Open Source RTOS

A comparison on features and design of some implementations

Sacchi Riccardo, (riccardo.sacchi@mail.polimi.it)
Matr. 760490
Trevisiol Ettore, (ettore.trevisiol@mail.polimi.it)
Matr. 770628

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: May 22, 2011

Abstract

Real-time systems play a considerable role in our society, and they cover a spectrum from the very simple to the very complex. Examples of current real-time systems include the control of domestic appliances like washing machines and televisions, the control of automobile engines, telecommunication switching systems, military command and control systems, industrial process control, flight control systems, and space shuttle and aircraft avionics. All of these involve gathering data from the environment, processing of gathered data, and providing timely response. A concept of time is the distinguishing issue between real-time and non-real-time systems. When a usual design goal for non-real-time systems is to maximize system's throughput, the goal for real-time system design is to guarantee, that all tasks are processed within a given time. The taxonomy of time introduces special aspects for real-time system research. Real-time operating systems are an integral part of real-time systems. Future systems will be much larger, more widely distributed, and will be expected to perform a constantly changing set of duties in dynamic environments. This also sets more requirements for future real-time operating systems. This report has the humble aim to convey the main ideas on Real Time System and Real Time Operating System design and implementation, and give a feedback about the most popular open source RTOS.

1 Introduction to RTS

Real Time is a term used in computer science to identify those applications where the correctness of the result depends on the response time, when "to respond an event" means to do it at a speed that is possible to predetermine. This implies that such programs must respond to "external" events within timescale. The concept of the time has real meaning, however, and is therefore used, even outside the computer science field. If all application are, strictly speaking, Real Time, however, because we expect a response within a finite time, the term is in practice used only for those applications whose issues are related to response times outweigh the complexity of the algorithm to use. Inside the issues to be addressed in real time we tend to distinguish the real-time applications "tight", in which the response time is typically expressed in milliseconds or in an fraction of a second and failures are catastrophic, from a more "slack" real-time, in which response times are normally expressed in seconds and failures are detected like a performance degradation.
The time itself is not only the major component for classifying a real-time system, in fact:

$$Correctness \Rightarrow Logic + Temporal \qquad (1)$$

To solve Real Time problems are usually used dedicated hardware architectures, human interfaces, operating systems and application programs designed specifically for monitoring and controlling an object (or more than one). These four components (hardware, operator, basic software, application software) are often closely linked, so as to achieve the required time optimization.

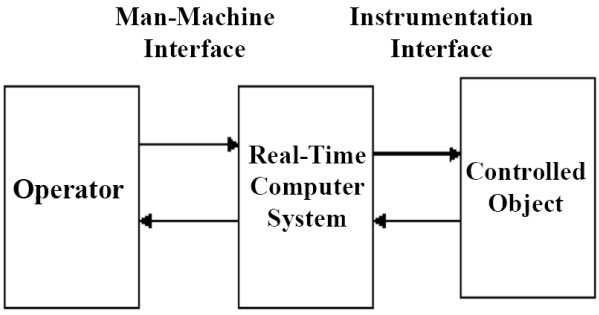

Figure 1: Real Time system interfaces [1]

Despite the initial differentiation, with the evolution of computer technologies, many hardware issues have gradually been developed with a software way, optimizing and finding better solution to solve problems.

1.1 Context

Usually the real time characteristics are required together with other special features. A requirement that appears frequently associated is that of operational continuity.
Some Real-time systems have to work continuously for long periods without any kind of interruption: the common case is the control of nuclear power stations or air traffic control. In other cases, however, the feature of Real Time is associated with embedded computers with the device that they should check. And this is for example the case of certain military applications, as in the artillery field. Continuing in the field of embedded architectures, there are many projects concerning Real Time application, particularly in the burgeoning field of wireless sensor networks.
In the world of wireless sensor networks, in fact, the time variable becomes crucial to ensure proper operation of the network. Generally, sensor nodes must be able to detect and process data from detector devices that were found in short time enough so as not to lose information. For example, the Prometheus project [2], by "Politecnico di Milano", aims to monitor the environment on the rock of the landslides S. Martino, in the province of Como, specifically nodes collect and process data from sensors at a frequency of 2 KHz. Among the others, an interesting project in this area is, for example, the "PerLa Project" [3] , that concerns the development of a high-level data manipulation and query language for wireless sensor networks. Aim of the project, developed by the "Politecnico di Milano" as well, is to extend the query language for databases with logical objects for other physical wireless devices, and to extend the language with instructions for physical actuators.
It's useful not to forget that, in a Real-Time System, the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced. In order to better imprint this concept, it is helpful to list some other examples of Real Time Systems: air traffic control systems, networked multimedia systems, command control systems, etc.

1.2 Problematics

A Real Time System changes its state as a function of a discrete (non-continuous) time, but, for example, a chemical reaction changes its state in a continuum domain of time, even after its controlling computer system has stopped. Easily a Real Time System should be a continuous function even if it is controlling a physical system like a chemical reaction. Thus, for example, a chemical reaction does no stop if the control system needs to reboot. In that moment, if the system does not work, the real time behavior is left and we have lost precious information.
On the basis of this, a Real Time System can be decomposed into a set of subsystems: the controlled object, the real time computer system and the human operator. A real time computer system must react to stimuli from the controlled object (or the operator) within time intervals dictated by its environment. The limit at which a result is produced is called a deadline. We have to divide three different cases: if the result is useful even after the deadline, the deadline is classified as *soft*, otherwise it is *firm*. Firm means that if any answer is provided before the deadline, there are no problems for the system. If a missing firm deadline might produce a catastrophe, such deadline must to be respected. In this case the deadline is called *hard* deadline and it must be respected.

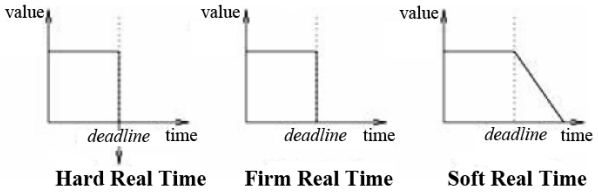

Figure 2: Real Time classes

If the firm aspect is rarely used (and also not fine defined), hard and soft system are very relevant, because they are the normal and common two uses of these systems. For example: commands and control systems, air traffic control systems are examples of hard Real Time Systems. On the other side, on-line transaction systems, airline reservation systems are soft Real Time Systems.
Scheduling is an other relevant problem that concerns the allocation of the resources in order to satisfy the timing constraints. It is divided in two classes: *static* and *dynamic*.

1.3 Soft and Hard systems

As already mentioned before, Real Time systems might be seen from different perspectives. The first classification, and the most important, consists in this type of division: *hard real-time* and *soft real-time*.

Figure 3: Real Time deadlines [4]

Main characteristics:

- The response time requirements of hard real-time systems are in the order of milliseconds or less and can result in a catastrophe if not met. In contrast, the response time requirements of soft real-time systems are higher and not very stringent.

- In a hard real-time system, the peak-load performance must be predictable and should not violate the predefined deadlines. In a soft real-time system, a degraded operation in a rarely occurring peak load can be tolerated.

- A hard real-time system must remain synchronous with the state of the environment in all cases. On the other hand soft real time systems will slow down their response time if the load is very high.

- Hard real-time systems are often safety critical and have small data files and real-time databases. Otherwise soft real-time systems, for example on-line reservation systems, have larger databases and require long-term integrity of real-time systems.

- If an error occurs in a soft real-time system, the computation is rolled back to a previously established checkpoint to initiate a recovery action. In hard real-time systems, roll-back/recovery is of limited use.

In synthesis:

Characteristics	Hard real-time	Soft real-time
Response time	Hard-required	Soft-desired
Peak-load performance	Predictable	Degraded
Safety	Often critical	Non-critical
Redundancy type	Active	Checkpoint-recovery
Data integrity	Short-term	Long-term
Error detection	Autonomous	User assisted

1.4 Schedulers

As written before, special emphasis is placed on hard and soft real-time systems. A missed deadline in hard real-time systems is catastrophic and in soft real-time systems it can lead to a significant loss.
It is right to say that a Real-time system must execute a set of concurrent real-time tasks in a such a way that all time-critical tasks meet their specified deadlines. Every task needs computational and data resources to complete the job. The scheduling problem is concerned with the allocation of the resources to satisfy the timing constraints.
In typical designs, a task has three states:

1. running (executing on the CPU),

2. ready (ready to be executed),

3. blocked (waiting for input/output).

The majority of the tasks are blocked or ready most of the time, because generally only one task can run at a time per CPU.

The number of items in the ready queue can vary, depending on the number of tasks the system needs to perform and on the type of scheduler the system uses. Predictability is the most important aspect in a Real Time System. In this context the scheduler helps the system in achieving the predictability for each deadlines of the tasks, but can also have other aims such as minimize response time or execution time, ensure the determinism of the system or maximize the total throughout. Real-time scheduling can be broadly classifies into two types: *static*, that makes scheduling decisions at compile time and is normally off-line, and *dynamic*, that is usually on-line and uses schedulability test to determine whether a set of tasks can meet their deadlines.

Static scheduling In static scheduling, the decisions are made at compile time. A run-time schedule is generated off-line, based on the prior knowledge of task-set parameters. So runtime overhead is small. This assumes parameters of all the tasks is known a priori and builds a schedule based on this. Once a schedule is made, it cannot be modified online. There are no explicit static scheduling techniques except that a schedule is made to meet the deadline of the given application under known system configuration. Most often there is no notion of priority in static scheduling. Based on task arrival pattern a time line is built and embedded into the program and no changes in schedules are possible during execution.

Dynamic scheduling On the other hand, dynamic scheduling makes its scheduling decisions at run time, selecting one out of the current set of ready tasks. Dynamic schedulers are flexible and adaptive. They can incur significant overheads because of run-time processing. Typical simple scheduling algorithms are: FIFO (simple first-in-first-out queue), Round Robin (circular queue), Earliest Deadline First (EDF) Algorithm, Shortest Time (the next task is that one which is rated to be the first to finish).
The scheduler looks very interesting to allow the virtual parallelism (or logical). It is possible with the allocation over time of the (scarce) resources to the ongoing actions during intervals of time inside their respective time purposes (multitasking). This parallelism is not closely linked to the use of multiple processors, but rather to make best use of a single processor. It should be noted that, if not exploited well, the use of multiple processors does not necessarily improves the timing characteristics of which requires a real-time system. Returning to consider the case of a system with a single processor. As long as the ratio of CPU usage is less than 100% and the scheduling time is likely to contain the execution of any action within its temporal order, the overall effect observed is "equivalent" to those obtained from other processors in parallel mode.

The scheduler might be implemented using one of the following virtual parallelisms:

Time-Driven The functionality of an automation system can be decomposed into a set of activities competing over time (which operate in parallel). Each activity can be seen as a transformational process that receives incoming information from the external world and (ideally continuously) produces output information and commands to give out to the external world.

Event-Driven Actions are performed by each process (task or thread) that is activated by external events or time events. Since the events associated information (data), this approach can also call data-driven. Synchronizations with the events are typically based on the mechanism of the interrupt.

Moreover, for the scheduler, we can talk a little about *preemptive* or *non-preemptive* scheduling of tasks. One, of both methods, is possible with static and dynamic scheduling. In preemptive scheduling, the currently executing task should be interrupted upon arrival of a higher priority task. In non-preemptive scheduling, the currently executing task will not be preempted until completion. Usually the data structure of the "ready list", in the typical design of a scheduler, is designed to *minimize the worst-case length of time* spent in the scheduler's critical section, during which preemption is inhibited, and, in some cases, all interrupts are disabled.

1.5 Further characteristics

Every Real Time system has to control and manage its behavior with the proper scheduler and following its application logic in the context of use. For this reason, and for the deterministic aspect that are very important for calculate the maximum time for each task, it is also important to take care about the fault tolerance. Fault tolerance features basic allow the computer to keep executing even with the presence of defects. These systems are usually classified as either highly reliable or highly available.

Reliability As a function of time, is the conditional probability that the system has survived the interval [0,t], given that it was operational at time t=0. Highly reliable systems are used in situations in which repair cannot take place (e.g. spacecraft) or in which the computer is performing a critical function for which even the small amount of time lost due to repairs cannot be tolerated like in Real Time system (e.g. flight-control computers). MTTF = mean time to failure.

Availability Is the intuitive sense of reliability. A system is available if it is able to perform its intended function at the moment the function is required. Formally, the availability of a system as a function of time is the probability that the system is operational at the instant of time, t. If the limit of this function exists as t goes to infinity, it expresses the expected fraction of time that the system is available to perform useful computations.

$$A = \frac{MTTF}{MTTF + MTTR} \tag{2}$$

Where MTTR (Mean Time To Repair in the case one single operation faults). For a software, is the time needed for restart the application where there was the fault. If we want to maximize the Availability, we need to use high reliable component (high MTTF value) and reduce the MTTR.

1.6 Conclusions

As noted, there are many different characteristics that describe a real-time system. The only, and trivial, definition of "must respond as quickly as possible" is not sufficient for certain applications. Define hierarchical classes such as Hard and Soft Real Time allows us to differentiate the concept of "miss deadline". It is also absolutely correct the management of the determinism of a set of answers, or a series of errors, which does nothing that improve the definition and characteristics of a system of this type. Finally we must underline how plays a major role the ability to handle multiple processes simultaneously with an appropriate scheduler, being still able to keep valid all time purposes.

2 Hardware platform for the Real Time

Hardware architecture is the set of design criteria according to which is designed and built a computer, or a device that is part of it.

2.1 Context

Approaching the topic in a very large way, it's really easy to say that anything can be real time if we operate in Real Time. A simple notebook can be real time if it is mounted on a system that supports the Real Time Paradigm. It must be said however that there are some areas of Real Time Systems where we use specific hardware and we always work in real time. This is the case of embedded devices such as: ATMEGA, TMOD, DSPIC, CORTEX, mainly used in the field of wireless sensor networks, where "embedded system" means a system with specialized hardware and software, dedicated to a single application context and immersed in its operating environment.

Otherwise if we want to get higher performance we must use dedicated hardware unit like programmable logic units such as ASIC, FPGA, SoC, or even dedicated systems for that specific purpose [5]. Of course it is obvious that the cost becomes much higher when we go more in the specific and we lose of generality. When designing embedded systems, timing is critical; whether it seeks to ensure that a control loop is implemented in a reliable manner, or we respond quickly to a signal I / O or the processing is fast enough to meet the application requirements. The construction of a system to handle these needs of timing starts from hardware; it must provide the right balance between size, power consumption, computational throughput and latency.
In addition, the software plays a key role in the regulation of execution, in response to interrupts and to balance the time between the computing tasks. Integrating hardware and software, taking into account the timing, can be a laborious process. As embedded systems are specific hardware and software cases applied to a single problem, to give less abstract to the topic, follow the descriptions of programmable devices most commonly used:

ASIC "Application Specific Integrated Circuit" refers to the area application of the product. ASIC is a integrated chip specifically designed to solve a precise application of calculation. The specificity of the design, focused on solving a single problem, can reach the performance which is hardly to obtain with the use of more generic solutions. It has high cost of licenses and initial production rates, and long development time. For these economic and temporal disadvantages, for common situations, it prefers to use FPGA components.

FPGA "Field Programmable Gate Array" refers to digital devices which can be programmed by the user in the "field", or "out of the factory." These are elements which have characteristics intermediate between the ASIC on one side and with PAL architecture (Programmable Logic Array) on the other. The use of these components has certain advantages over ASICs, not only in economic but also in the usage because they are simple to program.

SoC The term *System on a Chip* (often abbreviated *SoC*) is used to indicate these particular integrated circuit chip which containing an entire system in a single chip, or rather, in addition to the central processor, also incorporate a chipset and possibly other controllers such as the RAM memory, the circuitry input / output or the video subsystem. A single chip can contain digital components, analog and radio-frequency circuits in a single unit. Even thought this type of integrator is used commonly in embedded applications given the small size that they reach through the integration of all components, it is to be considered leased circuits that can be integrated in more complex situations than the embedded.

2.2 Technologies

Whatever is the specific architecture used, it is important to determine whether the information to be acquired are associated with "states" or "events", regardless of whether it is more convenient to get states or events, and then focus on hardware synchronization.

It's important to note that:

- by sampling (SW) is acquired states, not events.
- by interrupt (HW) is acquired events, but not states.
- by technical HW or SW can be: reconstructed the states by the events acquired and got events by states acquired.

Synchronization is based on three main components: a monitoring of program (*polling*), *interrupts* and Direct Memory Access (*DMA*). We must distinguish between synchronization (1^{st} level) and the operation of transferring information between computer and the external world (or vice versa). The producer-consumer relationship between processes (SW) inside the computer, requires a synchronization at 2^{nd} level.

Mechanism	Sync 1^{st}level	Transfer	Sync 2^{nd}level
Control program	SW	SW	SW
Interrupt	HW	SW	SW
DMA	HW	HW	Interrupt

Bearing in mind that:
SW: expensive to develop, cheap to reproduce flexible and articulated processing but slow.
HW: cost of implementation and testing of each unit fast, execute simple and rigid calculations.

Monitoring of Program (polling) Periodic sampling (or with waiting loop) of a state (indicator event occurring). The cause-effect chain, which is realized by this mechanism is:

- HW - the external event takes dedicated circuits in a certain state.
- SW - the program executes instructions to read the state.
- SW - the program relies on the state-occurrence of the check of an event or not.
- SW - the program execute the action (e.g. value reading).
- SW - the program execute the calculations of the value.

Management interruption It is associated with a channel of interrupt to any flow of events to be transposed. Interface circuits are provided for event management and their transformation into interrupt requests. Typically, processors allow us to enable or disable (*mask*) selectively the individual channels of interrupt. Many processors have a special non-maskable interrupt (*NMI = Not Maskable Interrupt*) for events to which it is essential to minimize the latency. The usage of this interrupt requires special attention. The priorities are intended to resolve conflicts of contemporary interrupt requests from different streams of events. Often the priorities of the various channels of interrupts are pursued, but in processors there is usually a control that allow their own reconfiguration.
The change of context is of paramount importance in these systems. Processors designed for real-time applications often have different dedicated registers for different levels of interrupt, with automatic switching at the time of interruption, in order to reduce the overhead of saving the context.

Direct Memory Access (DMA) allows us to transfer blocks of information at very short response time, high transfer rate and high efficiency but without elaboration (which are postponed to the end of the block transfer events/information). The cause-effect chain that is made is as follows.

- CPU, running some part of software, sets the parameters in the controller registers, enabling the transfer of the next block:
  - The DMA controller transposes an event.
  - The DMA controller assumes the role of bus master computer.
  - The DMA controller controls the operations of reading of the entrance door and writing in the appropriate memory address (or vice versa).
  - The DMA controller increments the memory address and transfers the counter.
- When the count reached (terminal count), the controller generates an interrupt request (2^{nd} level synchronized).

2.3 Conclusions

If we do not need embedded architecture to maximize the performance of the device, and then the entire real-time system, we can use type architectures like ASIC or FPGA. In fact, ASIC is a chip designed to perform a certain functionality. Being an hardware totally dedicated (or partially) allows to obtain benefits not available otherwise, but the costs are prohibitive and justifiable only for the production of a large number of pieces. In recent years, FPGA, with their performance and increase their cost vastly more affordable, have gained much of the ASIC market.

Whatever hardware devices we use, however, it's always necessary to keep in mind the aspect of synchronization that is required to develop a good solution for real-time in this devices.

3 Software platform for the Real Time

Software architecture is the set of implementation criteria with which an integrated software framework is designed and implemented; main issues are also the respect of all basic characteristics of each computer and their configurations, necessary for the proper interoperability. Rather than being faster than others, a real-time system is able to provide and guarantee the correct execution in time of a transaction. The system can then be designed with the guarantee that a program written in a language will react within a certain period of time.

3.1 Real Time Operating Systems

The Operating System is responsible for ensuring the link between user, software resources and applications. So when a program wants to access to an hardware resource, there is no need to send specific information to the device, but simply send the information to the operating system, which will transmit them to the device concerned by its driver. The operating system then allows for "decoupling" the programs and hardware, especially to simplify the management of resources and provide the user with a human-terminal interface simplified to allow him to get rid of the complexity of the terminal hardware.

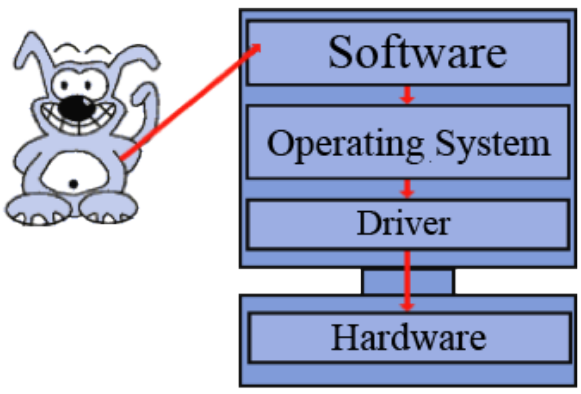

Figure 4: Real Time System layers [6]

The real-time systems, mainly used in industry, are systems whose objective is to work in an environment with a time limit. A real-time system must operate reliably following specific time limits, thus it must be able to deliver the proper treatment of information received at defined time intervals (regular or not).

Here there are some examples of real-time operating systems:

- FreeRTOS
- RTLinux
- ChibiOS/RT

3.2 Principal typologies

In practice, a real-time system must ensure that a process (or task) ends within a given time constraint or deadline. This is required to ensure that the scheduling of operations is feasible, and that the operating system facilitates these tasks. The products of Windows and Unix families do not meet the typical characteristics of a real-time: as example, while managing the execution of multiple processes with preempting, is not possible in any way to predict what will be the time performance of a single process.
Also the use of hard disk for data storage, USB or other devices that introduce high latencies of execution by the CPU, makes it impossible to determine with certainty how long it will take to retrieve the information needed for the proper execution of the code. Also Windows CE operating system is wrongly considered by someone like a real-time operating system, when in fact there is no privilege in hardware and software architecture that will enable us to pre-establish the running time of a series of operations of this OS. There are several factors that cause unpredictability in the response of the operating system. Among them, the main ones are as follows:

- *DMA*: can steal the bus to the CPU by delaying the execution of a critical task. In a real-time system we prefer and then turn it off or use it in time slice mode where we assign steadily and set the bus to the DMA even if there are operations to be done.
- *Cache*: can cause unpredictability because there are cases where it fails and may cause delays in memory access by the CPU. When considering the worst case then we would prefer not to use it at all.
- *Mechanisms of memory management*: these techniques do not introduce unpredictable delays during the execution of critical tasks, such as paging, page faults can cause intolerable delaying for a hard real-time. Typically we use the segmentation or the partition of the static memory.
- *Interrupts*: are generated by peripheral devices when they have some information to be exchanged with the CPU. These interruptions, during the execution of a critical task, generate delaying unpredictable and then we prefer to turn them off.
- *Power management systems*: they are hardware mechanisms that can slow down the CPU or to enforce its useful code to dissipate less energy.

  It is clear that in a real-time system is important not to break a deadline rather than consume little energy, and these mechanisms are disabled.

3.3 Characteristics

A real-time system should have (at least) the following characteristics:

- *Excellent Scheduling*: all tasks are known a priori as well as time constraints, it should be possible therefore to have a scheduler that implements a schedule that minimizes the cost time function of the sum of all tasks.
- *Determinism and Responsiveness*: operations are performed at fixed, predetermined times or within predetermined time intervals and, particularly, the system takes to service the interrupt in a reasonable and useful limit of time.
- *Resource sharing*: tasks are separate entities but which contribute to the same purpose, so it is not necessary to have separate address spaces.
- *Performance guarantee*: every hard real-time tasks must end before their deadline, then, if they have a new task or a task cannot be completed in its deadline, an advance notification of the system can be used to prevent the execution of the new task, or to recover the execution of the task that is about to break.
- *Predictability of system calls*: the system should be able to evaluate the computation time of each task to determine a feasible schedule, so every call to the system must have a well defined maxi- mum run time so as not to introduce indefinite delays.
- *Reliability*: the system have to be stable and response to the events non only in time, but also (theoretically) every time with no failure.
- *User control*: much more important than in normal OS, user should be able to specify paging and process swapping, decide which processes must remain in the main memory, establish the rights for all tasks, control priority of processes, control interrupt and manage external devices...

3.4 Features

In order to ensure all the characteristics shown above, the RTOS exploits some typical features found in quite every system for real-time. Typically these systems can manage at least 20 tasks, have characteristics for minimize interrupt and context switch latency, allow the total management of creation and modification of processes and tasks and hold mechanisms for manage time delays and timeouts.
We have already talked about the scheduling algorithms;

typically a RTOS use multiple round-robin queue with different priorities that can be changed dynamically and have the preemption component, regulated by a simple time-slice. But a large number of these systems can also use a particular real-time algorithm. The main aspect that makes the difference is the behavior when a real-time (very high priority) task or event arrives. If there is only a simple queue the task must wait for all the tasks queued before its. If the scheduler use different priorities, the task has to wait (if they exist in the relative queue) only for the other tasks with the same or more priority; in addiction, if the scheduler use preemption, the task could be executed also if other previous tasks haven't already completed their computation. In the end, some system can provide an "immediate" scheduler, that can start executing very high priority tasks (quite) in the moment they arrive.

A particular mention should be done about the kernel. Typically RTOS's kernels are essential and very small, so to not overcome the CPU with useless work. In main systems the kernel create only main process for manage memory, tasks and primary system service; all the remaining features are placed only if used.
The main issue related to the kernel is the capability of pause and resume its execution; some systems deny this possibility, but other have some features to ensure that it is possible safety. For example they use semaphores (that unable of disable the permission of write in particular memory locations) or the setting of critical zones, where the execution mustn't be interrupted.

Other important feature are the use of virtual memory, the capacity of have mechanism for the intertask communications and the contemporary use of static and dynamic memory (heap) and its protection and sharing, but these components very depend from the system considered, so can be very different.

Finally we should talk about a common issue with real-time systems, the priority inversion. The context in quite common: a high priority task ask for a resource that is used at the moment by one (or more) low-priority task. Because of the waiting for the release of the resource, the task became automatically a low-priority task itself. To avoid this problem many systems have the capability the recognize this situation and resolve it.

3.5 Conclusions

The software is very important factor for the development of a device or an application that operates in real time. The operating system is the first component that is considered for not to run into insurmountable problems of impracticably of the project. It does not necessarily have to be fast: it does not matter the time interval in which the operating system/application should react; the important thing is to respond within a maximum pre-determined time.
In other words, the system must be predictable. A programming language is also a predominant component and should be designed for the development of applications in which the time factor is critical. Real Time-oriented languages are designed so to give guarantees on the maximum execution time of an operation (worst case), designing a system so as to be sure that the generic software program will respond within a period of time. In fact, for real time languages, there is no real category only designed for this problem. We can say that all are very general and adaptable. We should keep in mind that the performance of a Real Time language is always linked to the operating system on which it operates. In the case of the Ada is the compiler to handle the operating system primitive efficiently. Platforms dedicated to these applications are useful to facilitate the programmer, and databases ad hoc are required to avoid wasting time on waiting for answers to critical queries.

4 Open source RTOS

Now we will analyze and describe some of the most active and popular open source RTOS. We will mainly focus on the aspects regarding the kernel, the memory management, the scheduler, the license and we will give some feedback about the documentation we have found and how active and good is the community. Finally we will analyze how is possible to start to develop an application related to the OS or how we can port the RTOS to a new architecture.

4.1 FreeRTOS

FreeRTOS [7] is a scalable real time kernel designed specifically for small embedded systems. Some of its highlights include the official support for 23 architectures, a powerful execution trace functionality, stack overflow detection options and a lot of free development tools.

4.1.1 Supported architectures

ARM, AVR, AVR32, Freescale Coldfire, HCS12, IA32, MicroBlaze, MSP430, PIC, Renesas H8/S, 8052, STM32.

4.1.2 Development

A goal of FreeRTOS is that is simple and easy to understand. For this reason, the majority of the RTOS source code is written in C, not assembler. There is a Windows AVR cross compiler based on GCC, and its WinAVR development tools. On the official website there are good tools for the software development and OS understanding, like Windows or Linux simulator and lots of loadable demos.

4.1.3 License

FreeRTOS is licensed under a modified GNU GPL and can be used in commercial applications under this license. An alternative commercial license option is also available in cases that:

- We cannot fulfill the requirements stated in the "Modified GPL license" column of the table below.
- We wish to receive direct technical support.
- We wish to have assistance with our development.
- We require legal protection or other assurances.

License feature comparison [8]:

	FreeRTOS	OpenRTOS
Is it free?	√	×
Can I use it in a commercial application?	√	√
Do I have to open source my application code that makes use of the FreeRTOS services?	×	√
Do I have to open source my changes to the kernel?	√	×
Do I have to document that my product uses FreeRTOS?	√	×
Do I have to offer to provide the FreeRTOS code to users of my applications?	√	×
Can I receive professional technical support on a commercial basis?	×	√
Is a warranty provided?	×	√
Is legal protection provided?	×	√

OpenRTOS Is a commercially licensed and supported version of FreeRTOS that includes fully featured professional grade USB, file system and TCP/IP components.

SafeRTOS Is a SIL3 RTOS version that has been certified for use in safety critical applications. It is a functionally similar product for which complete IEC 61508 compliant development/safety lifecycle documentation is available (conformance certified by TUV SUD, including compiler verification evidence).

4.1.4 Kernel

The FreeRTOS real time kernel is very small (typically its binary image is in the region of 4K to 9K bytes). Talking about its functionality, it measures time using a *tick* count variable. A timer interrupt (the RTOS *tick interrupt*) increments the tick count with strict temporal accuracy, allowing the real time kernel to measure time to a resolution of the chosen timer interrupt frequency. Each time the tick count is incremented the real time kernel must check to see if it's time to unblock or wake a task. It is possible that a task woken or unblocked during the tick ISR will have a priority higher than that of the interrupted task. If this is the case the tick ISR should return to the newly woken/unblocked task, effectively interrupting one task but returning to another. This is depicted below:

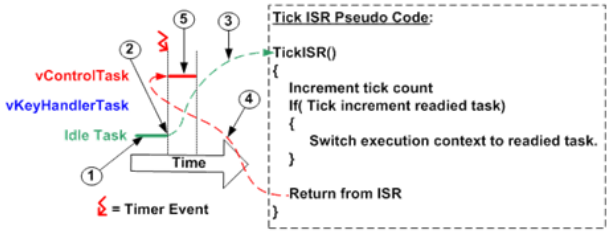

Figure 5: FreeRTOS tick interrupt [9]

A context switch occurring in this way is said to be *preemptive*, as the interrupted task is preempted without suspending itself voluntarily.

The scheduler is very simple and does not have a lot of features, but the basic ones should be enough for major part of the applications. The simple algorithm makes it fairly easy to determine the system behavior. The scheduler consist of a queue with threads of different priority. The threads are scheduled to the system after their priority, where higher priority get scheduled first. Threads in the queue that share the same priority will share the CPU with the round robin timeslicing. Preemption is implemented in this scheduler.

Idle Hook Function The idle task can optionally call an application defined hook (or callback) function, the idle hook. The idle task runs at the very lowest priority, so such an idle hook function will only get executed when there are no tasks of higher priority that are able to run. This makes the idle hook function an ideal place to put the processor into a low power state, providing an automatic power saving whenever there is no processing to be performed. The idle hook will only get called if `configUSE_IDLE_HOOK` is set to 1 within FreeRTOSConfig.h. When this is set the application must provide the hook function with the following prototype:

```
void vApplicationIdleHook( void );
```

The idle hook is called repeatedly as long as the idle task is running. It is paramount that the idle hook function does not call any API functions that could cause it to block. Also, if the application makes use of the *vTaskDelete()* API function then the idle task hook must be allowed to periodically return (this is because the idle task is responsible for cleaning up the resources that were allocated by the kernel to the task that has been deleted).

4.1.5 Statistics

FreeRTOS can optionally collect information on the amount of processing time that has been used by each task. The *vTaskGetRunTimeStats()* API function can then be used to present this information in a tabular format.

4.1.6 Memory

The RTOS kernel has to allocate RAM each time a task, queue or semaphore is created. The malloc() and free() functions can sometimes be used for this purpose, but:

- they are not always available on embedded systems
- take up valuable code space
- are not thread safe
- are not deterministic (the amount of time taken to execute the function will differ from call to call)

One embedded real time system can have very different RAM and timing requirements to another, so a single RAM allocation algorithm will only ever be appropriate for a subset of applications. To get around this problem the memory allocation API is included in the RTOS portable layer, where an application specific implementation appropriate for the real time system being developed can be provided. When the real time kernel requires RAM, instead of calling malloc() it makes a call to *pvPortMalloc()*. When RAM is being freed, instead of calling free() the real time kernel makes a call to *vPortFree()*.
Using a *Memory Protection Unit* (MPU) can protect applications from a number of potential errors, ranging from undetected programming errors to errors introduced by system or hardware failures. FreeRTOS-MPU can be used to protect the kernel itself from invalid execution by tasks and protect data from corruption. It can also protect system peripherals from unintended modification by tasks and guarantee the detection of task stack overflows. Each task maintains its own stack. The memory used by the task stack is allocated automatically when the task is created, and dimensioned by a parameter passed to the *xTaskCreate()* API function. Stack overflow is a very common cause of application instability. FreeRTOS therefore provides two optional mechanisms that can be used to assist in the detection and correction of just such an occurrence. The option used is configured using the `configCHECK_FOR_STACK_OVERFLOW` configuration constant. Note that these options are only available on architectures where the memory map is not segmented. Also, some processors could generate a fault or exception in response to a stack corruption before the kernel overflow check can occur. The application must provide a stack overflow hook function if `configCHECK_FOR_STACK_OVERFLOW` is not set to 0. The hook function must be called *vApplicationStackOverflowHook()*, and have the prototype below:

```
void vApplicationStackOverflowHook(
xTaskHandle *pxTask, signed portCHAR
*pcTaskName );
```

The `pxTask` and `pcTaskName` parameters pass to the hook function the handle and name of the offending task respectively. Note however, depending on the severity of the overflow, these parameters could themselves be corrupted, in which case the `pxCurrentTCB` variable can be inspected directly. Stack overflow checking introduces a context switch overhead so its use is only recommended during the development or testing phases.

4.1.7 Other features

The RTOS includes also other important real-time characteristics such as possibility of a (quite) full customization through the modification of a lot of parameters in the FreeRTOS Config.h file, support of both tasks and co-routines with no restriction of the number of tasks created and the priorities to be assigned to them and lots of feature for communication and synchronization (queues, semaphores, mutexes...).

4.1.8 Documentation and community

The FreeRTOS documentation is largely available on the official website in both, high and low level description. API are also described in detail about every function and with good example to understand their own behavior. And good guide are linked to install, try or load demos on lot of commercial boards. Lastly a community [10] can support the developers that use FreeRTOS, it's a good and active community, much more active than other RTOS that are analyzed hereafter.

4.1.9 Basic directory structure

The FreeRTOS download includes source code for every processor port, and every demo application. Placing all the ports in a single download greatly simplifies distribution. The directory structure is however very simple, and the FreeRTOS real time kernel is contained in just 3 files (4 if co-routines are used).

From the top, the download is split into two sub directories:

```
FreeRTOS
├── Demo
└── Source
```

Demo contains the demo application for FreeRTOS, and the source folder contains the real time kernel soruce code. Let's analyze the Source dir. The majority of the real time kernel code is contained in three files that are common to every processor architecture. These files, *tasks.c*, *queue.c* and *list.c*, are in the source directory. *croutine.c* implements the optional co-routine functionality, which is normally used on very memory limited systems. The Portable directory contains the files that are specific to a particular microcontroller and/or compiler. The Include directory contains the real time kernel header files. To use FreeRTOS we need to include the real time kernel source files in our makefile. It is not necessary to modify them or understand their implementation. Reassuming, all the common files can be found in the Source directory. The port specific files can be found in subdirectories contained in the Portable directory.
For example, if we are the MSP430 port with the GCC compiler we use the common tasks.c, queue.c and list.c located in the Source directory, the MSP430 specific file (port.c) can be found in the Portable/GCC/MSP430F449 directory, and all the other sub directories in the Portable directory relate to other microcontroller ports and can be ignored.

```
Source
├── include
├── portable
│   ├── MemMang
│   └── ...
├── croutine.c
├── list.c
├── queue.c
└── tasks.c
```

On FreeRTOS webpages [11] a complete list of all file included in the Portable directory can be found. The download also contains a demo application for every processor architecture and compiler port. The majority of the demo application code is common to all ports and is contained in a directory called Common, under the Demo directory. The remaining sub directories under Demo contain a project or a makefile ready to create files for building the demo for that particular port. If building the MSP430 GCC demo application makefile can be found in the Demo/MSP430 directory. All the other sub directories contained in the Demo directory (other than the Common directory) relate to demo application's targeted at other microcontrollers and can be ignored.

```
Demo
├── Common
│   ├── Minimal
│   ├── Full
│   └── include
└── ...
```

4.1.10 Create application

When writing our own application it is preferable to use the demo application makefile (or project file) as a starting point. We can leave all the files included from the Source directory included in the makefile, and replace the files included from the Demo directory with those for our own application. This will ensure both the RTOS source files included in the makefile and the compiler switches used in the makefile are both correct. Each demo application creates a set of demo real time tasks and/or co-routines - most of which are not specific to any one demo but common to many. These tasks are created within main(), which in turn is defined within main.c. For example, the main() function for the Luminary Micro LM3S811 GCC demo is contained within `FreeRTOS/Demo/CORTEX_LM3S811_GCC/main.c`. Most demos applications also create a "check" task in one form or another. The "check" task will execute infrequently (typically every 3 or 5 seconds) but has a high priority so is guaranteed to get processor time. Its primary responsibility is to check that all the other tasks are still operating as expected, and that no errors have been detected. The check task will report the system status either on an LCD (if present) or by toggling an LED.
A typical main() function will have the following structure:

```
int main( void ) {
 /* Setup the microcontroller hardware
 for the demo. */
 prvSetupHardware();

 /* Create the common demo application
 tasks, for example: */
 vCreateFlashTasks();
 vCreatePollQTasks();
 vCreateComTestTasks();

 Etc.

 /* Create any tasks defined within
 main.c itself, or otherwise specific
 to the demo being built. */

 xTaskCreate( vCheckTask, "check",
 STACK_SIZE, NULL, TASK_PRIORITY,
 NULL );

 Etc.
```

```
 /*Start the scheduler, this function
 should not return as it causes the
 execution context to change from
 main() to one of the created
 tasks. */

 vTaskStartScheduler();

 /* Should never get here! */

 return 0;}
```

For example in the CORTEX demo we find in the source code (it's an extract with the most important features): this project contains an application demonstrating the use of the FreeRTOS.org mini real time scheduler on the Luminary Micro LM3S811 Eval board.

main() simply sets up the hardware, creates all the demo application tasks, then starts the scheduler.

```
int main( void )
{
  /* Configure the clocks, UART and
  GPIO. */
  prvSetupHardware();

  /* Create the semaphore used to
  wake the button handler task from
  the GPIO ISR. */

  vSemaphoreCreateBinary(
  xButtonSemaphore );
  xSemaphoreTake(
  xButtonSemaphore, 0 );

  /* Create the queue used to pass
   message to vPrintTask. */
  xPrintQueue = xQueueCreate(
  mainQUEUE_SIZE, sizeof(
  portCHAR * ) );

  /* Start the standard demo
  tasks. */
  vStartIntegerMathTasks(
  tskIDLE_PRIORITY );
  vStartPolledQueueTasks(
  mainQUEUE_POLL_PRIORITY );
  vStartSemaphoreTasks(
  mainSEM_TEST_PRIORITY );
  vStartBlockingQueueTasks(
  mainBLOCK_Q_PRIORITY );

  /* Start the tasks defined
  within the file. */
  xTaskCreate( vCheckTask, "Check",
  configMINIMAL_STACK_SIZE, NULL,
  mainCHECK_TASK_PRIORITY, NULL );
  xTaskCreate( vButtonHandlerTask,
  "Status", configMINIMAL_STACK_SIZE,
  NULL, mainCHECK_TASK_PRIORITY + 1,
  NULL );
  xTaskCreate( vPrintTask, "Print",
  configMINIMAL_STACK_SIZE, NULL,
  mainCHECK_TASK_PRIORITY - 1,
  NULL );

  /* Start the scheduler. */
  vTaskStartScheduler();

  return 0;
}
```

In addition to a subset of the standard demo application tasks, main.c also defines the following tasks.

A "Print" task. The print task is the only task permitted to access the LCD, thus ensuring mutual exclusion and consistent access to the resource. Other tasks do not access the LCD directly, but instead send the text they wish to display to the print task. The print task spends most of its time blocked, only waking when a message is queued for display.

```
static void vPrintTask(
void *pvParameters )
{

/*declaration*/

 for( ;; )
  {
  /* Wait for a message to arrive. */
  xQueueReceive( xPrintQueue,
  &pcMessage, portMAX_DELAY );

  /* Write the message to the LCD. */
  uxRow++;
  uxLine++;
  OSRAMClear();
  OSRAMStringDraw( pcMessage, uxLine
  & 0x3f, uxRow & 0x01);
  }
}
```

A "Button handler" task. The Eval board contains a user push button that is configured to generate interrupts. The interrupt handler uses a semaphore to wake the button handler task, demonstrating how the priority mechanism can be used to defer interrupt processing to the task level. The button handler task sends a message both to the LCD (via the print task) and the UART where it can be viewed

using a dumb terminal (via the UART to USB converter on the Eval board).

```
static void vButtonHandlerTask(
void *pvParameters )
{
/*declarations*/

 for( ;; )
 {
  /* Wait for a GPIO interrupt to
  wake this task. */
  while( xSemaphoreTake(
  xButtonSemaphore,
  portMAX_DELAY ) != pdPASS );

  /* Start the Tx of the message
  on the UART. */
  UARTIntDisable( UART0_BASE,
  UART_INT_TX );
  {
   pcNextChar = cMessage;

   /* Send the first character. */
   if( !( HWREG( UART0_BASE +
   UART_O_FR ) & UART_FR_TXFF ) )
   {
    HWREG( UART0_BASE +
    UART_O_DR ) = *pcNextChar;
   }

   pcNextChar++;
  }
  UARTIntEnable(UART0_BASE,
  UART_INT_TX);

  /* Queue a message for the print
  task to display on the LCD. */
  xQueueSend( xPrintQueue,
  &pcInterruptMessage,
  portMAX_DELAY );

 }
}
```

A "Check" task. The check task only executes every five seconds but has a high priority so is guaranteed to get processor time. Its function is to check that all the other tasks are still operational and that no errors have been detected at any time. If no errors have every been detected "PASS" is written to the display (via the print task), if an error has ever been detected the message is changed to "FAIL". The position of the message is changed for each write.

```
static void vCheckTask(
void *pvParameters )
{
/*declarations*/

/* Initialise xLastExecutionTime
so the first call to vTaskDelayUntil()
works correctly. */
xLastExecutionTime =
xTaskGetTickCount();

 for( ;; )
 {
  /* Perform this check every
  mainCHECK_DELAY milliseconds. */
  vTaskDelayUntil(
  &xLastExecutionTime,
  mainCHECK_DELAY );

  /* Has an error been found
  in any task? */

  if( xAreIntegerMathsTaskStillRunning()
  != pdTRUE )
  {
   xErrorOccurred = pdTRUE;
  }

  if( xArePollingQueuesStillRunning()
  != pdTRUE )
  {
   xErrorOccurred = pdTRUE;
  }

  if( xAreSemaphoreTasksStillRunning()
  != pdTRUE )
  {
   xErrorOccurred = pdTRUE;
  }

  if( xAreBlockingQueuesStillRunning()
  != pdTRUE )
  {
   xErrorOccurred = pdTRUE;
  }

 /* Send either a pass or fail message.
 If an error is found it's never cleared
 again. We do not write directly to
 the LCD, but instead queue a message
 for display by the print task. */

  if( xErrorOccurred == pdTRUE )
  {
   xQueueSend( xPrintQueue,
   &pcFailMessage, portMAX_DELAY );
  }
  else
  {
```

```
    xQueueSend( xPrintQueue,
    &pcPassMessage, portMAX_DELAY );
   }
  }
}
```

4.1.11 Creating a new FreeRTOS port

Each port is moderately unique and very dependent on the processor and tools being used, so it's very difficult to provide specifics on the porting detail. There are however plenty of other FreeRTOS ports already in existence and it is suggested that these are used as a reference. Porting within the same processor family is a much more straight forward task, for example, from one ARM7 based device to another. The FreeRTOS kernel source code is generally contained within 3 source files (4 if co-routines are used) that are common to all ports, and one or two "port" files that tailor the kernel to a particular architecture. Suggested steps:

1. Download the latest version of the FreeRTOS source code.
2. Unzip the files into a convenient location, taking care to maintain the directory structure.
3. Familiarize ourselves with the source code organization and directory structure.
4. Create a directory that will contain the "port" files for the architecture port.
   Following the convention outlined in the link, the directory should be of the form: FreeRTOS/Source/-portable/compiler name/processor_name. For example, if we are using the GCC compiler we can create a [architecture] directory off the existing FreeRTOS/Source/portable/GCC directory.
5. Copy empty port.c and portmacro.h files into the directory we have just created. These files should just contain the stubs of the functions and macro's that require implementing. See existing port.c and portmacro.h files for a list of such functions and macros. We can create a stub file from one of these existing files by simply deleting the function and macro bodies.
6. Does the architecture stack grow downwards? If not the `portSTACK_GROWTH` macro in portmacro.h needs editing from -1 to 1.
7. Create a directory that will contain the demo application files for the architecture port. Following the convention again this should be of the form FreeRTOS/Demo/architecture_compiler, or something similar.
8. Copy existing FreeRTOS Config.h and main.c files into the directory just created. Again these should be edited to be just stub files.
9. Take a look at the FreeRTOS Config.h file. It contains some macro's that will need setting for our chosen hardware.
10. Create a directory off the directory just created and call it ParTest (probably FreeRTOS/Demo/architecture_compiler/ParTest). Copy into this directory a ParTest.c stub file.
11. ParTest.c contains three simple functions to:
    - setup some GPIO that can flash a few LEDs,
    - set or clear a specific LED, and
    - toggle the state of an LED.

These three functions need implementing for our development board. Having LED outputs working will facilitate the rest of the required work. Take a look at the many existing ParTest.c files included in the other demo projects for examples (the ParTest name is a historical anomaly for PARallel port TEST), and the page describing how to modify an existing demo application for information on writing and testing the ParTest.c functions.

4.2 Contiki

Contiki [12] is an open source, highly portable, multitasking operating system for memory-efficient networked embedded systems and wireless sensor networks. Contiki has been used in a large variety of projects, such as road tunnel fire monitoring, intrusion detection, water monitoring in the Baltic Sea, and in surveillance networks.

4.2.1 Supported architectures

MSP430, AVR.

4.2.2 License

Contiki is distributed under the BSD licensed, and unlikely other RTOS there isn't any commercial version.

4.2.3 Structure

A Contiki system is partitioned into two parts: the *core* and the *loaded programs*. The partitioning is made at compile time and it is specific to the deployment in which Contiki is used. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image that is stored in the devices prior to deployment. The core is generally not modified after deployment, even though it should be noted that it is possible to use a special boot loader to overwrite or patch the core.

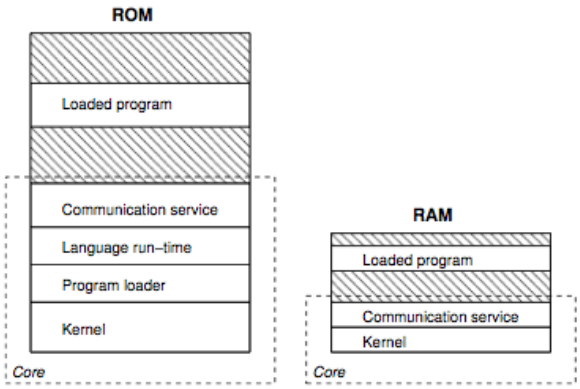

Figure 6: Contiki memory management [13]

4.2.4 Kernel

Contiki is based on an event-driven kernel, but provides support for both multi-threading and *protothreads*, a lightweight stackless thread-like construct. In Contiki, preemptive multi-threading is implemented as a library on top of the event-based kernel. The library is optionally linked with applications that explicitly require a multi-threaded model of operation. Unlike normal Contiki processes, each thread requires a separate stack. The library provides the necessary stack management functions. Threads execute on their own stack until they either explicitly yield or are preempted.
As we said before, the event driven Contiki kernel does not provide multi-threading by itself, instead, preemptive multi-threading is implemented as a library that optionally can be linked with applications.

Protothreads A process in Contiki consists of a single protothread. Protothreads are a type of lightweight stackless threads designed for severely memory constrained systems such as deeply embedded systems or sensor network nodes. Protothreads provides linear code execution for event-driven systems implemented in C. Protothreads can be used with or without an RTOS. This particular type of thread is useful to provide a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The purpose of protothreads is to implement sequential flow of control without complex state machines or full multi-threading. Protothreads provides conditional blocking inside C functions. In purely event-driven systems, blocking must be implemented by manually breaking the function into two pieces - one for the piece of code before the blocking call and one for the code after the blocking call. This makes it hard to use control structures such as if() conditionals and while() loops. The advantage of protothreads over ordinary threads is that a protothread do not require a separate stack. In memory constrained systems, the overhead of allocating multiple stacks can consume large amounts of the available memory.

In contrast, each protothread only requires between two and twelve bytes of state, depending on the architecture.

Scheduling The real-time module handles the scheduling and execution of real-time tasks (with predictable execution times). A protothread is driven by repeated calls to the function in which the protothread is running. Each time the function is called, the protothread will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

4.2.5 Memory

The memory block allocation routines provide a simple yet powerful set of functions for managing a set of memory blocks of fixed size. A set of memory blocks is statically declared with the *MEMB()* macro. Memory blocks are allocated from the declared memory by the *memb_alloc()* function, and are deallocated with the *memb_free()*.

4.2.6 Drivers

Due to its network-base approach Contiki provides IP communication, both for IPv4 and IPv6. Contiki and its uIPv6 stack are IPv6 Ready Phase 1 certified and therefore has the right to use the IPv6 Ready silver logo.

4.2.7 Community and support

Detailed documentation [14] and hardware specific guides [15] are available on the official Contiki website, but there isn't an official support forum, that however is replaced by an official mailing list [16].

4.2.8 Skeleton code for Contiki process

This is the skeleton code always used when writing Contiki processes:

```
#include "contiki.h"

/* The PROCESS() statement defines
the process' name. */
PROCESS(my_example_process,
"My example process");

/* The AUTOSTART_PROCESS()
statement selects what process(es)
to start when the module is loaded.*/
AUTOSTART_PROCESSES(
&my_example_process);

/* The PROCESS_THREAD() contains
 the code of the process. */
PROCESS_THREAD(
my_example_process, ev, data)
{
 /* Do not put code before the
```

```
 PROCESS_BEGIN() statement -
 such code is executed every
 time the process is invoked. */
 PROCESS_BEGIN();
 /* Initialize stuff here. */
 while(1) {
    PROCESS_WAIT_EVENT();
    /*Do the rest of the stuff here.*/
  }
  /* The PROCESS_END() statement
 must come at the end of the
 PROCESS_THREAD(). */
 PROCESS_END();
}
```

4.2.9 The Contiki build system

The Contiki build system is designed to make it easy to compile Contiki applications [17] for either to a hardware platform or into a simulation platform by simply supplying different parameters to the make command, without having to edit makefiles or modify the application code. The file example project in examples/hello-world/ shows how the Contiki build system works. The hello-world.c application can be built into a complete Contiki system by running the `make` command in the examples/hello-world/ directory. Running make without parameters it will build a Contiki system using the native target. The native target is a special Contiki platform that builds an entire Contiki system as a program that runs on the development system. After compiling the application for the native target it is possible to run the Contiki system with the application by running the file hello-world.native. To compile the application and a Contiki system for the ESB platform the command `make TARGET=esb` is used. This produces a hello-world.esb file that can be loaded into an ESB board. To compile the hello-world application into a stand-alone executable that can be loaded into a running Contiki system, the command `make hello-world.ce` is used. To build an executable file for the ESB platform, `make TARGET=esb hello-world.ce` is run. To avoid having to type TARGET= every time make is run, it is possible to run `make TARGET=esb savetarget` to save the selected target as the default target platform for subsequent invocations of make. A file called Makefile.target containing the currently saved target is saved in the project's directory.

Makefiles used in the Contiki build system The Contiki build system is composed of a number of Makefiles.

- Makefile: the project's makefile, located in the project directory.
- Makefile.include: the system-wide Contiki makefile, located in the root of the Contiki source tree.
- Makefile.$(TARGET) (where `$(TARGET)` is the name of the platform that is currently being built): rules for the specific platform, located in the platform's subdirectory in the platform/ directory.
- Makefile.$(CPU) (where `$(CPU)` is the name of the CPU or microcontroller architecture used on the platform for which Contiki is built): rules for the CPU architecture, located in the CPU architecture's subdirectory in the cpu/ directory.
- Makefile.$(APP) (where `$(APP)` is the name of an application in the apps/ directory): rules for applications in the apps/ directories. Each application has its own makefile.

The Makefile in the project's directory is intentionally simple. It specifies where the Contiki source code resides in the system and includes the system-wide Makefile, Makefile.include. The project's makefile can also define in the APPS variable a list of applications from the apps/ directory that should be included in the Contiki system. The Makefile used in the hello-world example project looks like this:

```
CONTIKI = ../..
all: hello-world
include \$(CONTIKI)/Makefile.include
```

First, the location of the Contiki source code tree is given by defining the `CONTIKI` variable. Next, the name of the application is defined. Finally, the system-wide Makefile.include is included. The Makefile.include contains definitions of the C files of the core Contiki system. Makefile.include always reside in the root of the Contiki source tree. When make is run, Makefile.include includes the Makefile.$(TARGET) as well as all makefiles for the applications in the APPS list (which is specified by the project's Makefile). Makefile.$(TARGET), which is located in the platform/$(TARGET)/ directory, contains the list of C files that the platform adds to the Contiki system. This list is defined by the `CONTIKI_TARGET_SOURCEFILES` variable. The Makefile.$(TARGET) also includes theMakefile.$(CPU) from the cpu/$(CPU)/ directory. The Makefile.$(CPU) typically contains definitions for the C compiler used for the particular CPU. If multiple C compilers are used, the Makefile.$(CPU) can either contain a conditional expression that allows different C compilers to be defined, or it can be completely overridden by the platform specific makefile Makefile.$(TARGET).

4.2.10 Porting Contiki

Contiki is designed to be easily portable across platforms. In general, no modifications to the core C source code files are necessary. Here is a quick guide for getting started with porting Contiki. When porting Contiki, there are two directories to be considered: `platform/` and `cpu/`. The platform/ directory contains platform-specific files for the platform on which Contiki runs and the cpu/ directory

contains CPU-specific files for the platforms in the platform/ directory. Each platform has its own directory under platform/. If there are more than one platform that uses a particular CPU it may make sense to create a CPU-specific directory under cpu/. Otherwise, we'll just keep all files in the platform's subdirectory in platform/.

To get started, we take a look at the simplest platform: platform/native/. This is defined for the x86 CPU, so it uses files in the cpu/x86/ subdirectory too. To get started, we copy the files from platform/native/ into a new subdirectory under platform/ (say, platform/my-platform/) and get to work on it. The main() function in contiki-main.c is responsible for setting up the Contiki system (calling all initialization code) and for running the actual Contiki main loop. A New users should take a look at the main() functions in the other platform/ subdirectories too to get a feeling for what typically needs to be done. Once a basic Contiki system is up and running, there are two remaining tasks that are a little more demanding than the other porting tasks: porting the architecture-specific parts of the multithreading library (mtarch.c) to the new CPU, and writing the architecture-specific parts of the ELF loader for the new CPU. However, these are only needed if the platform needs multithreading and dynamically loadable ELF files. Otherwise, nothing needs to be done here.

4.3 BeRTOS

BeRTOS [18] is a real time open source operating system supplied with drivers and libraries designed for the rapid development of embedded software. Perfect for building commercial applications with no license costs nor royalties, BeRTOS allows to cut the economic investment for our products.

4.3.1 Supported architectures

ARM7TDMI, Cortex-M3, AVR, PowerPC, x86, x86-64.

4.3.2 License

BeRTOS is distributed under the term of the GNU GPL with an exception regarding to the possibility of termination of the license in case of addition of some kind of files. In addition BeRTOS a commercial software development kit, that's testable with a 30-days free trial.

4.3.3 Kernel

BeRTOS features both a preemptive and a cooperative kernel with synchronization primitives. The developers decided to implement a strong and robust kernel with very low memory footprint and high modularity. This allows the kernel to be used on a variety of different CPUs, even the smaller ones. Currently the latest kernel code is very stable. The documentation offered isn't very detailed, but here are the main features:

- Preemptive and cooperative round-robin scheduling.
- Heap, for dynamic allocation of processes' memory.
- Stack process monitor, useful to prevent stack overflows.
- Inter-process messaging system (with very low overhead).
- Binary semaphores.
- Signals (for processes synchronization).

And finally the kernel has a port layer (a single assembly function) that needs to be reimplemented each time a new CPU is added.

4.3.4 File System

BeRTOS supports two file systems. The first one is the mainstream *FAT filesystem*, using the *FatFs* library; the second is *BattFs*, a file system specifically planned for embedded platforms. File system development takes place in the fs directory.

4.3.5 Core Drivers

BeRTOS is not only a kernel, it aims to supply full operating system services. To achieve this, it needs at least some core drivers for every CPU port. These drivers are: *system timer, debug system* and *serial comm driver*. Adding CPU support for an already present CPU family is quite simple since hardware manufacturers share peripherals design between the same CPU cores. Core drivers are completely supported on all platforms.

4.3.6 Documentation and community

The documentation is concise, but provides a good description of all the main features offered by BeRTOS. There aren't many guides available, but there is a good tutorial to learn how to use the wizard to install the RTOS on the most common board available on the market. The official website provides also a link [19] to the official community, but it does not appear very much alive, as can be the FreeRTOS's one.

4.3.7 Directory structures

The modules are sorted in subdirectories by their category:

```
BertOS
├── app
├── bertos
│   ├── algo
│   ├── cfg
│   ├── drv
│   ├── dt
│   ├── emul
│   ├── fonts
│   ├── fs
│   ├── gfx
│   ├── gui
│   ├── hw
│   ├── icons
│   ├── kern
│   ├── mware
│   ├── struct
│   └── os
├── boards
└── doc
```

On BeRTOS documentation [20] is possible to find a complete description of all files included in the subdirectories.

4.3.8 First application

The example shown below uses one of the cpu's currently supported by BeRTOS, the ATMega1281.

Project setup First of all, download BeRTOS and launch the wizard. From cfg select debug. This will automatically include debug support in BeRTOS.
Set the correct output port for our cpu and then create the project. Let's call our project hello_world. The wizard creates the directory hello_world/, in which we see:

- bertos/: the whole bertos source tree;
- Makefile: the makefile for the whole project. To build the project, simply launch the make command;
- hello_world/: our project directory
  - cfg/: directory with all the configuration files
  - hw/: hardware specific files
  - main.c: our application entry point

Coding Open main.c and input:

```
#include "buildrev.h"

#include <cfg/debug.h>
int main(void)
{
    IRQ_ENABLE;

    kdbg_init();
    kprintf("Program build: %d\n",
    VERS_BUILD);
    kputs("Hello world!\n");
    return;
}
```

Then compile the program; we will see some warnings but for now we can safely ignore them. Flash our board and reset. On the debugging serial we'll see these messages:

```
**** BeRTOS DBG START ***
Program build: 17
Hello world!}
```

Line by line tutorial

```
\#include <cfg/debug.h>
```

The first line includes function prototypes. Since this is often included by other files, we can omit it in larger projects.

```
IRQ_ENABLE;
```

This line enables IRQs on our CPU. We don't need it for this simple example, but we will need for almost every other module in BeRTOS, so it's a good habit to learn it as soon as possible.

```
kdbg_init();
```

This line initializes the debug subsystem, opening a serial port for debugging. The parameters for this port are in hello_world/cfg/cfg_debug.h.

```
kprintf("Program build: %d\n",
       VERS_BUILD);
kputs("Hello world!\n");
```

Writes a debug string on the output. We can also use a printf-like function to format the output. Be aware, though, that the delay introduced byprintf is high, so use it sparingly. To reduce footprint and cpu usage, BeRTOS implements different types of formatting for printf. We can select our formatting options modifying hello_world/cfg/cfg_formatwr.h. Remember the warnings we got on the first build? These were due to using full formatting on an underpowered cpu. To avoid the warnings we can change printf format option to `PRINTF_NOFLOAT`.

4.3.9 Binary file flashing

Compiled files will be placed in the directory images/ of our project directory. We will find lots of formats, for example .elf, .s19, .hex or .bin. The last BeRTOS version has a flashing and debugging infrastructure integrated directly into the makefile to speed up these repetitive operations. The operations to flash and debug a target vary greatly from CPU to CPU, so we have to write a script which will flash our CPU. For example, when using AVR targets it's enough to run Avrdude or Avarice with the correct parameters. The script may use one of the following environment variables, which are defined directly in the makefile:

- `PROGRAMMER_CPU`: the CPU we are using in our project.
- `PROGRAMMER_TYPE`: the programmer type to use
- `PROGRAMMER_PORT`: the port to which the programmer is connected
- `GDB_PORT`: port to wait GDB connections
- `ELF_FILE`: file path with debug symbols for GDB

Variables `PROGRAMMER_TYPE` and `PROGRAMMER_PORT` are defined in the project's makefile (for example hello_world.mk). Once we have created the scripts, we just need to execute one of the following commands:

```
make flash_hello_world
# or
make debug_hello_world
```

4.3.10 Porting BertOS to a new CPU

This short guide will show how to create support for a new CPU from an architecture already supported by BeRTOS (eg. Cortex-M3).
A port to a new CPU will generally follow these steps:

- find a way to flash our cpu
- create a linker script with the memory map for our CPU
- add macros to let BeRTOS recognize our cpu. Have a look at:
  - cpu/detect.h
  - cpu/attr.h
  - cpu/frame.h (needed only for kernel)
- create a file to init the cpu (.crt file) if needed (see cpu/cortex-m3/crt_cm3.S)
- now try to print something on the serial debug console. We'll need to implement cpu/drv/kdebug_ourcpu.c
- create the timer driver, implement cpu/drv/timer_ourcpu.c

Now we have a minimal BeRTOS system ready to use and all modules except the internal CPU peripherals should work without further effort.

4.4 eCos

eCos (embedded Configurable operating system) [21] is an open source, royalty-free, real-time operating system intended for embedded systems and applications. The highly configurable nature of eCos allows the operating system to be customized to precise application requirements, delivering the best possible run-time performance and an optimized hardware footprint.

4.4.1 Supported architectures

Currently eCos supports a large range of different target architectures: ARM, FR-V, SH2/3/4, H8/300H, x86, MIPS, AM3x, PowerPC, 68k/Coldfire, V850, SPARC including many popular variants of these architectures and evaluation boards.

4.4.2 Boot loader

eCos is provided with the RedBootROM monitor. It's an application that uses the eCos for portability. It provides serial and ethernet based booting and debug services during develop.

4.4.3 License

eCos is distributed under the GPL license with an exception which permits proprietary application code to be linked with eCos without itself being forced to be released under the GPL. It is also royalty and buyout free.

4.4.4 Structure

eCos is designed around a single-process, single-address space, multiple-thread model. Application code is linked directly with the eCos kernel to be deployed in the target environment. Most of eCos is written in C++, with some machine-dependent portions written in C and assembly language. eCos also includes a complete C-language kernel API. C++ is used with care taken to exploit the high-level abstractions available in that environment without introducing unnecessary overheads. To keep run-time overhead low, neither C++ exceptions nor RTTI (RunTime Type Information) are used within the kernel. Additionally, the kernel is completely static (for instance, there are no uses of new), and the only use of C++ templates is to provide type-checked macros.

An example of the use of templates is in the memory-pool management classes, wherein a template is used to wrap generic thread synchronization, waiting, and atomicity around any of several simple memory-pool managers.

4.4.5 Kernel

The eCos kernel was designed to satisfy four main objectives:

- Low interrupt latency, the time it takes to respond to an interrupt and begin executing an ISR.
- Low task switching latency, the time it takes from when a thread becomes available to when actual execution begins.
- Small memory footprint, memory resources for both program and data are kept to a minimum by allowing all components to configure memory as needed.
- Deterministic behavior, throughout all aspects of execution, the kernel's performance must be predictable and bounded to meet real-time application requirements.

Process manager A complete set of kernel features is provided to support applications built using eCos. These include:

- *Thread primitives:* functions to create, destroy, suspend, and resume threads are provided.
- *Synchronization primitives:* multiple mechanisms for thread synchronization are provided, including mutex and condition variables, binary and counting semaphores, message/mail boxes, and general-purpose event flags.
- *Thread synchronization:* to allow threads to cooperate and compete for resources, it is necessary to provide mechanisms for synchronization and communication. The classic synchronization mechanisms are mutexes/condition variables and semaphores. These are provided in the eCos kernel, together with other synchronization/communication mechanisms that are common in real-time systems, such as event flags and message queues
- *Timers, counters, and alarms:* a set of flexible counter and alarm functions is provided. If the hardware provides a periodic clock or timer, it will be used to drive timing-related features of the system. Many CPU architectures now have built in timer registers that can provide a periodic interrupt. This should be used to drive these features where possible. Otherwise an external timer/clock chip must be used.

Scheduling eCos has two kinds of schedulers implemented; bitmap and MLQ (Multi Level Queue) scheduling. Both of them supports preemption. It is possible to extend the eCos kernel to handle other schedulers as well. Both schedulers uses numerical priority levels that ranges from 0 to 31, where 0 is the highest priority.

- *Bitmap scheduling:* Bitmap scheduling consists of a queue with threads that are loaded into the memory. Each thread in the queue has an associated priority. There are 32 different priority levels available and each priority level can only be associated with one thread. This limits the amounts of total of queued threads in the systems to 31 (one being the idle thread). The scheduler always runs the thread with the highest current priority. The scheduler can be configured with or without preemption. If preemption is enabled and a thread with a higher priority than the current running thread enters the queue the scheduler will preempt and run the thread with the highest priority.
- *MLQ scheduling:* The MLQ scheduling is a bit more complicated than bitmap scheduling. Instead of just having one queue for all threads and all priorities, this scheduler uses a set of queues where every queue contains a number of threads all with the same priority. Each queue has its own scheduling algorithm that by default is a time slicing algorithm that shares the CPU among the threads in the queue equally. The different priority levels may have different schedulers. These queues are scheduled by a normal priority queue where higher priority queues gets scheduled first. The threads are divided into the different priorities based on some property of the thread. This scheduler also supports preemption. MLQ also have support for multiple processors with SMP (Symmetric Multi Processing) where each processor has its own scheduler.

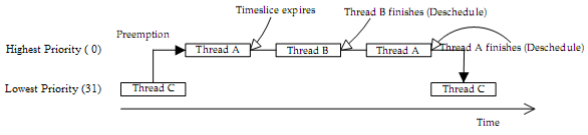

Figure 7: Time-chart of scheduling with MLQ [22]

Summarizing, the Bitmap scheduling, because of its simplistic design is very fast and thus the dispatch latency is low. The simplistic design also makes it easy to predict system behavior, spawning the possibility for a deterministic system overall. One important aspect with this scheduler is the constraint of being limited to 31 threads in the system.

The MLQ scheduling, having a more complicated algorithm, makes the dispatch latency slightly higher than with bitmap scheduling. On the other hand this scheduler contains more advanced features and allows the developer to build systems with a distinct separation of background and foreground tasks, where tasks have different time requirements. Since this scheduler uses a more advanced algorithm it is harder to predict system behavior, which makes this scheduler hard to use when trying to create a deterministic system. In contrast to the bitmap scheduler the MLQ scheduling can handle as many threads as possible (as long as they can fit into the memory).

4.4.6 Documentation

eCos documentation include user guide, the reference manual and the components writer's guide. These are being continually updated as the system develops. Furthermore it is available a book written by eCos developers about every part of the OS and about how to develop software. It's free and downloadable from the official website.

4.5 ChibiOS/RT

ChibiOS/RT [23] is designed for deeply embedded real time applications where execution efficiency and compact code are important requirements. This RTOS is characterized by its high portability, compact size and, mainly, by its architecture optimized for extremely efficient context switching.

4.5.1 Supported architectures

ARM, AVR, MSP430, e200z, STM8.

4.5.2 Footprint

Note, first "fast" then "compact", the focus is on speed and execution efficiency and then on code size. This does not mean that the OS is large, the kernel size with all the subsystems activated weighs around 5.5KB (STM32, Cortex-M3).

4.5.3 License

ChibiOS, in the unstable version, is released under the license of GPL3. This version is intended only for development and evaluation purpose. Stable versions of ChibiOS are released under a modified license of GPL3 with linking exception and under certain conditions. Commercial version of the OS is also available.

Here is a summary table about the various versions:

Question	GPL3	mod.GPL3	Comm.
Is ChibiOS/RT free?	√	√	×
Can I use ChibiOS/RT in my commercial embedded product?	×	√	√
Do I have to release my source code?	√	×	×
Do I have to open source my changes to ChibiOS/RT?	√	√	×
Do I have to document that my product uses ChibiOS/RT?	√	√	×
Do I have to offer to offer the ChibiOS/RT code to users of my product?	√	√	×
Is public support available?	√	√	√
Is professional technical support available?	√	√	√
Is a warranty provided?	×	×	√

4.5.4 Kernel

The kernel itself is very modular and is composed of several subsystems, most subsystems are optional and can be switched of in the kernel configuration file *chconf.h*.

Base Kernel Services This category contains the mandatory kernel subsystems:

- *System*, low level locks, initialization.
- *Timers*, virtual timers and time APIs.
- *Scheduler*, scheduler APIs, all the higher level synchronization mechanism are implemented through this subsystem, it is very flexible but not recommended for direct use in application code.
- *Threads*, thread-related APIs.

Synchronization This category contains the synchronization-related subsystems, each of the provided mechanism can be configured out of the kernel if not needed.

- *Semaphores*, counter and binary semaphores subsystem.
- *Mutexes*, mutexes subsystem with support to the priority inheritance algorithm (fully implemented, any depth).

- *Condvars*, condition variables, together with mutexes the condition variables allow the implementation of monitor constructs.

- *Events*, event sources and event flags with flexible support for and/or conditions and automatic dispatching to handler functions.

- *Messages*, lightweight synchronous messages.

- *Mailboxes*, asynchronous messages queues.

Memory management This category contains the memory management related subsystems:

- *Core Allocator*, centralized core memory manager, this subsystems is used by the other allocators in order to get chunks of memory in a consistent way.

- *Memory Heaps*, central heap manager using a first fit strategy, it also allows the creation of multiple heaps in order to handle non uniform memory areas.

- *Memory Pools*, very fast fixed size objects allocator.

- *Dynamic Threads*, usually threads are static objects in ChibiOS/RT but there is the option for dynamic threads management.

4.5.5 Priorities in ChibiOS/RT

Priorities in ChibiOS/RT are a contiguous integer range starting from `LOWPRIO` up to `HIGHPRIO`, the *main()* thread is started at the `NORMALPRIO` priority level which is in the middle of the allowable range. When designing our application we should never think in terms of absolute priorities because the numeric range may change in future versions, much better reason in terms of relative priorities from the above symbolic levels, for example `NORMALPRIO+5`, `NORMALPRIO-1`, this is guaranteed to be portable.

Priorities table The following priority levels are defined:

Priority	Description
`zero`	Reserved priority level, all the possible priority levels are guaranteed to be greater than zero.
`IDLEPRIO`	Special priority level reserved for the Idle Thread.
`LOWPRIO`	Lowest priority level usable by user threads.
`NORMALPRIO`	Central priority level, the main() thread is started at this level. User thread priorities are usually allocated around this central priority.
`HIGHPRIO`	Highest priority level usable by user threads. Above this level the priorities are reserved.
`ABSPRIO`	Absolute priority level, highest reserved priority level. Above this levels there are the hardware priority levels used by interrupt sources.

How priorities work The rule is very simple, among all the threads ready for execution, the one with the highest priority is the one being executed, no exceptions to this rule. Priorities are not related to CPU use percentages, a thread at level `NORMALPRIO+40` is perfectly equivalent to a thread at `NORMALPRIO+5` if in between there are no other threads. Remember, the absolute values have no meaning, only the relative priorities among all the existing threads are important. Unlike many other RTOSs, ChibiOS/RT allows for multiple threads at the same priority level, such threads are rotated using a Round Robin algorithm.

What about interrupts In ChibiOS/RT we should consider interrupts as special tasks having a priority range allocated above the priority range reserved for normal threads. The base rule still applies, the highest priority task/thread is the one being executed. This is especially true in those architectures, like the Cortex-Mx for example, that allow the preemption of interrupt handlers.

4.5.6 Memory

In an RTOS like ChibiOS/RT there are several dedicated stacks, each stack has a dedicated RAM space that must have a correctly sized assigned area. There are several stacks in the systems, some are always present, some others are present only in some architectures:

- *Main stack*, this stack is used by the main() function and the thread that executes it. It is not a normal thread stack because it is initialized in the startup code and its size is defined in a port dependent way. Details are in the various ports documentation.

- *Interrupt Stack*, some architectures have a dedicated interrupt stack.

This is an important feature in a multithreaded environment, without a dedicated interrupt stack each thread has to reserve enough space, for interrupts servicing, within its own stack. This space, multiplied by the total threads number, can amount to a significant RAM overhead.

- *Thread Stack*, each thread has a dedicated stack for its own execution and context switch.

- *Other Stacks*, some architectures (ARM) can have other stacks but the OS does not directly use any of them.

Thread management In ChibiOS/RT threads are divided in two classes:

- *Static threads.* The memory used for static threads is allocated at compile time so static threads are always there, there is no management to be done.

- *Dynamic threads.* Dynamic threads are allocated at runtime from one of the available allocators.

Dynamic threads create the problem of who is responsible of releasing their memory because a thread cannot dispose its own memory, this is handled in ChibiOS/RT through the mechanism of "thread references". When the `CH_USE_DYNAMIC` option is enabled the threads become objects with a reference counter. The memory of a dynamic thread is freed when the last reference to the thread is released while the thread is in its `THD_STATE_FINAL` state. The following diagram explains the mechanism:

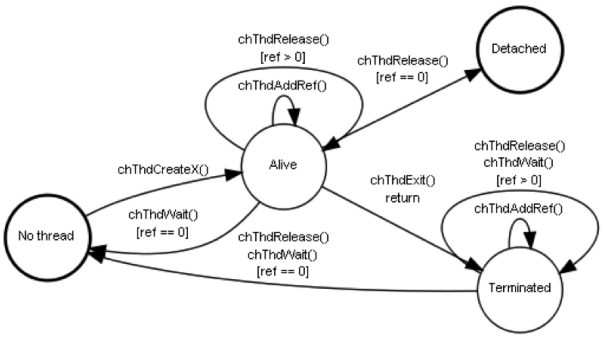

Figure 8: ChibiOS/RT scheduling [24]

As we can see the easiest way to ensure that the memory is released is to make another thread perform a *chThdWait()* on the dynamic thread. If all the references to the threads are released while the thread is still alive then the thread goes in a "detached" state and its memory cannot be recovered unless there is a dedicated task in the system that scans the threads through the registry subsystem, scanning the registry has the side effect to release the zombies (detached and then terminated threads).

4.5.7 Documentation and community

The official site contains a lot of document including brief introduction, large kernel description, requirements, supported architectures and a good support forum [25]. It's also present some full kernel reference manual for the main supported boards. In addiction the site contains general articles about kernel, RTOS and guides useful to perform common RTOS-related task in ChibiOS/RT [26].

4.5.8 Files organization

```
root
├── readme.txt
├── documentation.html
├── license.txt
├── boards/
├── demos/
├── docs/
│   ├── html/
│   ├── reports/
│   ├── src/
│   ├── rsc/
│   ├── Doxyfile
│   └── index.html/
├── ext/
├── os/
│   ├── hal/
│   │   ├── include/
│   │   ├── src/
│   │   ├── platforms/
│   │   │   ├── /ldots/
│   │   │   └── Win32/
│   │   ├── templates/
│   │   │   └── meta/
│   │   ├── ports/
│   │   └── kernel/
│   │       ├── include/
│   │       ├── src/
│   │       └── templates/
│   └── various/
├── test/
│   └── coverage/
└── testhal/
```

4.5.9 Getting started

To develop our application we can start, as usual, by modifying existing demos present in the code. We have just to modify the main.c. After initializing ChibiOS/RT using *chSysInit()* two threads are spawned by default:

- *Idle thread.* This thread has the lowest priority in the system so it runs only when the other threads in the system are sleeping. This threads usually switches the system in a low power mode and does nothing else.

- *Main thread*. This thread executes our main() function at startup. The main thread is created at the `NORMALPRIO` level but it can change its own priority if required. It is from the main thread that the other threads are usually created.

As we said before, there are two classes of threads in ChibiOS/RT:

- Static Threads. This class of threads are statically allocated in memory at compile time.
- Dynamic Threads. Threads created by allocating memory at run time from a memory heap or a memory pool.

Creating a static thread In order to create a static thread a working area must be declared using the macro `WORKING_AREA` as shown:

```
static WORKING_AREA(myThreadWorkingArea,
128);
```

This macro reserves 128 bytes of stack for the thread and space for all the required thread related structures. The total size and the alignment problems are handled inside the macro, we only need to specify the pure and simple desired stack size.
A static thread can be started by invoking *chThdCreateStatic()* as shown in this example:

```
Thread *tp = chThdCreateStatic(
    myThreadWorkingArea,
    sizeof(myThreadWorkingArea),
    NORMALPRIO, /* Initial priority. */
    myThread, /* Thread function. */
    NULL); /* Thread parameter. */
```

The variable *tp* receives a pointer to the thread object, this pointer is often taken as parameter by other APIs. Now a complete example:

```
#include <ch.h>

/* Working area for the LED
  flashing thread. */
static WORKING_AREA(
myThreadWorkingArea, 128);

/* LED flashing thread. */
static msg_t myThread(void *arg) {

  while (TRUE) {
    LED_ON();
    chThdSleepMilliseconds(500);
    LED_OFF();
    chThdSleepMilliseconds(500);
  }
}

int main(int argc, char *argv[]) {

  /* Starting the flashing LEDs thread.*/
  (void)chThdCreateStatic(
  myThreadWorkingArea,
  sizeof(myThreadWorkingArea),
  NORMALPRIO, myThread, NULL);
}
```

Note that the memory allocated to *myThread()* is statically defined and cannot be reused. Static threads are ideal for safety applications because there is no risk of a memory allocation failure because progressive heap fragmentation. Now we can compile and run our simple thread. For example, if the demo is located under `./demos/ARMCM3-STM32F100-DISCOVERY`, it can be compiled by simply giving the make command. We may consider to import the whole project under an Eclipse Toolchain.

4.5.10 Porting ChibiOS/RT

Porting the operating system on a new platform is one of the most common tasks. The difficulty can range from easy to very difficult depending on several factors. We can divide in problem in several classes of progressively increasing difficulty:

- Porting the OS to a different board using the same MCU.
- Porting the OS to a different MCU belonging to the same family.
- Porting the OS to another MCU using the same core.
- Porting the OS to a whole new architecture.

Another kind of port type is porting to another compiler and this is an added complexity level on the above classes. The kernel itself is portable but the port-specific code usually contains compiler specific extensions to the C language and the asm files syntax is almost never compatible.

Porting to a new board This is the easiest port type, the scenario is that the specific MCU is already supported and a demo exists. This scenario also applies when porting the OS on a custom hardware using a supported MCU. This is the procedure to follow:

- Create a new directory under `./boards` and copy inside the board files from another board using the same MCU.

- Customize the board files:
  - *board.h*, this file contains the I/O pins setup for the MCU, it may also contain other board-dependent settings, for example, the clock frequency. Customize this file depending on our target hardware.
  - *board.c*, this file contains the initialization code, often we just need to customize @p board.h and not this file. If we have some board-specific initialization code then put it here.
- Create a new directory under the ChibiOS/RT installation directory: *./myprojects/<my_app_name>*.
- Copy an existing demo code under the newly created directory.
- Customize the following demo files:
  - *Makefile*, we may edit this file in order to remove the test-related sources and/or add our application source files.
  - *main.c*, this file contains the demo code, clean it and write our own main() function here, use this file just as a template.
- Compile our application and debug.

Porting to a closely related MCU In this scenario all the above steps are required but an analysis must be performed to evaluate the differences between from the supported micro and the target micro. Often the micros just differ for the memory area sizes and a change to the linker script is enough (this file is usually named ch.ld). Chips just having more or less peripherals, everything else being the same or compatible, should not a problem too. If there are differences in the internal peripherals, for example non compatible interrupt controllers (this happens in the LPC2000 family) or differences in UARTS, timers etc, then the port belongs to the following category.

Porting the OS to another MCU using the same core This kind of port is required when the target MCU has the same core (a common example: ARM7) of a supported MCU but has differences in the internal peripherals. If this is our case proceed as follow:

- Create a new directory under ./os/hal/platforms and then name it with the MCU name (or family name). In case of an ARM-based MCUs we also need to create an equally named directory under ./os/-ports/<compiler>/<arch> and put there the MCU related files such as the vectors table, see the existing ports for example.
- Copy into the newly created directory the most closely related existing platform files or the naked template files from ./os/hal/templates.
- Work out the differences in the drivers or implement them from scratch if we started from the templates.
- Edit/create the documentation file platform.dox, this is only required if we want to regenerate the documentation including our work.

Usually this kind of ports just require a serial driver (and those are very similar each other) and some code for the interrupt controller (is some cases this one can be part of the core port instead, for example the Cortex-M3 core has an integrated interrupt controller). When the platform port is completed we can create our application as seen in the previous sections.

Porting the OS to a whole new architecture This is the hardest scenario, the time required by core ports depends strongly by the target architecture complexity and the level of support we need for the architecture specific features. Porting an OS to a new architecture is not an easy task and it required a very big experience for such an effort. Just follow the directory patterns and fill the OS template files, the hardest part is decide the correct and efficient way to implement the context switching.

5 Comparison

In this section we will compare the features of every RTOS we've analyzed. All these systems have common features, for example they are all almost free (and open-source), they require little memory and hardware specific and quite all of them are very customizable, but everyone has some different components and could be more useful for a specific ambient or another one. Comparing them side by side is useful to make a final choice about the OS that best fit our requirements.

5.1 General features

Name	Target	Platforms	Site
FreeRTOS	embedded	ARM, AVR, AVR32, HCS12, IA32, MSP430, PIC, H8/S, 8052, STM32	[7]
Contiki	embedded	AVR, MSP430	[12]
BertOS	embedded	DSP56K, I196, IA32, ARM, AVR	[18]
eCos	general purpose	ARM, RISC, 68000, fr30, FR-V, H8, IA32, MIPS, MN10300, PowerPC, SPARC, SuperH, V8xx	[21]
ChibiOS/RT	embedded, small footprint	x86, ARM, Cortex-M3, PowerPC e200z, STM8, AVR, MSP430, Coldfire, H8S	[23]

5.2 License

FreeRTOS Modified GNU GPL license, but with alternative commercial licenses like OpenRTOS, SafeRTOS.

Contiki BSD license, no commercial versions.

BeRTOS GNU GPL, but with a commercial software development kit (testable with a 30-days free trial).

eCos Modified GNU GPL.

ChibiOS/RT GPL3 for unstable version, GPL3 modified for stable version; commercial versions also available.

5.3 Main features

FreeRTOS Very small kernel, idle hook, semaphores and mutexes, Memory Protection Unit, stack overflow detection, co-rutines, unlimited number of tasks.

Contiki Event-driven kernel, protothreads, preemptive multithreading, native TCP/IP stack support, small memory requirements.

BeRTOS Heap for dynamic allocation memory, stack overflow protection, semaphores, inter-process messaging system.

eCos Highly configurable to reach best performances, low interrupt and task switching latency, small memory footprint, multiple synchronization mechanisms.

ChibiOS/RT High portability, efficient context switching, many synchronization mechanisms, full dynamic memory management.

5.4 Scheduling

FreeRTOS Queue with threads of different priorities, managed with round-robin policy; preemptive feature.

Contiki Handled by the event manager, that calls the thread; preemptive feature.

BeRTOS Preemptive and cooperative with round-robin policy.

eCos Bitmap (one queue with different single priorities) or MLQ (multiple level queue with customizable policies). Preemptive feature for both methods.

ChibiOS/RT Threads with relative priorities, rotated using Round-Robin policy.

5.5 Development

FreeRTOS Almost in C, Windows AVR cross-compiler based on GCC.

Contiki Presence of drivers for TCP/IP communication; very simple compilation through C code and make command

BeRTOS Presence of core drivers for all supported CPUs; simple and complete wizard for coding, porting and debugging.

eCos Provided with the RedBootROM monitor, application useful for booting and debugging during develop.

ChibiOS/RT Lot of forum and example but unluckily very poor development tools.

6 Conclusion

The software is a very important factor for the development of a device or an application that operates in real time. The operating system is the first component that is considered for not to run into insurmountable problems of impracticably of the project. It does not necessarily have to be fast, it does not matter the time interval in which the operating system/application should react, the most important thing is to respond within a maximum pre-determined time. In other words, the system must be *predictable*.

The purpose of this document was to explain why a RTOS is developed, how it works and in which environment it can be used. We analyzed the most active open source Real Time Operating Systems in the main features like kernel, scheduling, drivers and supported architectures. We also reported how they are documented and how the community support works. We explained how a developer can start to write code that can run in a board with a specific RTOS. Finally we focus on how the operating systems can be ported on a new board or architecture.

References

[1] Juvva, K.: Real-time systems. (1998)

[2] Alippi, C.: Prometheus project. http://www.sport.lecco.polimi.it/index.php?option=com_content&view=article&id=337 Politecnico di Milano.

[3] Schreiber, F.A.: Perla project. http://perla.dei.polimi.it Politecnico di Milano.

[4] Instruments, N.: Do i need real-time system? http://zone.ni.com/devzone/cda/tut/p/id/10342

[5] Brandolese, C., Fornaciari, W.: Sistemi Embedded. Pearson Prentice Hall (2007)

[6] Kioskea.net: Sistema operativo. http://it.kioskea.net/contents/systemes/sysintro.php3

[7] FreeRTOS: official website. http://www.freertos.org/

[8] FreeRTOS: licensing. http://www.freertos.org/a00114.html

[9] FreeRTOS: the rtos tick. http://www.freertos.org/implementation/a00011.html

[10] FreeRTOS: community. http://sourceforge.net/projects/freertos/forums/forum/382005

[11] FreeRTOS: source tree. http://www.freertos.org/a00017.html

[12] Contiki: official website. http://www.sics.se/contiki/

[13] Dunkelsa, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. (2007)

[14] Contiki: documentation. http://www.sics.se/~adam/contiki/docs/

[15] Contiki: wiki pages. http://www.sics.se/contiki/wiki/index.php/Main_Page

[16] Contiki: support. http://sourceforge.net/projects/contiki/support

[17] Contiki: tutorial. http://www.sics.se/contiki/wiki/index.php/Develop_your_first_application

[18] BeRTOS: official website. http://www.bertos.org/

[19] BeRTOS: support. http://www.bertos.org/support/

[20] BeRTOS: source tree. http://www.bertos.org/support/

[21] eCos: official website. http://ecos.sourceware.org/

[22] Salenby, G., Lundgren, D.: Comparison of scheduling in freertos and ecos. http://www.ida.liu.se/~TDDB72/rtproj/reports2006/36-danlu911gussa217-FreeRTOS_eCos_scheduling.pdf

[23] ChibiOS/RT: official website. http://www.chibios.org/

[24] ChibiOS/RT: documentation. http://chibios.sourceforge.net/html/index.html

[25] ChibiOS/RT: support forum. http://forum.chibios.org/phpbb/index.php

[26] ChibiOS/RT: tutorial. http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:stm32vl_discovery