every Milli second.[1] Here, "finished" would mean having performed this action. When the value of this Milli second has been processed, the task would suspend until the next Milli second interval begins. Suspending always includes a condition under which the suspended state ends – in our example it would be the absolute timer event and its parameter *when* would be set to the next Milli second. From the perspective of the task code flow, voluntarily suspending always means to wait for something and doing nothing until. This explains the name of the suspend function. With the view on the complete system, suspending means to return the CPU and to pass it to other tasks, which currently don't have to wait for whatever events.

In RT*uin*OS a task can suspend and wait

- until a point in time,
- for a while,
- until a set of events has been posted by other tasks (or a timeout has elapsed meanwhile),
- until at least one event out of a set has been posted by other tasks (or a timeout has elapsed meanwhile).[2]

The signature of the suspend command has a set of events as bit vector (with up to 16 bits or events respectively), a Boolean operator (all events required or any event releases the task) and a time parameter.

The events can be of any type, ordinary broadcasted, mutex or semaphore. Successfully waiting for an event of kind mutex or semaphore means to acquire the synchronization object, or getting ownership of the associated, managed resource respectively.

[1] In the RT*uin*OS default configuration the system timer tic is about 2 ms; a one Milli second task can't be implemented without a configuration change.

[2] Actually, the first two conditions are special cases of the last two: The set of events to wait for just contains a timer event but nothing else.

```
static void regularTask(uint16_t initialResumeCondition)
{
#define MY_TASK_CYCLE_TIME 1 /* Unit is system timer tic, e.g. 1 ms */
    do
    {
        /* Actual task implementation: read ADC, process value... */
        readADCAndProcess1ms();
    }
    while(rtos_waitForEvent( RTOS_EVT_ABSOLUTE_TIMER
                           , /* waitForAllEvents */ false
                           , /* when */ MY_TASK_CYCLE_TIME
                           )
         );
} /* End of regularTask */
```

Listing 3.1: Typical use case: regular task

The time parameter doesn't care if no timer event is part of the set of events. If the absolute timer event is in the set the time parameter has the meaning *when*. If the relative or delay timer event is in the set the time parameter has the meaning *after*. Consequently, it is not allowed to have both timer events in the set.

A bit specific is the parameter *when* of the absolute timer. The most typical use case of the absolute timer event is the implementation of a regular task; in our example above a task, which is activated every Milli second. See listing 3.1: The implementation will place the action into an infinite loop. The *while* condition at the end of the loop will be a call of *rtos_waitForEvent*, addressing to the absolute timer event. In each loop the next Milli second is passed as parameter *when*. This would require an accumulating variable in the implementation, which is updated in every loop. To avoid this, the parameter *when* is defined to be a difference, the difference to the last recent reference to the absolute timer. This would mean in our typical use case, that parameter *when* becomes a constant. In every loop, the parameter simply is 1 [ms] in the call of *rtos_waitForEvent*.

Please note, if a task waits for a set of events and it wants to be released when all events are posted to the task, then the condition *all* does only refer to the application events (ordinary, mutex or semaphore) but not to a timer event. Timer events will still release the task as soon as they occur. The meaning of timer events being timeouts is not affected by the choice waiting for all or for any event.

3.6.1 Events of Kind Mutex

An event to wait for can be a mutex, a synchronization object for mutual exclusion of tasks from a resource. The kind of an event is specified at compile time, it's part of the configuration of RT*uin*OS, see [4].

From the perspective of a task waiting for a mutex is just like waiting for ordinary broadcasted events. The task is suspended until the mutex is available. This will be the case when the other task, which currently owns the mutex will release it using *rtos_sendEvent*. Or immediately, if no other task currently owns the mutex.

If a task suspends waiting for a mutex it can happen that *rtos_waitForEvent* doesn't suspend the task but immediately returns. Please refer to section 3.6.3 for more.

3.6.2 Events of Kind Semaphore

An event to wait for can be a semaphore, a synchronization object for managing access to a pool of resources shared between different tasks. The kind of an event is specified at compile time, it's part of

the configuration of RT*uin*OS, see [4].

Waiting for a semaphore and getting it means to decrement the semaphores internal count. The task is granted access to one instance of the resources which are managed by the semaphore.

Please note, that handling semaphores as Boolean valued events (being there or not being there) hinders to provide a more generalized API to semaphores. In RT*uin*OS it is by principle impossible to acquire several instances from the resource pool at once. An application which needs to have several instances is obliged to repeat the invocation of *rtos_waitForEvent* that number of times. Although this eventually leads to the required number of instances it's not equivalent with respect to the distribution pattern of the shared resources. The same consideration applies to returning/releasing a semaphore using *rtos_sendEvent*.

From the perspective of a task waiting for a semaphore is just like waiting for ordinary events. The task is suspended until one instance (or counter value respectively) of the semaphore is available. This will be the case when any other task releases such an instance using *rtos_sendEvent*. Or immediately, if there's currently at least one instance left in the semaphore.

If a task suspends waiting for a semaphore it can happen that *rtos_waitForEvent* doesn't suspend the calling task but immediately returns. Please refer to section 3.6.3 for more.

3.6.3 Notes on waiting for Events of Kind Mutex or Semaphore

It may happen that immediately available mutexes or semaphores satisfies the wait condition of the calling task: It waits for any event among which an immediately available mutex or semaphore. Or it waits for all events out of a set, but all of these are immediately available. If so, the call of function *rtos_waitForEvent* won't suspend the task. Instead *rtos_waitForEvent* simply returns like an ordinary sub-routine and the return value indicates the acquired mutexes and semaphores.

If a task waits for one or more mutexes or semaphores it needs to evaluate the return code of the function. If a set bit corresponds to a mutex event the task is the new owner of this mutex. If a set bit corresponds to a semaphore event the task got access to one instance of the resource pool managed by this semaphore. It'll own the mutex or have access to the resource until it voluntarily releases the mutex or semaphore using *rtos_sendEvent*. Since wait conditions can be complex, the return from the function as such is no safe indication that a mutex or semaphore has been acquired by the task. The simplest example is a wait for a mutex event with timeout and the timeout elapses before the mutex is released by the it owning task.

Caution, properly keeping track of the ownership of mutexes and instances of semaphores is fully in the responsibility of the application. RT*uin*OS won't bother with who has which mutexes or semaphore. It'll accept a mutex/semaphore release command by any task regardless whether this task actually owns it or not. By means of the return value of *rtos_waitForEvent* RT*uin*OS just provides the information to the application to enable it to implement proper ownership handling.

3.6.4 *rtos_suspendTaskTillTime*

To further support the typical use case of regular tasks, there's an abbreviation of the call of *rtos_waitForEvent* as sketched in listing 3.1. Instead of calling *rtos_waitForEvent* one can call *rtos_suspendTaskTillTime*. The only parameter of the function is the parameter *when*.

rtos_suspendTaskTillTime is implemented as preprocessor macro, so there's no difference in comparison to directly using *rtos_waitForEvent* except for the readability of the code.

3.6.5 *rtos_delay*

Another abbreviated call of *rtos_waitForEvent* supports the condition "wait for a while" (but not for any specific event). You may use the preprocessor macro *rtos_delay* for this. The only macro parameter is the timer parameter, which specifies the delay time (or the time to stay suspended respectively).

There's no difference in comparison to directly using *rtos_waitForEvent* except for the readability of the code.

3.7 Awake suspended Tasks: *rtos_sendEvent*

Timer events are entirely managed by the system, all other events will only occur if they are posted by the application code. This can be done either by application interrupts or by invoking the API function *rtos_sendEvent*.

The only parameter of the function is the set of events to be posted, implemented as a bit vector of 16 bits. Neither the timer events nor application interrupt events must be posted; there remain (dependent on the configuration of RT*uin*OS) twelve to fourteen events, which are directly handled by the application task code.

There are three kinds of events: Ordinary events, mutex events and semaphore events. The type of each of the available events is defined at compile time as part of the configuration of RT*uin*OS done in [4].

There are predefined names for the available events, please refer to [2]. Regardless, you may also define your own, suitable names. Each name is defined to be the value of the event bit; consequently, sets of events can be expressed by sums or binary OR (`|`) terms of these names.

A typical use case of application handled events are producer-consumer models. One task prepares some data and signals availability to the data consuming task by setting an event. Obviously, the consumer starts with waiting for this event.

An ordinary event is broad-casted only to the currently suspended tasks and is not stored besides that. If a task suspends shortly after another one has posted such an event, the suspended task will never receive this event and may stay suspended forever.

Mutex and semaphore events differ and are the better choice to handle inter-task communication in many use cases. Please, refer to sections 2.3.1 and 2.3.2 for details about the meaning and handling of these kinds of events.

3.8 Data access: *rtos_enter/leaveCriticalSection*

In all relevant use cases, tasks will share some data. Some tasks will produce data, others will read it. If your application has tasks of different priority, this becomes a matter. Except for a few trivial examples like reading or writing a one Byte word, all data access is not atomic, i.e. can be interrupted by any system interrupt. The software has to anticipate that this is an RT*uin*OS system timer interrupt or an application interrupt which can easily cause a task switch. A task can be in-activated while it is busy updating the data and another task can continue operating on the same, half way completed data. The results are unpredictable and surely wrong. Be aware, even an atomic looking operation like `++u`, where *u* is a of type *uint8_t*, is unsafe and requires protection.

The pair of API functions *rtos_enterCriticalSection* and *rtos_leaveCriticalSection* makes any portion of code which they enclose atomic – and thus safe with respect to shared access from different tasks.

rtos_enterCriticalSection simply inhibits all those interrupts, which can cause a task switch and *rtos_leaveCriticalSection* re-enables all those interrupts.

An application may implement interrupts, which can set an RT*uin*OS event and cause a task switch. These interrupts are obviously relevant to *rtos_enter/leaveCriticalSection*, they need to be inhibited also. Consequently, if your application implements interrupts you will have to extend the default implementation of the pair of functions. The functions are implemented as preprocessor macros in the application owned RT*uin*OS configuration file [4] and their modification should be straight forward.

The two functions do not save and restore the interrupt-inhibit state. After any *rtos_leaveCriticalSection*, all interrupts are surely enabled. Therefore, the pairwise calls of the functions can't be nested. The code in the outer pairs wouldn't be protected.

The pair of functions *cli* and *sei* from the AVR library nearly has the same meaning and can also be used to make data access operations atomic. The difference is that they inhibit all interrupts. The

responsiveness of the system could be somewhat degraded without need, e.g. the Arduino time functions like *delay* or *millis* could suffer. On the other hand, these two functions are a bit cheaper in terms of CPU load. We suggest to use them if the protected code sequence is rather short, e.g. just one or a few simple assignments and to use *rtos_enter/leaveCriticalSection* otherwise.

In RT*uin*OS a task of higher priority will never become inactive in favor of a lower prioritized task as long as it doesn't suspend itself voluntarily. And if the task is not a round robin task it'll even never become inactive in favor of an equally prioritized task (as long as it doesn't suspend itself). Therefore,

- a normal task of same or higher priority doesn't need atomic operations to access data it shares with other tasks of same or lower priority,
- a round robin task of higher priority don't need atomic operations to access data it shares with other tasks of lower priority.

But vice versa, their counterparts of same or lower priority of course need to protect their access code to the same, shared data.

In cooperative systems tasks generally don't need to protect their access to shared data as tasks will never be interrupted at unforeseeable (and undesirable) points in time. In RT*uin*OS cooperative multi-tasking applications are implemented by tasks all belonging to the same priority class.

To summarize,

- always put your data access code into a pair of protective functions if the task shares this data or parts of it with at least one other task of higher priority,
- always put your data access code into a pair of protective functions if the task has round robin characteristics and if it shares this data or parts of it with at least one other task of same or higher priority,
- use *rtos_enter/leaveCriticalSection* as pair of protective functions if the access code is complex,
- use *cli/sei* as pair of protective functions if the access code is trivial.

3.8.1 Mutex versus Critical Section

Accessing shared data from different tasks can safely be done with different code patterns. The critical section can be used or a mutex can be applied. There are significant differences and implications.

Using a mutex is a kind of contract between different tasks. They agree to access the shared data only when they own the mutex and they promise to own it no longer than absolutely necessary. There's however no technical means which would ensure this; any task can actually access the shared data at any time and corrupt it. Moreover, there's no direct, visible link in the source code between the mutex and the shared data. The code samples coming along with RT*uin*OS demonstrate two techniques to overcome this: they either re-define the generic name of the mutex so that the relation to the shared objects become apparent or they hide the operations to acquire and release the mutex in the same class that implements the shared object.

A critical section is a technical mechanism, which hinders the scheduler to switch to any other task (including interrupts and their handler tasks) while the enclosed code is executed. Handling an interrupt or activating another task is moved in time until immediately after the execution of the code inside the critical section.

Blocking the scheduler is the reason, why critical sections may be used only for short code sequences, typically some update operations of a data structure. An example would be queue, which an element is appended to or popped off. The code to append and to pop (a few lines each) would be placed into a critical section. It would already be bad style to place some debug or feedback code like *Serial.println("Element appended")* in the critical section. The execution of this operation takes much too long.

A very common code pattern when using critical sections is to have a local copy of the shared data in the task. The critical section is just used to copy the shared data into the local copy; all further,

time consuming processing and user feedback operates on the copy only and is of course done outside the critical section. And the same vice versa for updating the shared data. An n-fold of data consumption is the consequence, as each accessing task needs to have its local copy.

A mutex grants ownership to the shared data across any number of task switches. The scheduler stays fully operational. Only those tasks, which also want to have the mutex (but uttered their demand a bit later than the lucky one) are temporarily out of the loop. They are suspended. Please be aware, the mutual blocking of tasks that all demand the same shared resource will probably be in conflict with the requirement of having strictly regular tasks. If you have regular tasks, you will have to take care when applying mutexes to organize them. Specifying a timeout when waiting for a mutex may be a way out.

Basically, a mutex could be acquired using *rtos_waitForEvent* and released using *rtos_sendEvent* for a short, fast operation; the pair of functions could be used like the other pair *rtos_enter/leaveCriticalSection*. However, this is much, much more expensive in terms of CPU load and must not be done if a critical section is applicable. Examples, where a mutex reasonably replaces a critical section is when doing console output with *Serial* or when accessing the LCD using the LiquidCrystal library.

3.9 Diagnosis: *rtos_getTaskOverrunCounter*

Each task has a built-in overrun counter. The meaning of this counter is well defined only for regular tasks. These tasks want to become due at fixed points in time. If too many tasks have too much to do it may happen that it is not possible to make a task due at the desired point in time. This is then a task overrun event. It is counted internally. This function reads the current value of the counter for a given task.

Using this function, the application can write some self-diagnostic code. However, if such events are seen, there's barely anything to do at runtime. Evaluating the counters should be considered a kind of debug information, a hint at development and testing time that the implementation is still insufficient and needs changes.

There are two pitfalls. The function checks for a task overrun as part of a suspend command submitted by a task. It recognizes a task overrun if the demanded time of resume is found to be already over. The current system time is compared to the demanded time of resume. The comparison may cause an arithmetic overflow because of the limited length of the internally used system time. Particularly if an application uses the 8 Bit system time there's a significant probability of taking the false decision:

If a regular task chooses a too long period time a task overrun might be recognized where actually no such overrun occurs. This will not only lead to a wrong counter value but also to bad task timing. A task found to be overdue is made due immediately, which is the wrong decision in this case.

If a task excessively exceeds its nominal regular execution time the task overrun won't be recognized. Task calls are silently skipped.

Please refer to section 4.3. Here you find a discussion how to choose the appropriate system time for your application.

The function *rtos_getTaskOverrunCounter* may be called by any task. Please note, that the function is not defined for the idle task, but may be called by the idle task for querying any of the other tasks.

3.10 Diagnosis: *rtos_getStackReserve*

A simple algorithm determines the usage of the task stacks. (In any RTOS, each task has its own, dedicated stack.) The maximum size of the stacks is predetermined at compile time and determining the actual stack usage at runtime is just a development tool. If the maximum stack size of any task is exceeded the system will surely crash and the cause of the crash will be hard to find. Allocating the stack sizes much too large is too expensive with respect to RAM usage. Therefore, by applying this function, you can keep an eye on the actual stack usage during development and testing phase and reduce the allocation to a reasonable value.

There are two pitfalls. The algorithm searches the stack area for a specific byte pattern the complete area has been initialized with, and which is typically not written into the stack at runtime. However, it could be written at runtime with a low but significant probability. As a result, the actual stack usage can be one Byte more than computed with a probability that must not be neglected. It's however much less probable that two such bytes will be written consecutively into the stack at runtime – the probability that the computed number of bytes is too little by two is much less. And so forth. If you add a number of five Byte to the computed stack usage the remaining probability that this is less than the actual stack usage is negligible.

The current stack usage increases suddenly by 36 Byte in the instance of a system interrupt (the CPU context of the interrupted task is saved onto the stack of this task). *rtos_getStackReserve* returns the maximum stack usage so far (actually it returns the inverse value, the *reserve*, but this is equivalent). This is a useful value once your testing code ran through all code paths, particularly through the deepest nested sub-functions. You can ensure this by dedicated test routines. But can you also be sure that a system interrupt occurred in the very instance of being inside the deepest nested sub-routine? If not, it'll surely happen sooner or later and another 36 Byte of stack will be consumed. It's good practice to add another 36 Byte to the computed stack usage.

Summarizing, you should add 41 Byte to the computed stack usage before reducing the stack size to the really needed value.

The function *rtos_getStackReserve* may be called by any task. Please note, that the function is not defined for the idle task, but may be called by the idle task for querying any of the other tasks.

3.11 Diagnosis: *gsl_getSystemLoad*

The measurement of the system load is an important development tool for RTOS applications. The system load gives an indication of the average usage of the CPU, i.e. the time the CPU spends on task execution in comparison to the time left, its idle time. This value completes the information provided by the API function *rtos_getTaskOverrunCounter*. An accurate measurement of this quantity is difficult and RT*uin*OS doesn't really offer it. However, a simple algorithm has been implemented, that computes a quite good and useful estimation of the CPU load.

The algorithm is available as a independent C source file. It had already been published in a previous release of RT*uin*OS as part of one of the code samples. Now, the required files, [7]and [8], have been integrated into the RT*uin*OS installation and can be used by any application just like that. Please note, that it must still not be considered as a true part of RT*uin*OS.

There are barely prerequisites to apply the CPU load estimation in your RT*uin*OS application. Include the header file *gsl_systemLoad.h* and call function *gsl_getSystemLoad* from the idle task. It returns the estimated average CPU load after about a second of system observation time. Please refer to [8] and [7] for details.

The idea of the load estimation is to compare the world time spent on a known sequence of machine codes to the sum of CPU clock tics required by the sequence. The execution of the test sequence is done in a task having a priority being lower as any relevant application task, typically the idle task. Due to its low priority, this task will have an impact on the scheduling scheme of the other, observed application relevant tasks. The ratio of the two time designations indicates the percentage of the world time the CPU spends on the application relevant tasks but not the the test executing task. The value is a direct measure for the average CPU load, if the duration of the test sequence execution is long in comparison to the application task switches.

By principle, the implementation of the load estimation has the disadvantage of either consuming all the system idle time or - if interleaved with other idle task operations - only returning estimated samples of the system load. It depends on the application design and the configuration of the function how well the samples match the reality.

Chapter 4

Writing an RT*uin*OS Application

4.1 Short Recipe

Create an empty folder in folder *code/applications*. The name of the folder is the name of your application. Copy the configuration template file [3] into this folder and rename it to *rtos.config.h*.

Open *rtos.config.h* and configure the number of tasks, the number of different priority classes and the size of these. Do not use empty priority classes, this wastes expensive memory. Priorities should always be counted 0, 1, ..., max. Configure the number of mutexes and semaphores in use. Select the word width of the system time. Often an eight Bit value will be sufficient. Please refer to 4.3. In general, you'll find a lot of hints and comments in the configuration file telling you what to do in detail.

Open a new C source file in the same folder. This file implements the core of your application. You need two standard functions, *setup* and *loop*, your task functions and some static data.

Create a static array for each task. The type is *uint8_t*. This array will become the task's stack area. As a rule of thumb a size of 100 ... 200 Byte is a suitable starting point. Later, you may apply *rtos_getStackReserve* to get a better idea.

Create empty task functions: `static void taskFct(uint16_t)`.

In *setup* you will call *rtos_initializeTask* once per task. Pass the pointer to the task function, the stack area and specify the priority and the condition under which the task becomes initially due (probably: immediately).

Create the empty function *loop*: `void loop(void)`.

Now fill the task functions with useful functional code. Be aware, that a task function must never be left, a system reset would be the consequence. Therefore, you will always implement an infinite loop, e.g. using *rtos_suspendTaskTillTime*. Find an example in listing 4.1. *loop* may remain empty if you don't need idle operations.

```
static void task10ms(uint16_t initialResumeCondition)
{
    do
    {
        /* Place actual task code here, e.g. the call of an external
           function. */
        myActual10MsTask();
    }
    while(rtos_suspendTaskTillTime(5 /* unit: 2ms */);
}
```

Listing 4.1: Typical task, regularly activated

When implementing the functional code always be aware of the discussion of protecting the access to data shared between tasks; please refer to section 3.8.

Compile your application using the generic makefile. Double-check your environment: The make tool needs to be on the Windows/Linux search path and the environment variable *ARDUINO_HOME* needs to point to your Arduino installation (see section 4.2.1). If you placed all your code in the single folder created at the beginning, all you have to do now is to run

```
make -s APP=myFolderName build
```

Start your application using the makefile. Your Arduino board is connected via the USB cable. Be this COM6.[1] Now run

```
make -s APP=myFolderName COM_PORT=COM6 upload
```

The RT*uin*OS source file [1] must not be touched at all. Just open it for reading if you what to understand how RT*uin*OS works.

4.2 The Makefile

RT*uin*OS as such can't be compiled, it's just a C source file which has to be compiled with your application. (Without application code you'd end up with unresolved externals when linking the code.) However, RT*uin*OS is distributed with some autonomous test cases which are true RT*uin*OS applications. All of these can be built and uploaded using the make tool, which is part of the Arduino installation and the makefile which is part of the RT*uin*OS distribution.

You can organize your applications similar to the test cases. Then you will be able to use the makefile without changes for your application, too. Even if your application grows and needs a more complex folder structure than the test cases to organize the source code you will still be able to use the makefile, however with some simple changes.

Most often, you will do as follows: Open a Shell window (i.e. a Command Prompt or Powershell window under Windows or Bash under Linux) and cd to the root directory of the RT*uin*OS folder. This is where the file GNUmakefile is located. Connect your Arduino MEGA board[2] to the USB port, e.g. port number 6. Now type

```
make -s APP=tc05 COM_PORT=COM6 upload
```

and test case (or application) *tc05* is compiled and uploaded to the board. The Arduino board is reset and the RT*uin*OS application is started. Now you might consider to open the Arduino IDE and open the Serial Monitor (i.e. the console window) to see what's going on. Later, you might replace the default COM port in the makefile with your specific port number and the command line becomes even shorter.

The makefile explains itself by calling make as follows. The command requires that the current working directory is the root directory of RT*uin*OS. Calling make with target *help* will print a list of all available targets to the console with a brief explanation:

```
make -s help
```

An RT*uin*OS application can't be developed as a sketch in the Arduino IDE. We made some minor changes of the Arduino file *main.c*, which would be lost when using the IDE. You may however continue to use the IDE for console I/O if you use the object *Serial* for communication – which is particularly useful during application development and which is supported by the makefile process even better than by the IDE as the makefile enables you to have the I/O commands only conditionally in the code. Their presence may be restricted to a development compilation configuration by means of a `#ifdef DEBUG`.

[1] Use the Arduino IDE to find out which port your board is connected to. Typically, after connecting your board it'll be the last port shown in menu Tools/Serial Port. Or open the Windows Device Manager and look at "Ports (COM & LPT)". Here you find your connected board with name and assigned COM port.

[2] Other boards need some code customization first. Please, refer to section 4.7.

4.2.1 Prerequisites

The name of the makefile is GNUmakefile. It is located in the root directory of the RT*uin*OS installation. The makefile is compatible with the GNU make of Arduino 1.0.5, which is "GNU Make 3.81".

Caution: There are dozens of derivates of the make tool around and most of these are incompatible with respect to the syntax of the makefile. Even GNU make 3.80 won't work with RT*uin*OS' makefile as it didn't know the macro `$(info)` yet. Furthermore, there are incompatible Windows ports of the GNU make tool, e.g. by MinGW and Cygwin.

To run the make tool it might be required to add the path to the binary to the front of – to avoid shadowing by an incompatible derivate of make – your Windows search path.[3] Inside your Arduino 1.0.5 installation, you'll find the make tool as arduino-1.0.5/hardware/tools/avr/utils/bin/make.exe.

The makefile compiles the Arduino standard source files to the standard library *core.a*. To do so, it needs to know, where the Arduino sources – which are not part of this project – are located. It expects an environment variable *ARDUINO_HOME* to point to the Arduino installation directory, e.g. c:/ProgramFiles/arduino-1.0.5 on a Windows system. You will probably have to add such a variable to your system variables as it is not defined by the Arduino standard installation process.

Another prerequisite of successfully running the makefile is that your application doesn't have any two source files of same name – although this would be basically possible with respect to compiler and linker if they were located in different folders.

Finally, all relevant paths to executables and source files and the file names themselves must not contain any blanks. This includes the path to the Arduino installation. If you are a Windows user, it particularly means that you will have to reinstall Arduino if you should have placed it into the standard installation path c:/Program Files/arduino-1.0.5.

4.2.2 Concept of Makefile

The makefile has a very simple concept. A list of source code directories is the starting point of all. All of these directories are searched for C and C++ files. The found files are compiled and linked with the Arduino library *core.a*.

Post processing steps create the binary files as required for upload to the Arduino board.

An optional, final rule permits to upload the binary code to the Arduino board. If your application makes use of the USB communication with *Serial* all you still have to do on a Windows system to make your application visibly run is an ALT-TAB to switch to the (open) Arduino IDE and a Ctrl-Shift-m to open the console window.

On the compilation output side, for sake of simplicity of the makefile, the folder structure of the source code is not retained. All compilation products (*.o, among more) of an RT*uin*OS application are collected in a single output folder dedicated to this application. This is the reason, why there must never be two source files of same name. Even *myModule.c* and *myModule.cpp* would lead to a clash.

In the Arduino IDE the library *core.a* is source code part of the sketch. It is rebuilt from source code after a clean. The RT*uin*OS makefile also contains the rules to build *core.a* but it considers it a static part of the software, which is in no way under development. It'll be built if it's not up-to-date but it'll neither be deleted and rebuilt in case of a rebuild (i.e. target *clean*) and nor does its build depend on the compilation configuration.

As said, the compilation is mainly controlled by a list of source code directories. This list is implemented as value of macro *srcDirList*. The default is to have two directories: The RT*uin*OS source code directory *code/RTOS* and a second, variable directory. This directory is located in *code/applications* but its name is provided by macro *APP*. This enables you to select an application for compilation simply by stating `APP=myRTuinOSApplication` on the command line of make.

If your application demands more than a simple, flat directory to manage all its source files, you can continue to use the makefile. The makefile knows a kind of "callback". If present, an application owned makefile fragment is read prior to do compilation and linkage. This makefile fragment can override or

[3] After installation, type make –version to find out.

```
#ifdef DEBUG
    /* Some self-diagnostic code */
    Serial.print("Current stack reserve: ");
    Serial.println(rtos_getStackReserve(IDX_MY_TASK));
    if(rtos_getTaskOverrunCounter(IDX_MY_TASK, /* doReset */ true) != 0)
        doBlinkLED(3 /* times */);
#endif
```

Listing 4.2: Usage of preprocessor switches supporting different compile configurations

extend the default value of the makefile variable *srcDirList*. Specify different folders or just add some. The fragment is looked for by name: It needs to be called <*appName*>*.mk*, where <*appName*> is the value of the makefile variable *APP*.

Folder Structure

The makefile is organized in different files. The entry point is the file *GNUmakefile*. This file name avoids the need to use the make tool's command line switch *-f* <*makefileName*> and it avoids conflicts withs other, incompatible derivates of make.

File *GNUmakefile* only contains some settings common to all RT*uin*OS applications. Here, you would e.g. find the default setting for the COM port to be used for upload of the compiled software (please see 4.2.5 also).

After reading these settings, the "executable" makefile *compileLinkAndUpload.mk* is read. It is located in the sub-folder *makefile*. It in turn builds on some sub-routines, which are implemented in further included makefile fragments. All of these are located in the same sub-folder *makefile*.

4.2.3 Compilation Configurations

The makefile supports different compilation configurations. On makefile level, a configuration is nothing else than a set of C preprocessor macros, which is passed on to the compiler. The meaning of the configurations is completely transparent to the makefile and just depends on the usage of the preprocessor macros in the C source code.

Two configurations are predefined (and used by the RT*uin*OS source code) and any additional number of configurations can be created by a simple extension of the makefile.

The standard configurations are called *DEBUG* and *PRODUCTION*. *DEBUG* defines the C preprocessor macro *DEBUG* and configuration *PRODUCTION* defines the C preprocessor macros *PRODUCTION* and *NDEBUG*.

Our recommendation is to place some appropriate self-diagnostic code in the C/C++ source code, which is surrounded by preprocessor switches. This includes diagnostic output using the *Serial.println*, the USB communication with the Windows machine, which is particularly useful during development and testing phase but usually not required in a true application of the Arduino board. An example can be found in listing 4.2.

A specific example of such code is the macro *ASSERT*, which expands to nothing if *DEBUG* is not defined (as in configuration *PRODUCTION*) but which double-checks some invariant code conditions during development, when configuration *DEBUG* is used. Please see [5]. The RT*uin*OS code itself make intensive use of *ASSERT* in order to report the most probable user errors in *DEBUG* configuration.

The configuration is chosen by the value of the makefile variable *CONFIG*. The default value is *DEBUG* but this may be overridden on the command line of the makefile; please find an example:

```
make -s APP=myFolderName CONFIG=PRODUCTION COM_PORT=COM6 upload
```

4.2.4 Selecting the Arduino Board

The target platform is selected as value of macro *targetMicroController* in the makefile. However, not all Arduino boards are currently supported by the implementation of RT*uin*OS. If you select a micro controller, which is not yet supported, you will run into error directives in the source code. Please refer to section 4.7 for more.

The makefile has not been tested with any Arduino boards other than the Arduino Mega 2560. Please be aware that additional changes on the makefile could be necessary: Different controllers may require different command line options of compiler, linker and flash tool. These differences are not yet anticipated by the makefile. You need to double-check all recipe lines, which are typically composed using macros like *targetMicroController*, *cFlags* and *lFlags*.

You can use the Arduino IDE to find out, which command lines are appropriate in your specific environment. In the file menu of the IDE you can navigate to the properties dialog. Here, you should check the verbose output for both, compilation and upload. Now select, build and upload one of the sample sketches in the IDE. Copy the contents of the IDE's output window and paste them into a text editor. You will find appropriate command lines for all the tools.

There's a pit-fall: When running the flash tool avrdude the Arduino IDE uses a protocol which unfortunately requires an additional, preparatory reset command. This protocol works with the makefile only if you press the reset button shortly before avrdude is run, which is at least inconvenient if not unacceptable. The makefile uses a quite similar protocol, which works well and doesn't require the reset. Place `-cWiring` in the command line of avrdude instead of `-cstk500v2`.

4.2.5 Selecting the USB Port

The USB (or COM) port which is used for the connection between PC and Arduino board has to be known by the rule, which uploads (and flashes) the compiled and linked application. You can specify the port as part of the command line of the makefile (type e.g. `COM_PORT=COM6`).

However, in your specific environment you'll probably end up with the always same port designation, so it might be handy to choose this port designation as the makefile's default. Look for the initial assignment of macro *COM_PORT* in the makefile.

4.2.6 Weaknesses of the Makefile

Unsafe Recognition of Dependencies

The makefile includes the compiler generated *.d files. The *.d files contain makefile rules, which describe the dependencies of object files on source files. They are created as "side-effect" of the compilation of a source file. This means, they are not present at the very beginning and after a clean and they are invalid after source code or configuration changes which have an impact on the actual tree of nested include statements. Particularly in the latter situation the recognition of dependencies is not reliable and the compilation result could be bad. Moreover, if a header file is renamed or removed then it's problematic, that it is still referenced by the rules in the *.d files. Make aborts with a message like: "Don't know how to make ....h"

Consequently, in all cases of non-trivial changes of preprocessor macros and include statements or when renaming or removing header filer you should always call the rules to rebuild the application.

g++ versus gcc

The makefile compiles all source files regardless on their file name extension with g++, i.e. the source code is always treated as a C++ file. This is an a bit strange decision, which has been adopted from the original Arduino IDE, which behaves the same. There's no technical drawback of doing so, the generated machine code is just the same. The problem rather is that one easily and unintendedly writes C++ statements in a file he believes to be a C source files. The compiler won't complain about. RT*uin*OS

itself should be clean C source code and can be compiled with gcc also. (It's trivial to change the makefile to do so; there are anyway separate rules for *.c and *.cpp files.) However, when compiling *.c as C and *.cpp as C++ one has to properly use the *extern "C"* declaration, when accessing C headers from C++ files. This has not been done in the RT*uin*OS samples as we followed the Arduino style.[4]

4.3 Configuring the System Time

A central element of RT*uin*OS is its system time. This time is e.g. the parameter of a suspend command if a task wants to wait until a specific point in time or for a specific while. Many of the operations in RT*uin*OS and its application code deal with the system time. Therefore, we decided to make the implementation type of the system time subject to the configuration of the application; you have to customize the type in your application's copy of *rtos.config.h*. In many situations a short one Byte integer will be sufficient, but not in general. The intention of this section is to explain all implications of the type choice to enable you to choose the optimal, shortest possible type for your application.

4.3.1 The Unit of the System Time

The system time is a cyclic integer value. The unit is the period time of the main interrupt, which is associated with the system time. Each interrupt will clock the time by one unit. If the use case of RT*uin*OS is a traditional scheduling of regular tasks of different priorities it's good practice to choose the period time of the fastest regular task as unit of the system time. This yields the best performance, or the least system overhead respectively. But in general the unit of the system time doesn't matter with respect to the function of RT*uin*OS and the time even doesn't need to be regular.

All time related operations use the tic of the system time as unit. The delay time of *rtos_delay* can for example be specified only in multiples of this unit. The application required resolution of these timing conditions will determine the tic duration of the system time. (And your application will configure the driving interrupt source accordingly.) With respect to system overhead this resolution should be chosen as low as acceptable. Each timer interrupt causes at least one save and restore operation of the complete CPU context and this becomes a significant CPU load if the tic duration is about or even below 1 ms.

4.3.2 Recommended Timer Tic for regular Tasks

If applicable and if the application uses regular tasks we strongly recommend to choose the timer tic duration identical with the period time of the fasted task. Now each interrupt will really cause a task switch and there's no (useless) overhead just to clock the time. The timing resolution is identical to the task period. This means that the fastest task can barely operate with timeouts, for example if it needs to wait for events posted by other tasks. The only timeout condition it can use is to state the next regular due time. Nonetheless, reasonable error handling code is still possible as the task code can distinguish between becoming active because of the received event, the timeout or because it's its normal due time.

4.3.3 Upper Limit of the Task Period Time

The task period time of a regular task must not be greater than half the maximum of the system time. In practice, this is a limitation only if using the 8 Bit time. Now, the parameter of a call to *rtos_suspendTaskTillTime* must not exceed 127. The task overrun recognition can fail if this rule is disregarded.

A task overrun recognition is implemented in RT*uin*OS to support the use case of regular tasks. The system decides on a task overrun if a task referring to the absolute timer demands to be resumed in the past. The implementation uses signed operations, which undergo an arithmetic overflow at the middle of the binary range of the data type. This is where the limitation comes from.

[4]We guess that the Arduino designers decided to so because they wanted to disburden the users from understanding these kind of things and from using *extern "C"*.

Here's an example of the normal situation: The tic duration of the system time is 1 ms and a regular task of 100 ms period time is implemented. The system time has 8 Bit. An infinite loop is implemented, which uses `while(rtos_suspendTaskTillTime(100))` as always true condition. Given the task execution takes between 45 and 67 ms the RT*uin*OS code will safely find the next demanded due time between 55 and 33 ms in the future. It won't see a task overrun as 100-45 and 100-67 both is positive in the chosen 8 Bit signed arithmetics. Given a bad, too long task execution time of e.g. 102 ms RT*uin*OS would recognize a task overrun as 100-102 is negative.

Let's assume the period time of the same regular task would be changed to 180 ms. There's definitely no risk of getting into task overruns. However, at the end of the task execution, when the next resume time is demanded it depends if this demanded time is seen in the future or in the past. If the execution took 67 ms the remaining time to suspend the task is 180-67=113, which is a positive number, whereas 180-45=135 in case of the shorter task execution time is interpreted as the negative number -121 in the chosen 8 Bit signed arithmetics. For RT*uin*OS the demanded resume time is seemingly in the past and it decides on a task overrun.

Caution, the false decision on a task overrun, which actually isn't an overrun, is much worse than just an increment of the diagnostics counter (see *rtos_getTaskOverrunCounter*). If the demanded resume time of a task is found to be in the past this task is not suspended but made due immediately. Our example task would not be made due (and active) every 180 ms but irregularly depending on its actual execution time. This is the real fault!

To avoid this problem the period time of regular tasks must not exceed half the range of the chosen system timer. If a task period of 127 tics to too little you will have to choose the 16 Bit time, which produces of course more system overhead.

As a simple implication for systems applying regular tasks, the ratio of period times of slowest and fastest task is limited by half the range of the chosen system timer. If you implement regular tasks of e.g. 10 ms, 100 ms and 1000 ms period time, this could be handled with a *uint8_t* (ratio 1:100). If you want to have an additional 1 ms task, *uint8_t* will no longer suffice (ratio 1:1000), you need at least *uint16_t*. (*uint32_t* is probably never useful.)

4.3.4 Unrecognized Task Overruns

The shorter the chosen type of the system time the higher the probability of not recognizing task overruns when implementing regular tasks: Due to the cyclic character of the time definition a time in the past is seen as a time in the future if it is over more than half the maximum integer number. This leads to the wrong decision whether we have a task overrun or not. See the example:

Data type be *uint8_t*. A task is implemented as regular task of 100 units period time. Thus, at the end of the functional code it suspends itself with time increment 100 units. Let's say it had been resumed at time 123. In normal operation – no task overrun – it will end e.g. 87 tics later, i.e. at 210. The demanded next resume time is 123+100 = 223, which is seen as +13 in the future. If the task execution was too long and ended e.g. after 110 tics, the system time was 123+110 = 233. The demanded resume time 223 is seen in as 10 tics in the past and a task overrun is recognized.[5] A problem appears at excessive task overruns. If the execution had e.g. taken 230 tics the current time was 123 + 230 = 353 – or 97 due to its cyclic character. The demanded resume time 223 is 126 tics ahead, which is considered a future time – no task overrun is recognized.[6] The problem appears if the overrun lasts more than half the cycle time of the system time. With *uint16_t* this problem becomes negligible.

4.3.5 Summary

Here's a short summary of the system time data type discussion:

- Choose the data type of the system time as short as possible

[5] When a task overrun is recognized the task becomes due immediately. The task call is not omitted but made somewhat too late.

[6] One call of the task has been silently omitted and the next one is timely again.

- Choose the duration of the timer tic as long as acceptable. If applicable make it identical with the period time of the fastest task
- The duration of the timer tic is the resolution of timeout operations
- The task period time of a regular task must not exceed half the range of the chosen system time data type. False task overrun recognitions and resulting bad task timing would probably be the consequence
- The maximum reasonably expected time a task could exceed its nominal period time should be no more than half the range of the chosen system time data type. A task overrun would otherwise not be recognized as such

Choosing the type is done with macro *RTOS_DEFINE_TYPE_OF_SYSTEM_TIME*, please refer to [4].

4.4 Configuring the System Timer Interrupt

The interrupt service routine (ISR), which clocks the system time and which performs all related actions like resuming tasks, which are waiting for a timer event, is a core element of the implementation of RT*uin*OS. The implementation leaves it however open, which actual hardware event, i.e. which interrupt source, is associated with the service routine. In the standard configuration, the interrupt source is the overrun event of the timer 2 (*TIMER2_OVF*), but this can easily be changed by the application.

In the AVR environment, an ISR is implemented using the macro *ISR* as function prototype. A specific interrupt source is associated with the ISR by the macro's parameter. The name of the interrupt source is stated. A pre-defined, micro controller dependent list of available interrupt sources exists, please refer to [6], section 14. In RT*uin*OS the parameter of macro *ISR*, the name of the interrupt source, is implemented as other macro *RTOS_ISR_SYSTEM_TIMER_TIC*, which is defined in the application owned configuration file [4]. By simply changing the macro definition any other interrupt source can be chosen.

Typically, a few hardware related operations are needed to make a peripheral device a useful interrupt source. In case of timers, the timing conditions have to be stated (how often to see an interrupt); and generally, most peripherals require to set a so called interrupt mask bit in order to enable it as interrupt source.

The RT*uin*OS standard configuration uses timer 2 as is in the Arduino standard configuration. Arduino uses this timer for PWM output and has chosen appropriate settings. The only thing RT*uin*OS adds to the configuration of the timer is to set the interrupt mask bit of the overflow event. The frequency of the interrupt is not changed, the Arduino PWM functionality is not affected at all. The Arduino configuration causes an overflow event about every 2 ms.[7] This is thus the standard system clock of RT*uin*OS.

The hardware configuration of the interrupt source is done in the void function *rtos_enableIRQTimerTic*. The function is implemented as a *weak* function. In the terminology of the GNU compiler this means that the application may redefine the same function. Rather than getting a linker error message ("doubly defined symbol") the linker discards the RT*uin*OS implementation and will instead put the application's implementation in the executable code. The standard implementation is overridden by simply re-implementing the same function in the application code. Caution: The signature of the overriding function needs to be identical, the type attribute *weak* must however not be used again.

The application will put all operations to configure the interrupt source selected by macro *RTOS_ISR_SYSTEM_TIMER_TIC* into its implementation of the function and timer 2 will become like it used to be in the Arduino standard environment.

[7] The precise value can be found as a macro in the RT*uin*OS configuration file [4]. Changing the definition of this macro belongs to the code adaptations, which are required if the system timer interrupt is reconfigured.

Another, related code modification has to be made by the application programmer. The function pair *rtos_enter/leaveCriticalSection* inhibits and re-enables all those interrupts, which may lead to a task switch – which the timer interrupt evidently belongs to (see section 3.8). If you change the interrupt source, i.e. if you alter the value of macro *RTOS_ISR_SYSTEM_TIMER_TIC*, you will have to modify the code of these functions accordingly. They are implemented as macros in the application owned configuration file [4] and can thus be changed easily.

Please, find an example of a re-configured system timer as code sample tc05 in the RT*uin*OS distribution.

4.5 Using Application Interrupts

RT*uin*OS supports two application interrupts. The application configures the hardware interrupt source and associates it with the already existing interrupt service routine. The ISR itself must never be changed. It sets a specific pre-defined event, which is related to the application interrupt. Setting the event is done like any task could do using the API function *rtos_sendEvent*. The intended use case is that your application has a task which cyclically suspends itself waiting for this interrupt related event and which is hence awaken each time the interrupt occurs. This task is than the actual handler, which implements all required operations to do the data processing.

The application interrupt 0 is enabled in your application by turning the preprocessor switch *RTOS_USE_APPL_INTERRUPT_00* from *RTOS_FEATURE_OFF* to *RTOS_FEATURE_ON*. Now, the related ISR will be compiled and one of RT*uin*OS' general purpose events is renamed into *RTOS_EVT_ISR_USER_00* indicating the specific meaning this particular event gets; it should never be set by an ordinary task.

The existing ISR is associated with the interrupt source your application demands by means of macro *RTOS_ISR_USER_00*. The value of this macro is set to the name of the interrupt source. A table of available interrupt sources is found in the manual of your specific controller, see e.g. [6], section 14, for the micro controller of the Arduino Mega board.

The switch *RTOS_USE_APPL_INTERRUPT_00* and the macro *RTOS_ISR_USER_00* are found in the application owned configuration file [4].

The associated interrupt source needs to be configured to fire interrupt events. Most often, the interrupt sources are peripheral devices, which have some hardware registers which must be configured. For example, a regular timer interrupt would require to set the operation mode of the timer/counter device, the counting range and the condition, which triggers the interrupt. You will have to refer to your CPU manual to find out. All required settings to configure the interrupt are implemented in the callback function *rtos_enableIRQUser00*.

rtos_enableIRQUser00 does not have a default implementation, a linker error will occur if you do not implement it in your application code. Caution: It is invoked by the RT*uin*OS initialization code at a time when all tasks are already configured (*setup* has completed) and when the system timer of RT*uin*OS is already running. This means that all multi-tasking considerations already take effect. You need to anticipate task switches and resulting race conditions. Actually, the invocation of *rtos_enableIRQUser00* is done early from within the idle task, just before *loop* is executed the very first time.[8] Consider to use the function pair *rtos_enter/leaveCriticalSection* to sort out all possible problems.

When using application interrupts another, related code modification has to be made by the application programmer. The function pair *rtos_enter/leaveCriticalSection* inhibits and re-enables all those interrupts, which may lead to a task switch – which your interrupt evidently belongs to (see section 3.8). The functions need to additionally inhibit and re-enable the interrupt you choose as source. They are implemented as macros in the application owned configuration file [4] and can thus be changed easily.

[8] Take care designing your application: If you define a set of tasks, which do not leave any time for ideling your application interrupts won't be safely started. Take a set of always due round robin tasks as an example. Consider to start those tasks with a delay.

```
void rtos_enableIRQUser00(void)
{
    /* Inhibit all task-switch relevant interrupts. */
    rtos_enterCrtiticalSection();

    /* Configure the peripheral device to produce your application
       interrupt but do not enable the interrupt in its interrupt mask
       register yet. */
    ...

    /* Re-enable all task-switch relevant interrupts. Since you modified
       the implementation of rtos_leaveCrtiticalSection this will also
       set the appropriate bit in the mask register of your peripheral. */
    rtos_leaveCrtiticalSection();
}
```

Listing 4.3: Initialization of an application interrupt

Please consider that *rtos_leaveCriticalSection* partly implements what *rtos_enableIRQUser00* is expected to do, refer to listing 4.3 for more.

The second available application interrupt 1 is handled accordingly, you just have to replace the index 00 by 01 in all function and macro names referred to before.

4.6 Usage of Arduino Libraries

The biggest problem using RT*uin*OS is the lack of adequate multi-tasking libraries. It's not generally forbidden to still use the Arduino libraries but this introduces some risks.

For some reasons an existing library function can not be used just like that:

Particularly when addressing to hardware devices, strict timing conditions can exist. In an RTOS, when a task is suspend for an unpredicted while, this can make an operation timeout and fail. However, even in a single threaded system, the code execution speed is not fully predictable as it depends on compiler settings and interruptions by all the system interrupts. Particularly, if serving the hardware device is done in a task of high priority the discontinuity of the code execution should not be that much worse in comparison to a single tasking system that a timeout becomes a severe risk. Accordingly, time critical code should not be placed in the idle task.

Existing library functions could use static data, i.e. data which is not local to the invocation of the function. This can be data defined using the keyword `static` but also hardware registers, which do exist only in a single, global instance. In this case arbitrary invocation from different tasks will surely produce unpredictable results including the chance of a crash. Since most Arduino library functions deal with hardware entities, it's highly probable that they belong to this group.

Using the Arduino libraries is still possible if your tasks cooperatively share the global entities. If there's e.g. only one task which serves the PWM outputs and which lets other tasks access them only indirectly – via application owned, safely implemented inter-task communication.

A specific example is the global object *Serial*, which performs high level, stream based communication via USB. There's obviously no chance to access this channel in an uncontrolled, arbitrary fashion by several tasks. However, if only a single tasks does the console output at a time it works fine, even if this task is the most heavily interrupted idle task. By experience, *Serial* is quite tolerant against timing discontinuities of the invoking code and can be used for debugging purpose during application development.

Moreover, *Serial* is implemented as blocking function; the call of e.g. *println* returns only after all characters have been processed. Therefore the object can be successfully passed on from one task to

another after return from a *println*. Any task gets its turn to write a message. This can be done by a mutex or by non-preemptive, cooperative multitasking.

When RT*uin*OS' function *setup* is executed, no multi-tasking is active yet. Here you can use the Arduino libraries without more. In particular, you can initialize all required ports and devices and the communication with *Serial*.

It's similar for library *LiquidCrystal*. The implementation is basically thread-safe; the library can be used. Of course, the tasks must cooperatively implement a mutual exclusion when accessing the display. The source code contains some hardware-required delays, which are of course not implemented RTOS conform and which could be replaced by *rtos_delay* as an improvement of the library for RT*uin*OS.

The delay is implemented using Arduino's library function *delayMicroseconds*, which is based on a CPU consuming waiting loop. This does not necessarily mean that task switches are inhibited or delayed; just avoid to use the library in a task of high priority. Nonetheless, the CPU time is lost; these function calls raise the CPU load without need.[9]

Most of the significant, CPU consuming delays occur in the initialization of the display and this can be done in *setup* before multitasking actually takes place. The only other undesired delays are found in the control commands *clear* and *home* – if these are avoided or if their waste of CPU time causes no pain there's no concern to use *LiquidCrystal* from RT*uin*OS tasks.

Despite of all, using a library developed for single tasking in a multi-tasking environment remains a risk, which must not be taken in a production system. In a production system, any library function needs to be reviewed and maybe modified before using it. Fortunately, all Arduino code is available as source code, so that a code review can be done.

4.6.1 Changes of Arduino's *main* Function

The file main.cpp of the Arduino standard sketch has been replaced by main.c of RT*uin*OS. The implementation of the contained function *main* is nearly the same. The standard sketch finally enters a short, infinite loop, which repeatedly invokes the application's "task-function" *loop*. The same loop continues to exist in RT*uin*OS (although it has been moved to the end of *rtos_initRTOS*) with one major change: In the standard sketch the loop contains the following code, which is executed alternatingly with function *loop*:

```
if (serialEventRun) serialEventRun();
```

This function implements a kind of dispatcher for characters received via the serial channels. Clients can define their handlers by overriding "weak" empty default handlers. It depends on the optional handlers if this would continue to work well with the RT*uin*OS task scheduler. We decided not to put the call into the RT*uin*OS code. Whoever knows to need it can consider to place the statement into function *loop* of his RT*uin*OS application. Now, the behavior is just the same as in the standard sketch – besides all the obvious differences because of task switches.

4.7 Support of different Arduino Boards

RT*uin*OS has been developed on an Arduino Mega 2560 board and this is the only supported board so far. There are some obvious dependencies on the micro controller:

- The size of the program counter is three Byte for an ATmega2560 but only two Byte for some other derivates
- The availability of peripheral devices depends on the micro controller and moreover,
- the naming of the registers may differ between micro controllers even for the same peripheral.

[9] Arduino's delay function *delay* is different. It compares the desired return time with the global system time, which is updated by an interrupt. If a task using *delay* is interrupted by other tasks and if the time it gets re-activated again is ahead the desired return time it'll not continue to loop and consume the CPU like *delayMicroseconds* would.

The implementation of RT*uin*OS uses a preprocessor switch based on the macro *__AVR_ATmega2560__* from the AVR library anywhere we have such an obvious platform dependency. The else case is "implemented" as error directive so that you are directly pointed to all these code locations by simply doing a compilation with another micro controller selected in the makefile.

All code locations, where such an error directive is placed are easy and straight forward to modify. You will find some guidance in the code comments close to the error directives. Nonetheless, we decided to not try an implementation as it would be not tested.

Unfortunately, there's a remaining risk, that there are more platform dependencies than currently anticipated in the code. This can only be found out by doing the migration and testing. Feedback is welcome.

The makefile controlled build process depends on the Arduino board, too. An obvious change is the specification of the micro controller in use when compiling the code. However, there are more changes required in the makefile. Please refer to section 4.2.4 for more.

Chapter 5

Outlook

We hope that RT*uin*OS is deemed useful as it is and that it adds some value to the Arduino world. Nonetheless, we are aware that it has its limitations and that a lot of improvements are imaginable and some even feasible. Some ideas and comments have been collected here.

The lists which hold the due tasks are implemented as arrays of fixed length. All priority classes use a list of identical length. This has been decided just because of the limitations of constant compile time expressions and macros. The code would run without any change if the rectangular array holding all lists would be replaced by a linear array of pointers, which are initialized to point to class-individual linear arrays. This construction would save RAM space in applications which have priority classes of significantly differing size. Besides some saved RAM and the less transparent initialization on application side[1] there's absolutely no difference of both approaches and therefore the urgency of this change isn't considered high enough to actually do it.

Currently, the idle task is described by an additional object in the task array – although it lacks most of the properties of a true task. Actually, only the stack pointer save location and the (always zero) vector of received events are in use. If the task object is split into two such objects (holding the properties of all tasks and holding the properties of true tasks only) some currently wasted bytes of RAM could be saved.

The priority of a task could be switched at runtime if only the arrays are large enough – but this is anyway in the responsibility of the application. The implementation is simple as it is close to existing code. The API function would be implemented as software interrupt similar to the suspend commands. The active task would be taken out of its class list and put at the end of the targeted class list. The list lengths would be adapted accordingly. Then the normal step of looking for the now most due task and making this the active one would end the operation. *rtos_sendEvent* would probably be the best fitting starting point of the implementation. This idea has not been implemented as we don't see a use case for it.

Currently, the round robin time (including round robin mode on/off by setting the time to 0) is predetermined at compile time – but without any technical need. It would be easily possible to change it by API call at runtime. If we specify that a change shall not affect the running time slice it's very easy as the call of this function won't cause a task switch. A software interrupt is not required, just write the reload value of the round robin counter in an atomic operation.

There's no strong technical reason, why a task should not end. At the moment the return address of a task function is the reset address of the micro controller. By modifying *prepareTaskStack* it could become any other address, e.g. the address of a function implemented similar to the suspend commands. It would not put the active task into the list of suspended tasks but in a new list of terminated tasks. This list is required as task termination makes sense only if there's also a chance to create new ones. The list of terminated tasks would be the free-list of objects to reuse whenever a new task is created at

[1] We don't like to do dynamic initialization using a loop and a call of *malloc* inside in an embedded environment. This would probably consume administrative RAM space on the heap in the same magnitude than what can be saved by the changed layout of the RT*uin*OS data structure.

runtime.

Starting a new task at runtime would mean to let *prepareTaskStack* operate on a currently unused task object and to place the object in a critical section into the list of suspended tasks. As currently, the application is responsible for obeying the size of the lists, in particular if there's still room in the targeted priority class.

Since we do not want to introduce dynamic memory allocation into the application, any started task needs to be pre-configured. It has to be decided if under these circumstances terminating/restarting a task has a big advantage over just suspending existing tasks permanently.

Regardless of these and many other imaginable improvements, we consider RT*uin*OS to be a complete and useful real time operating system. Its limited capabilities are a well chosen trade off with the capabilities of the AVR target system which offers only little RAM space and no overwhelming CPU power. Most real time applications, that are feasible with this architecture should not be too demanding for our system and will surely benefit from being built on RT*uin*OS.

GNU Free Documentation License

Version 1.3, 3 November 2008

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section

does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "**Title Page**" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "**publisher**" means any person or entity that distributes copies of the Document to the public.

A section "**Entitled XYZ**" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "**Acknowledgements**", "**Dedications**", "**Endorsements**", or "**History**".) To "**Preserve the Title**" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front

cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See `http://www.gnu.org/copyleft/`.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.