A Survey of Operating Systems Infrastructure for Embedded Systems

Luís Fernando Friedrich

DI-FCUL TR–09–3

Fevereiro 2009

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749-016 Lisboa
Portugal

Technical reports are available athttp://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

A Survey of Operating Systems Infrastructure for Embedded Systems

Luís Fernando Friedrich [1]
Departmento de Informática e Estatística
- Centro Tecnológico
Universidade Federal de Santa Catarina, Brazil
fernando@inf.ufsc.br

Abstract

Since early applications in the 1960s, embedded systems have come down in price and there has been a dramatic rise in processing power and functionality. In addition, embedded systems are becoming increasingly complex. High-end devices, such as mobile phones, PDAs, entertainment devices, and set-top boxes, feature millions of lines of code with varying degrees of assurance of correctness. Nowadays, more and more embedded systems are implemented in a distributed way, a wide range of high-performance distributed embedded systems have been designed and deployed. As a lot of aspects of embedded system design become increasingly dependent on the effective interaction of distributed processors, it is clear that as much effort needs to be focused on software infrastructure, such as operating systems, with respect to how to provide functionality in order to fulfill these requirements. This technical report presents some of the approaches associated to operating systems that have been used in order to fulfill these needs.

1. Introduction

Embedded systems can be defined as a combination of any device that includes a programmable computer and perhaps additional parts, either mechanical or electronic, designed to perform a dedicated function, but is not itself intended to be a general purpose computer. The word embedded reflects the fact that these systems are typically a fundamental part of a larger system. Typically, embedded systems are also called embedded real-time systems. We can find a large variety of applications where embedded systems play an important role, from small stand-alone systems, like a network router, to complex distributed real-time embedded systems (DRE) supporting several large scale mission-critical domains as avionic applications.

All embedded systems have a dedicated functionality and are therefore dedicated systems. Dedicated functionality means that the system has been designed for a specific purpose and pre-defined tasks. Moreover, the system functionality is predefined in the hardware and software.

The variety of applications implies that the properties, platforms, and techniques on which embedded systems are based can be very different. The hardware needs can

[1] This work was supported by CAPES / MEC – Brazil through Project BEX3342/08-5 developed at Department of Informatics – FCUL, Lisbon – Portugal.

sometimes be achieved with the use of general purpose processors. For instance, high-end devices, including mobile phones, PDAs, and consumer electronics (entertainment devices such as TVs and DVD players, set-top boxes, etc.), have incorporated microprocessors as a core system component, instead of using specific hardware. But in many systems specific processors are required, for instance, specific DSP devices to perform fast signal processing. In exceptional cases where reaction times are extremely small, microprocessor technology cannot always deal with the timing constraints and the microprocessor technology is then replaced by hard cabled electronic logic devices.

The functionality can be modified or adjusted through software changes. It is possible to add functionality to the devices via a software upgrade as long as the hardware doesn't need a modification, or the available memory space is large enough to accommodate the changes. For instance, high-end devices feature millions of lines of code with varying degrees of assurance of correctness. They might incorporate third party components, and even complete operating systems (such as Linux) that can be installed by the manufacturer, suppliers and even the end user. In such cases, it becomes impossible for embedded system vendors to provide guarantees about the behavior of the device, when supporting such devices using traditional real-time executive unprotected approach. Failure or malicious behavior of a single software component on the device will affect the whole device.

It is also possible to modify (part of) the hardware functionality, using technology such as FPGA for example. Today these modifications are limited and can only be done when the device is not performing its normal tasks. However in the future, this might change, and such modifications are going to be executed while the device continues to perform other of its functions.

Memory management capabilities are necessary in some systems to provide memory protection and virtual memory. Special purpose interfaces are also needed to support a variety of external peripheral devices, energy consumption control, and so on.

In mission-critical systems besides having to meet deadlines, tasks are said essentially critical and require special real-time responsiveness. However, beyond survivability mission-critical systems must also satisfy the same rigid dependability requirements of reliability and fault tolerance. A high amount of adaptability of system functions is demanded when dealing with such requirements.

Nowadays, the use of processor-based devices has increased dramatically for most of our activities, both professional and leisure. This trend is expected to grow exponentially in the near future. The rapid progress in processor and sensor technology combined with the expanding diversity of application fields is placing enormous demands on the facilities that software infrastructure like operating systems must provide.

This paper surveys the current state of embedded systems from small stand-alone to distributed real-time looking at some of the software infrastructure used to provide the functionality they need. Software infrastructure is referring to what is usually called standard software and include: embedded operating systems, real-time operating systems and other forms of middleware. First we present concepts and characteristics of embedded systems and propose a classification. Then, it is presented some requirements

that usually are very useful for embedded systems. Next, we present software infrastructure alternatives which are intended to provide the necessary functional and non-functional requirements for embedded system software to execute. Finally, we conclude with some tendencies of software infrastructure for the next generations embedded applications.

2. Characterization of Embedded Systems

An embedded system is hidden inside a system or environment, performing some dedicated function. The word embedded indicates that the system or device is part of another (larger) system. The hosting system may be a specific system such as a car or an aircraft, a machine or a factory, but it may also be a person in the case of an intelligent pace maker or some hearing device, where the embedded system replaces or extends the human capabilities. In some types of embedded systems or devices, the term ubiquitous is sometimes used to point out that the computation is integrated in the environment. Embedding computation into the environment and everyday objects (also called pervasive computing) would enable people to cooperate with information-processing devices in a more informal way than they currently do, and independent of their location or situation they find themselves.

As there is an enormous variety of embedded systems from small intelligent sensors to vast aircraft control systems and vehicle simulators, the functions they performed may vary a lot. Some of the categories these functions can be put in are: Computation (giving the intelligence to the overall system); Measurement via sensors; Control of the environment via actuators after decisions made during the computation; Communication (including data, music, video etc..); Human interface (via a display and some buttons or a (limited) keyboard or touch screen). Different embedded systems will need different functions, not all functions are necessary on an embedded system. For instance, a robot will not always have a human interface.

Examples of how real-time and embedded systems provide us with services are in our daily life when we go from one place to another we have services of *Automotive* or *Avionics*, when we take care of our health in the hospital or medical office we may use services of *Health and Medical Equipment*, when we decide to relax at home we probably use services of *Consumer Electronics and Intelligent Homes*, and even when we are taking care of our future investing on stock exchange we are using services of *Telecommunications*. The following describes a bit more some of the domains of embedded systems:

Automotive: It includes electronic control units in chassis systems power train electronics, body electronics/security systems, information and computing systems, e.g. for traffic control.

Avionics/Aerospace: It schedules and monitors the takeoff and landing of planes, make it fly, maintain its flight path, and keep it out of harm's way. It includes commercial aircraft, military aircraft, and satellite systems.

Industrial Automation: It includes manufacturing and process controls, motion controllers, intelligent Homes, operator interfaces, robotics, HVAC and other controls, e.g. for energy distribution.

Telecommunications: It provides us with up-to-date information, such as stock quotes. It includes infrastructure, services and end services.

Consumer Electronics and Intelligent Homes: It entertains us with electronic games and joy rides. It includes set-top boxes, Internet access devices, home audio/video, and household appliances.
Health and Medical Equipment: It includes patient monitoring equipment, medical therapy equipment, diagnostic equipment, imaging equipment, and surgical systems.

Unlike PCs and workstations that execute regular non-real-time general purpose applications, such as our editor and network browser, the computers and networks that run embedded real-time applications are often hidden from our view.

Actually, embedded systems are heading more and more towards networked. Rapid advances in microelectronic technology coupled with integration of microelectronic radios on the same board or even on the same chip has been a powerful driver of the proliferation of a new kind of Networked Embedded Systems (NES) over the last decade.

Concurrently, embedded systems are becoming smaller and smaller. In reality, while previously sensors were directly connected to the central computing elements (mostly in an analogue way), today, they are becoming embedded systems themselves. Sensors have a processor included and do some preprocessing of the measured physical property (like temperature, displacement, pressure, etc.) sending the results of this preprocessing to a central management subsystem via a digital network.

We are not intended to establish a new embedded systems taxonomy, but in order to be coherent our philosophy is one of defining embedded systems which in turn can be subdivided in 2 categories: Stand-alone embedded systems, networked embedded systems.

2.1 Stand-alone embedded systems (SES):

Work in a stand-alone mode taking input and producing the desired output. The last two decades have witnessed a significant evolution of stand-alone embedded systems from being assembled from discrete components on printed circuit boards, even if, they still are, to systems being assembled from Intellectual Property (IP) components which are assembled onto silicon of the system on a chip (SoC). SoCs offer a potential for embedding complex functionalities, and to meet demanding performance requirements of applications such as DSPs, network, and multimedia processors.
Most of Consumer Electronics (CE) devices are classified as SES. For instance, the explosion of the CE market over the past decade has generated products mainly in three categories:
Low-end devices generally are built around application specific hardware like ASICs or System-on-Chip (SoC) with small amounts of program memory (ROM), usually around 256 kbytes. They normally use an inexpensive processor, are manufactured in high volumes, and are in general developed by a single programming team. Representative examples of this type of device include many digital cameras and inkjet printers.
Mid-range consumer devices, such as video cameras, are characterized by moderate amounts of program memory like 1 to 2 Mbytes and multiple programming teams.

High-end devices, such as smart phones and set-top boxes, usually have much more memory, up to 32 Mbytes. In most cases, they use powerful processors and are developed by large programming teams.

2.2 Networked embedded systems (NES)

Networked embedded systems may come in many different forms. These systems have been variously referred as EmNets (Network Systems of Embedded Computers) [1], NEST (Networked Embedded System Technology) [2], and DRE (Distributed Real-Time Embedded) [3]. Fundamentally, networked embedded system is a collection of spatially and functionally distributed embedded nodes interconnected by means of wireline or wireless communication infrastructure and protocols, with some sensing and actuation elements interacting with the environment, and, maybe, a master node which is responsible for some control and coordination functions, to organize computing and communication in order to achieve certain goal(s) [1]. The networked embedded systems appear in a variety of application domains such as, automotive, train, aircraft, office building, and industrial — mainly for monitoring and control.
In order to be in accordance to the different forms that Networked Embedded Systems may come, in this survey we consider four types of networked embedded systems, based on what is proposed in [4]: Embedded Systems, Sensor Systems, and Distributed Real-Time and Embedded.

Embedded Systems are systems where the computing components are embedded into some other purpose built device (an aircraft, a car, or a home). Here the characteristic is that these systems are usually not mobile and often not all devices are connected, usually with only one other server machine and most of the time not to external networks. The type of the connection is often wired.

Sensor Systems are most of the time composed by a large number of possibly tiny devices having a single task which is monitoring some conditions within an environment and report back to a central server. The most widespread sensor networks are usually not mobile but the sensors are connected through a wireless network. Wireless sensor networks are a widely deployed example of networked embedded systems. There is a great interest from both the industry and academia in wireless sensor networks technologies that enable deployment of a wide range of applications, such as military, environmental monitoring, e-health applications, etc.

Distributed real-time and embedded systems play an increasingly important role in modern application domains, including military command and control, avionics and air traffic control, and medicine and emergency response. Distributed real-time and embedded (DRE) outline a computational infrastructures of many large scale mission-critical domains where are used to control a variety of artifacts across a number of sites.
DRE systems can be characterized by the fact that the right answer delivered too late becomes the wrong answer. In life-critical military DRE systems, such as those defending ships against missile attacks or controlling unmanned combat air vehicles through wireless links [5], to provide the right answer at the right time is crucial.

3. Requirements of Embedded Systems

In software engineering, functional requirements specify specific behavior or functions of a software system or its component. In general, functional requirements define what a system is supposed to *do*. They are supported by non-functional requirements, which specify criteria that can be used to judge the operation of a system, rather than specific behaviors. In general, non-functional requirements define how a system is supposed to *be*. Non-functional requirements are often called qualities of a system. A system for Cardiac tele-monitoring may be required to present the medical center with a display of data such as heart rhythm. This is a functional requirement. How recent this data needs to be is a non functional requirement. If the data needs to be updated in real time, the system architects must ensure that the system is capable of updating the displayed data within an acceptably short interval of the data changing.

Traditional, embedded software can be quite complex and have a number of requirements. These have implications both for the application and for the software infrastructure, such as the operating system. According to [6], and ***Computer Science and Telecommunications Board [1]*** embedded software have several common features such as the following:

(i) *Resource-constrained computing*. They are frequently rigorously constrained regarding available resources. Especially because of the constraints of cost and size which are duo to mass production and strong industrial competition, resources such as CPU, memory, devices have been designed to meet these requirements. In addition, especially for mobile or autonomous embedded systems energy is a priceless resource. As a result of these restrictions, the system needs to efficiently use its computational resources. For instance, the operating system must be able to operate in resource-constrained environments.
(ii) *Real-time requirements*. Because many embedded applications interact deeply with the real world, they often have strict real-time requirements. These applications require functionalities such as process control, multimedia processing, instrumentation, and so on, where the system has to fulfill a temporal requirement, or deadline. Deadlines are qualified as soft, firm, or hard. Depending on the kind of deadline, the methods to guarantee that a certain deadline is met are different. Occasionally, *preemption points* need to be inserted in critical execution paths in order to reduce scheduling latency.
(iii) *Portability*. Many different types of CPUs, peripheral chips, and memory architectures may be used in embedded systems. Thus, for low cost, any embedded OS or other reusable component that is meant to be used on multiple applications should be commonly portable to custom hardware platforms.
(iv) *High reliability*. Embedded systems are deployed remotely, often in infrastructure-critical applications. Software faults are thus very problematic and are extremely expensive or even impossible to fix.
Stand alone (SES) high-end devices in general use relatively expensive high-end processors, and frequently also coprocessors, that deliver high throughput. As these high-end devices usually have a high degree of human interaction, and users are slow enough, high-end processors in these devices usually have no problem keeping up. For that reason, the real-time requirements are not particularly demanding. In contrast, SES lower-end devices in general have relatively little human interaction and on the contrary consist largely of processes whose timing needs to be tightly controlled. For example, pressing the Digital Camera(DC) shutter starts a series of threads that might involve

tasks such as measuring the ambient light, focusing the camera, capturing the image on the charge-coupled device (CCD), moving the image to memory, etc. These tasks all must be completed before the exposure period calculated by the DC has elapsed and the shutter closes. High-end devices often have special-purpose hardware, such as the Digital Signal Processors (DSP) used in Set Top Boxes (STB). In contrast, lower-end devices do not allow having dedicated hardware in order to complete tasks such as IP routing and video decoding. As they can not remove the responsibility for that tasks from the OS, they must rely on the Operating System real-time performance.

As embedded systems get networked like NES, the scenario gets a little more complex and while some common requirements become more stringent other new requirements also become very important to provide, particularly regarding some qualities the system must attend. Now, support for many of the important attributes for mobile, embedded, sensor and distributed real-time embedded systems are extremely important. Such attributes include requirements that are common in distributed systems as listed in [7]. Also includes other requirements that are relevant to NES [1,3,4,6]. Based on those, it is presented a description of the non-functional requirements which we think represent qualities that are very important to be supported on NES systems. This means that it is important to software infrastructure (operating system, middleware, or else) be able to provide these qualities in order to support these systems. The list includes the following requirements: dependability, robustness, stability, failure handling, safety, security, privacy, scalability and upgradability.

(a) Dependability, Robustness and System Stability
The notion of dependability includes aspects of reliability and availability. Reliability and availability relate to the probability of working continuously for a given duration and the percentage of uptime, respectively. For safety critical real-time applications, reliability is a key concern. This means that the software infrastructure needs to provide built-in support for redundancy management to guard against component and link failures. Robustness is the ability of the system to perform acceptably when the system operates outside of nominal conditions. For example, the ability to create disjoint routing paths may improve robustness by eliminating certain common failure modes. Stability is concerned with the system's ability to keep disruptions contained. Both of these quality attributes are critical to the success of software infrastructures that support networked and distributed real-time and embedded applications. How to specify, identify and validate these attributes is an important research topic, particularly when disruptions are caused by software faults or malicious attacks. The rate of software faults has far exceeded the rate of hardware faults.

(b) Failure handling
In a network embedded system, nodes may fail or experience problems due to several reasons including physically broken, environmental obstruction, unreliable transmission medium, presence of undetected collisions, increased and unpredictable delay, high packet loss, etc. The failure of individual nodes should not affect the overall task of the network embedded system, thus leading to an increased need for providing mechanisms to ensure fault tolerance to the applications. In addition, after deployment of a network embedded system topology changes are likely to occur because of changes in the location of the sensor nodes. Also, extra nodes can be added at any time to replace other nodes and some nodes can stop functioning due to lack of power. Even in a continuously changing network topology, the software infrastructure such as operating

system or middleware system should be able to perform its tasks and provide reliable services to the application.

(c) Safety

This property is concerned with the prevention of the loss of life and/or serious damage to people, property or the environment. This is straightforward for medical devices, but the characteristic is also valid for an aircraft, a car, etc. For example, many medical dispensing devices are safety critical. They are currently certified as stand alone devices under a specific set of application contexts. Sharing of resources in the network compounds the challenges of safety. When we connect systems with different degrees of criticality together and let them share resources, the development of a certifiably safe software infrastructure becomes a serious challenge. If we connect several of them together and something goes wrong, how can we define and identify which device is responsible? Technologies for isolation and protection across all the layers, including software infrastructure like OS, along with controlled graceful degradation of service can arise as an important research topic.

(d) Security

This property is concerned with the capability to prevent information and system resources from being used or altered by unauthorized users. A secure system is one where only intended use of the system will be permitted. This also means avoiding unpermitted access or modification. In addition to cyber attacks, there are unique challenges from the perspective of embedded systems. Unlike office computers, embedded monitoring devices are often left unattended for long periods of time in remote areas and may be subject to tampering. For example, compromised unattended physical sensors may feed the system with false data while the device remains authenticated and the transmitted data encrypted properly. It is also easy to manipulate the environment to fool many forms of sensors such as temperature, chemical concentration, electromagnetic and acoustic sensors.

(e) Privacy

Privacy is concerned to the ability of an individual or group to isolate themselves or information about themselves and thereby reveal selectively. Privacy is sometimes related to anonymity, the wish to remain unnoticed or unidentified in the public realm.
The pervasiveness of networked and distributed real-time embedded systems brings the difficult issue of providing privacy. Privacy policies are and will be increasingly specified at different levels of detail and have different semantics for different systems. Techniques for specifying and checking the consistency of policies across separate systems are needed. It is very important to consider the impact of privacy solutions on the software infrastructure, including services such as addressing, routing, time-stamping, encryption and avoidance of communication patterns that can reveal private information.

(f) Scalability

This property indicates the system ability to either handle growing amounts of work in a graceful manner, or to be readily enlarged. Compared to existing typical distributed systems, whose components communicate either with fixed or with mobile connections, the number of nodes in a network embedded system can be several orders of magnitude higher than the nodes in a traditional distributed system. In the future, sensor and embedded devices will be used at ultra-large scales, including millions of devices

connected over the Internet. For example, world-wide weather monitoring based on millions of new sensor devices is being contemplated. As a consequence, it is important that new software infrastructure approaches are designed to support such a large number of devices that interact across large geographic areas. In addition, be able to manage such a large-scale network embedded system that spans multiple administrative domains becomes a major challenge. Also, with scalability, the issues of self-calibration, self-configuration and "self-healing" become crucial.

(g) Upgradability

This property is concerned with the degree to which a computer may have its specifications improved by the addition or replacement of components. Traditionally, most embedded devices, once deployed, have rarely been upgraded. In a world of networked embedded systems, upgrades will be more frequent and often far more invisible to end users of the systems. For instance, an NES may be in service for many years, and the environment to which they are connected and the functionality requirements for the device may change considerably over that time. In some cases, such upgrades are driven by a knowledgeable user, who purchases a new component of functionality and installs it, a nearly automatic procedure. In other cases, updates or upgrades may be invisible to the end user, such as when protocols or device addresses change. Most system designers will expect the operating system to make the task easier and to handle some difficult problems like upgrade policy, verification, and security. Furthermore, in some cases the operating system itself may need to be field upgraded, a process that almost certainly requires operating system cooperation and that extends beyond the device being updated. There is no consensus on how online field upgrade will work for the billions of networked embedded systems components that will be deployed. Field upgrade is likely to become an important focus of research and development work over the next several years as numerous systems are deployed that challenge the ability of simple solutions to scale up to adequate numbers and reliability.

EmS/Req.	(*i*)	(*ii*)	(*iii*)	(*iv*)	(*a*)	(*b*)	(*c*)	(*d*)	(*e*)	(*f*)	(*g*)
SES											
Low-end	M	M	M	M	W	W	M	W	O	O	W
Med-end	W	W	M	M	W	W	W	W	O	O	W
High-end	O	W	M	M	W	W	W	W	W	O	W
NES											
Embedded	M	M	M	M	M	M	M	W	W	W	W
Sensor	M	W	M	M	M	W	W	M	W	M	M
Distributed	W	M	M	M	M	M	M	M	M	M	M

Table 3.1 Requirements for different categories of Embedded Systems

Table 3.1 summarizes the different categories of embedded systems and what properties (requirements) are wish to be found in each of them. In order to establish the importance of the requirements to the various embedded systems they are classified as *mandatory* (M) in case you must fulfill the requirement, *wanted* (W) when it is advantageous to have it, *optional* (O) when have it or not is not a big issue.

4. Software infrastructure for embedded systems

Embedded systems have been around at least as long as the microprocessor. The software for these systems has been built, more or less successfully, using several different paradigms. Some systems are built from scratch by the manufacturer with all software being created specifically for the device in question. This software may be written in assembly language or may use a higher-level language. Not all components of embedded systems need to be designed from scratch. Instead, there are standard components that can be reused. Software infrastructure, especially standard software components such as operating systems (OS), are examples of these reusable software components. Such components are available from independent software vendors and in some cases as open source software.
Operating systems, interface with hardware to provide the necessary services for application software. Software infrastructure is responsible for providing application software with services that allow them to fulfill the requirements.

The rapid progress in processor and sensor technology combined with the expanding diversity of application fields is placing huge demands on the facilities that an embedded operating system must provide. The variety of applications where embedded systems are being important also implies that the properties, platforms, and techniques on which embedded systems are based can be very different. All the types of embedded systems, from stand-alone embedded systems-SES, including the variety of devices and applications such as digital camera, set-top boxes, and smart phones, to Networked embedded systems-NES, including mobile systems, embedded systems, sensor systems and distributed real-time and embedded system, need some specific type of service from software infrastructure such as operating system or middleware. For instance, these services are supposed to be prepared to attend to functional and non-functional requirements. In this section we give an overview of presently available software infrastructures for embedded systems, particularly regarding operating systems.

4.1 Operating Systems for Embedded Systems

Not all embedded systems need to be supported by operating system functionality the simplest embedded systems are usually built without an explicit operating system. Such systems do not have behaviors that require complex mechanisms or real-time scheduling of concurrent tasks, and can therefore be implemented using a simple main loop or executive cyclic approaches. Also, dedicated networked embedded systems can usually be implemented without an operating system, but only with a stand-alone protocol stack. One example of a networked system that does not require an operating system could be a networked temperature controller that supplies temperature data over TCP/IP. The controller will only be running a single application and there is no need to have an operating system for dealing with multiple threads of control. For instance, many embedded TCP/IP stacks such as uIP [8], lwIP [9] and OpenTCP [10], have the ability to run either with or without a supporting operating system.

Embedded systems typically have requirements that are a lot different from the ones for desktop computers, and hence operating systems for embedded systems are diverse from general purpose operating systems (GPOS). Operating systems for embedded systems usually are designed to be tailored for a specific application and therefore are more static than GPOS. Many embedded systems require real-time guarantees to

function correctly, and most operating systems for embedded systems has real-time properties such as guaranteed response times, deterministic computations, and real-time scheduling algorithms. Due to the huge variety of embedded systems, there is also a huge variety of requirements for the functionality of embedded OSes. However, it is not effective to have an OS providing all the required functionalities. Besides, as most embedded systems are application specific, we need operating systems which can be flexibly customized towards the application at hand. So, configurability is one of the main characteristics of embedded OSes. In OS, configurability deals with kernel extension, as well as kernel customization and kernel adaptation [11]. In addition, (re) configuration can be related to fine-grain or coarse-grain components.

Conventionally, existing operating systems for embedded systems are divided into two categories: embedded operating systems and real-time operating systems [12]. In this report we assume that the purpose of the OS can be divided in two categories: real-time embedded systems that can also be called general embedded systems, and domain specific embedded systems which somehow provide specific characteristics for different domains of embedded systems like automotive, avionics, mission critical systems and wireless sensor networks systems, without loosing OS functionality.

4.1.1 Embedded and Real-Time Operating Systems

In many embedded systems there is effectively no device that needs to be supported by all versions of the OS, except maybe the system timer. Hence, it makes sense to handle relatively slow devices such as discs and networks by using special tasks (drivers) which are not integrating the kernel of the OS. Protection mechanisms are not always necessary, since embedded systems are typically designed for a single purpose and there is no well defined separation between application and OS functionality. For instance, in contrast to desktop applications, there is no desire to implement I/O instructions as privileged instructions and tasks can be allowed to do their own I/O. This matches nicely with the previous item and reduces the overhead of I/O operations. There is no need to go through an OS service call, which would create a lot of overhead for saving and restoring the task context (registers etc.). In addition, taking into account that embedded programs can be considered to be thoroughly tested, protection is not necessary, and it is required efficient control over a variety of devices, it is possible that interrupts can be employed to directly start or stop tasks. This is substantially more efficient than going through OS services for the same purpose. Moreover, many embedded systems are real-time (RT) systems and, hence, the OS used in these systems must be a real-time operating system (RTOS). After all, it is possible to say that features such as configurability, portability, real-time and reliability are very desirable in embedded OS.

Traditional embedded operating systems like VxWorks [13], WinCE [14], QNX [15,16], PalmOS [17], OS-9 [18], LynxOS [19], Symbian [20], are typically large (requiring hundreds of KB or more of memory), general-purpose systems consisting of a binary kernel with a rich set of programming interfaces. Such OSes target systems with greater CPU and memory resources, and generally support features such as full multitasking, memory protection, TCP/IP networking, and POSIX-standard APIs that are undesirable (both in terms of overhead and generality) for sensor network nodes. For example, a QNX context switch requires over 2400 cycles on a 33MHz 386EX

processor, and the memory footprint of VxWorks is in the hundreds of kilobytes. They provide memory protection given the appropriate hardware support. This becomes increasingly important as the size of the embedded applications grow. In addition to providing fault isolation, memory protection prevents corrupt pointers from causing seemingly unrelated errors in other parts of the program allowing for easier software development. VxWorks, WinCE and QNX are well ranked in the 2005 survey by Embedded System Design [21].

The *VxWorks* commercial RTOS from Wind River is the most widely adopted in the embedded industry (e.g., it is used on the International Space Station). VxWorks was first released in the early 1980s and provides a flexible API with more than 1800 methods, including network support, file system and I/O management. VxWorks provides the Wind River Workbench which is a collection of Eclipse-based tools that accelerates time-to-market for developers building devices with VxWorks. It is available on all popular CPU platforms. The development host can be Red Hat Linux, Solaris, SuSE Linux, Windows 2000 Professional, or Windows XP and provides a visual development environment. The kernel supports preemptive priority scheduling with 256 priority levels and round-robin scheduling. VxWorks is a multithreading RTOS that provides deterministic context switching and supports semaphores and mutual exclusion with inheritance. It can be set up so that each task has a private virtual memory upon request. This RTOS also provides message queues and Open-standard Transparent IPC for high-speed communications between threads. VxWorks architecture is showed in Figure 4.1.

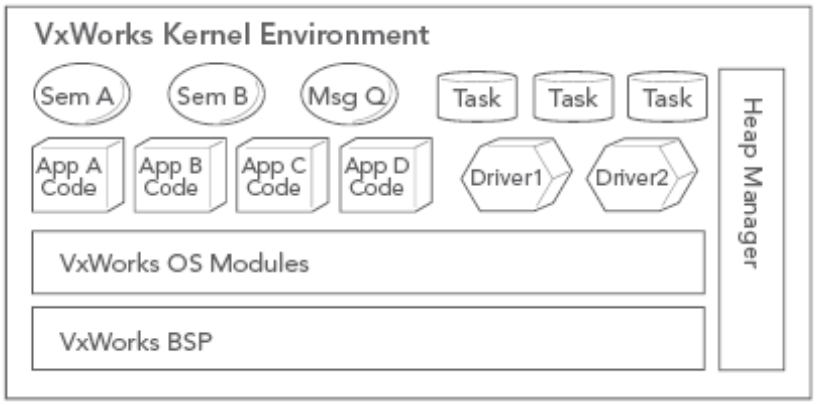

Figure 4.1 Architecture of VxWorks
http://www.windriver.com/products/product-notes/General-Purpose-Platform-ve-Note.pdf)

QNX is based on the idea of running most of the OS as a number of small tasks, known as *servers*. This differs from the more traditional monolithic kernel, in which the operating system is a single very large program composed of a huge number of "parts" with special abilities. QNX Neutrino (2001) has been ported to a number of platforms and now runs on practically any modern CPU that is used in the embedded market. It also provides QNX System Tools with a System Builder for offline and online configuration of the system, and some analysis tools. The QNX kernel contains only CPU scheduling, inter-process communication, interrupt redirection and timers. Everything else runs as a user process. QNX Real-time capabilities include priority-based preemptive scheduling with 256 priorities, aperiodic scheduling with sporadic server, nested interrupts and inheritance protocol. Figure 4.2 shows the Architecture of QNX OS.

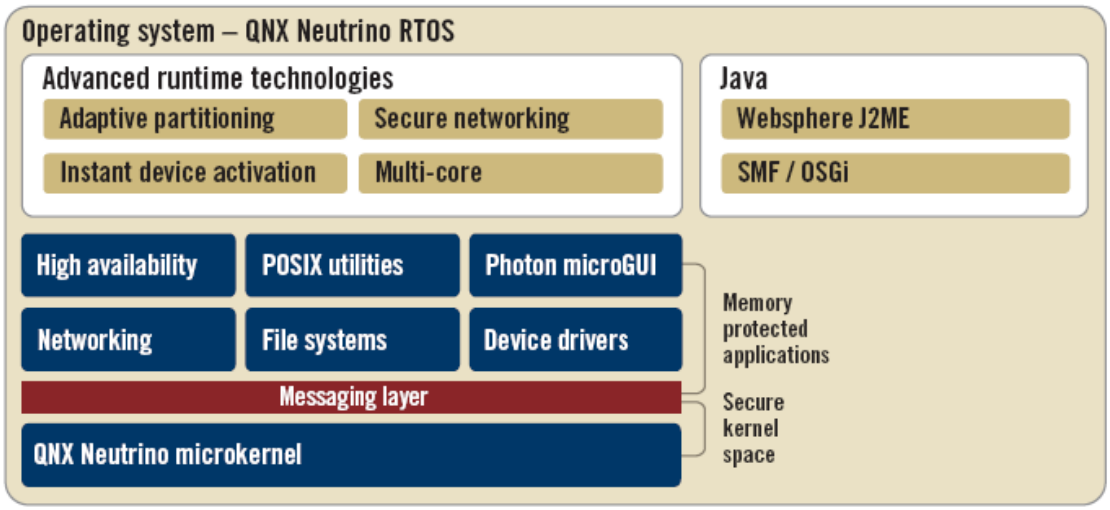

Figure 4.2 QNX Neutrino Architecture
http://www.qnx.com/download/feature.html?programid=8483

The *Windows CE* RTOS is a commercial RTOS developed in the late 1990s by Microsoft. Windows CE is a modular, portable real-time embedded OS developed especially for small memory, mobile 32-bit devices. There exist three main development platforms (Windows Mobile, SmartPhone, and Portable Media Center) that allow developers to use feature-rich tools to develop applications for x86 and other architectures. Windows CE can have up to 32 processes active with multiple threads in each process. The scheduler supports round-robin or priority-based preemptive scheduling with 256 priority levels, and uses the priority inheritance protocol for dealing with priority inversion. Windows CE supports OS synchronization primitives such as critical sections, mutexes, semaphores, events, and message queues to allow thread to control access to share resources. Architecture of Windows CE is showed in Figure 4.3.

Several embedded systems have taken a component-oriented approach for application-specific configurability [22]. eCos [23][24] and icWORKSHOP [25] have a goal of lightweight, static composition. These systems consist of a set of components that are wired together (either manually or using a composition tool) to form an application. Components vary in size from fine-grained, specialized objects (as in icWORKSHOP) to larger classes and packages (eCos). VEST [26] is a proposed toolkit for building component-based embedded systems that performs extensive static analyses of the system, such as schedulability, resource dependencies, and interface type-checking.

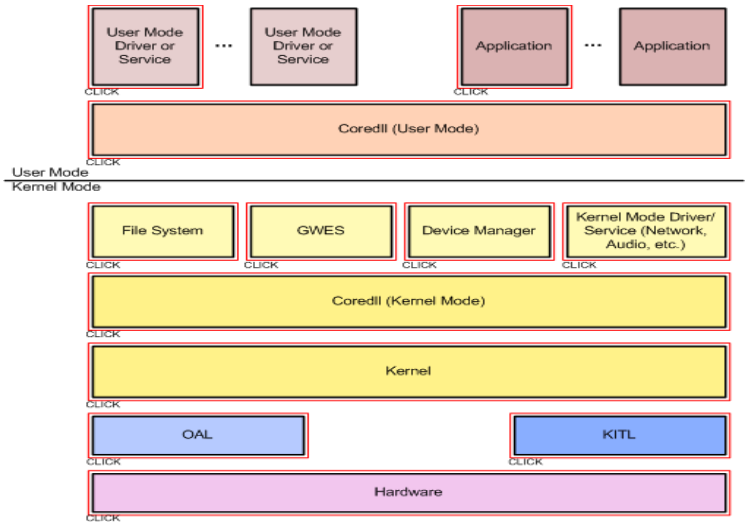

Figure 4.3 Architecture of Windows CE
http://msdn.microsoft.com/pt-br/library/aa924061.aspx

The most widely adopted free, open source RTOS, *eCos* (embedded Configurable operating system) was released in 1986. eCos provides a graphical-configuration tool and a command line-configuration tool to customize and adapt the RTOS to meet application-specific requirements. This feature allows the user to set the OS to the desired memory footprint and performance requirements. Development hosts are Windows and Linux and the supported target processors are various. The eCos kernel can be configured with the bitmap scheduler or the multilevel queue (MLQ) scheduler. Both schedulers support priority-based scheduling with up to 32 priority levels. The bitmap scheduler is somewhat more efficient and only allows one thread per priority level. The MLQ scheduler allows multiple threads to run at the same priority. First in, first out (FIFO) or round-robin is used to schedule threads with the same priority. The eCos RTOS supports OS primitives such as mutexes, semaphores, mailboxes, and events for synchronization and communication between threads. eCos architecture is showed in Figure 4.4.

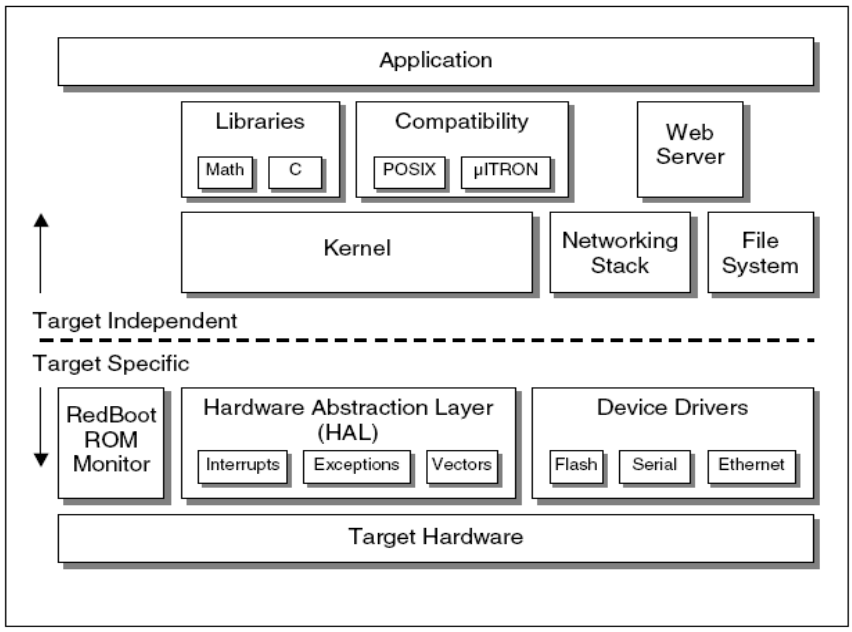

Figure 4.4 Architecture of eCos [24]

There is also systems that are specially design for smaller embedded systems. family of smaller real-time executives, such as CREEM [27], pSOSystem [28], OSEKWorks [29], and Ariel [30], that while providing support for preemptive tasks, they have severely constrained execution and storage models.

Contemporary OS such as Linux, also have extensions that enable them to support real-time applications: RTAI [31], and RT-Linux [32]. These RTOSs are based on the same principle of operation which is through interrupt control between the hardware and the operating system. Interrupts needed for deterministic processing are processed by the real-time core, while other interrupts are forwarded to the non-real time operating system. These systems are only suitable for large real-time systems due to footprint required and their architecture are similar as in Figure 4.5.

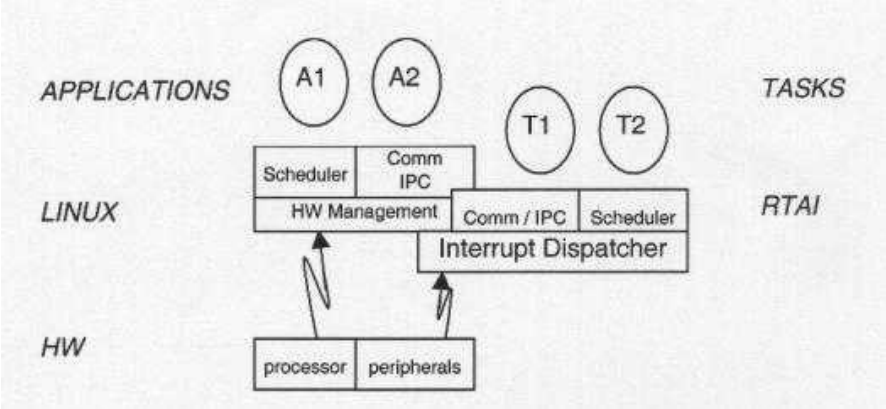

Figure 4.5 RTAI Architecture [31]

Other Linux extensions like Embedded Linux [33] and µClinux [34] are more embedded compliant but do not support real-time. These extensions have an architecture which is very much like the Linux architecture in Figure 4.6 except they do not have to deal with memory management units (MMU).

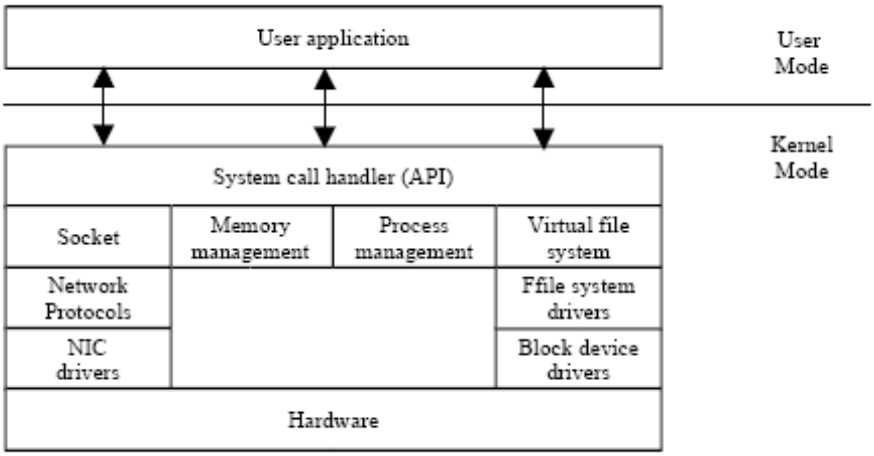

Figure 4.6 Linux architecture

As embedded systems are getting more and more complex, dynamic, and open, while interacting with a progressively more demanding and heterogeneous environment, the

reliability and security of these systems have become major concerns. An increasing number of external security attacks as well as design weaknesses in operating systems have resulted in large economic damages, which results in difficulties to attain user acceptance and getting accepted by the market. Consequently, there is a growing request from stakeholders in embedded systems to make available execution platforms which address both integrity and security concerns. Recently, we have seen some proposals which are really concern with security, reliability, dependability and resilience of operating systems [35,36,37,38]. Basically, they propose to use micro-kernel based operating systems [73, 74] to provide security and dependability for embedded system application. One of the proposals is based on L4 OS [36, 38] which architecture is showed in Figure 4.7. Another proposal is based on Minix OS [35, 37]. Figure 4.8 shows the architecture of Minix OS.

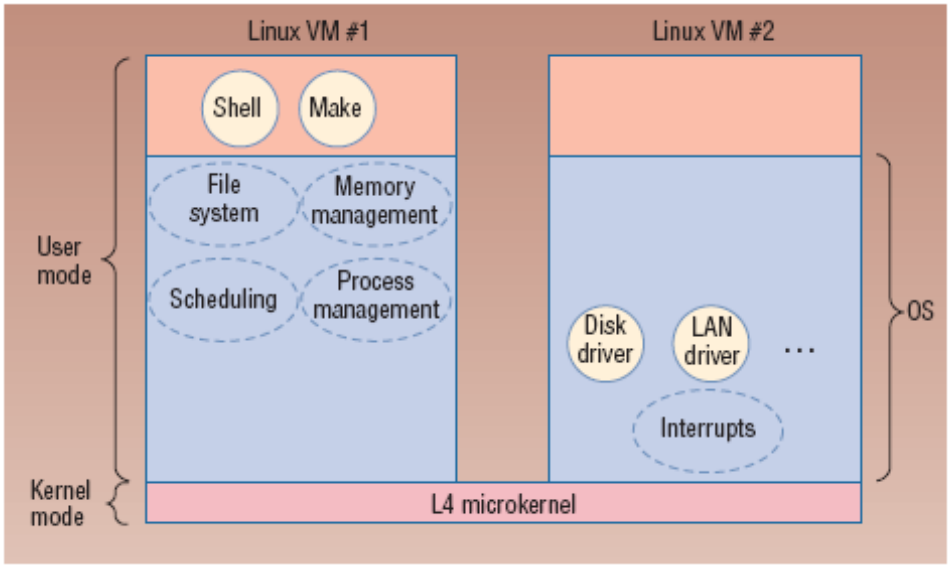

Figure 4.7 L4 based Architecture [35]. The resulting system comprises a microkernel that also provides for virtual machines and a microkernel-based system on top that includes components for maintaining the virtual machines.

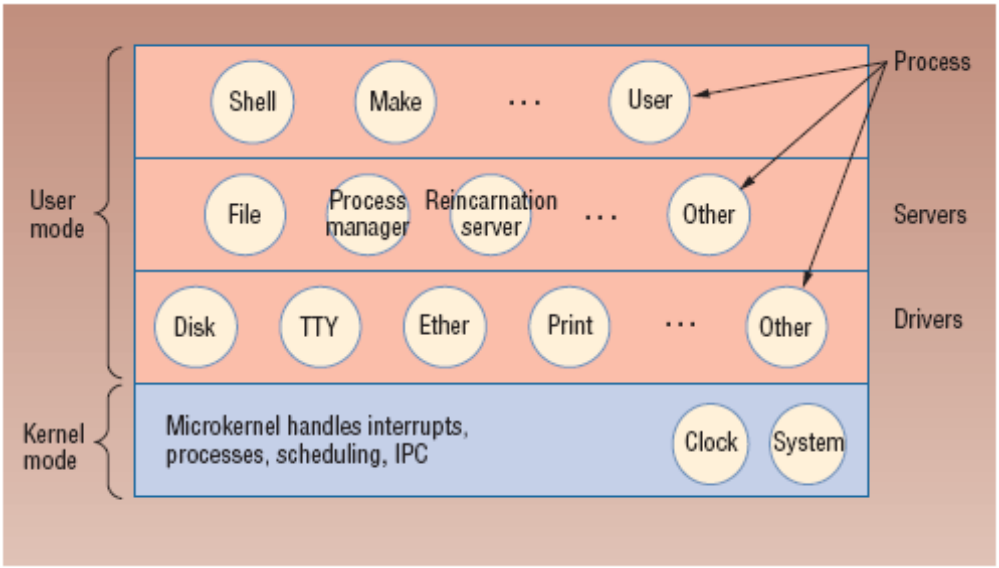

Figure 4.8 Minix Architecture [35].The microkernel handles interrupts, provides the basic mechanisms for process management, implements interprocess communication, and performs process scheduling.

Table 4.1 summarizes some operating systems for embedded systems and what properties (requirements) they provide. The table refers to the same set of requirements we have introduce earlier at chapter 3. In this table we want to give you a view of how the systems provide and enforce a certain property: *adequately* (A) in case the system provides and enforces the property, *poorly* (P) when the system provides the property but not as required (for example, when the system is said to be real-time but do not support hard real-time behavior), and *non* (N) when the system does not provide the requirement.

OS/Req.	**(*i*)**	**(*ii*)**	**(*iii*)**	**(*iv*)**	**(*a*)**	**(*b*)**	**(*c*)**	**(*d*)**	**(*e*)**	**(*f*)**	**(*g*)**
VxWorks	P	P	A	A	P	P	P	A	P	A	P
QNX	P	P	A	A	P	P	P	A	P	A	P
Windows CE	P	P	A	P	N	N	N	P	P	P	N
eCos	A	P	A	P	N	N	P	P	P	A	N
pSOS	A	P	P	P	N	N	P	P	N	P	N
RTAI	N	A	P	P	N	N	N	P	N	P	N
uClinux	A	N	A	P	N	N	N	P	N	P	N
L4	P	P	P	A	A	A	P	A	A	A	P
Minix	P	N	P	A	A	A	P	A	A	A	P

Table 4.1 Embedded Systems Requirements provided by OSes for EmS

Most of the OSes for embedded systems provide the necessary functionalities such as multitasking, memory management, file system, network, etc., through its API. They provide various architecture approaches such as monolithic kernel, micro-kernel based, components based, and library based. However, they do not provide and enforce a lot of non-functional requirements that are necessary in embedded systems, specially the properties usually required in networked embedded systems. Also, most of the OS we listed required large amount of memory to run and do not deal with low power consumption. According to [21], the type of operating system that embedded systems developers pick in almost half the cases (44%), is one of the many commercial operating systems or RTOS products for their current project. The remainder was about equally divided among internally developed operating systems, open-source operating systems, and no operating system at all.

4.1.2 Domain Specific Embedded Operating Systems

This section presents three domains, Avionics, Automotive and Embedded Sensor Networks which needs specific requirements.

Avionics

Significant work has been performed within the avionics domain to achieve the stated objectives. The main body of work has been performed under the banner Integrated Modular Avionics (IMA) [39,40,41].

The ARINC 653 [42,43] is a standard that specifies a programming interface for a RTOS. In addition, it establishes a particular method for partitioning resources over time and memory. The ARINC 653 specification for system partitioning and scheduling is often required in safety- and mission-critical systems, particularly in the avionics industry. ARINC 653 defines an APplication EXecutive (APEX) for space and time

partitioning that may be used wherever multiple applications need to share a single processor and memory, in order to guarantee that one application cannot bring down another in the event of application failure. Each partition in an ARINC 653 system represents a separate application and makes use of memory space that is dedicated to it. Similarly, the APEX allots a dedicated time slice to each, thus creating time partitioning. Each ARINC 653 partition supports multitasking. Applications that use the ARINC 653 application programming interface (API) can be more easily ported from one ARINC 653 operating system to another than those which do not.
ARINC standards allow aircraft manufacturers to ensure that new installations are compatible and interchangeable. A study called "The Economic Impact of Avionics Standardization on the Airline Industry," from Georgia State University's Aviation Policy Research, Aviation and Transport Studies, estimates annual savings by the airlines industry of more than $291 million annually through the use of ARINC standards.

The MILS –Multiple Independent Levels of Security and Safety [44] approach is proposed to provide a reusable formal framework for high assurance system specification and verification. Separation of kernel is the big issue. The traditional monolithic kernel is intended to provide as many services as possible to the application. Kernels compete with each other based upon the richness of the API that they offer to the programmer. In the MILS architecture, the separation kernel only does four very simple things. A MILS kernel is responsible for enforcing data isolation, control of information flow, periods processing and damage limitation policies, and nothing else. Each of these policies counters one or more of the basic foundational threats to system assurance.
The MILS separation kernel, Figure 4.9, has two special characteristics. First, it is the only code that runs in supervisor or privileged mode. No other code, not even device driver code, has the ability to affect the processor’s protection mechanisms, particularly the MMU.

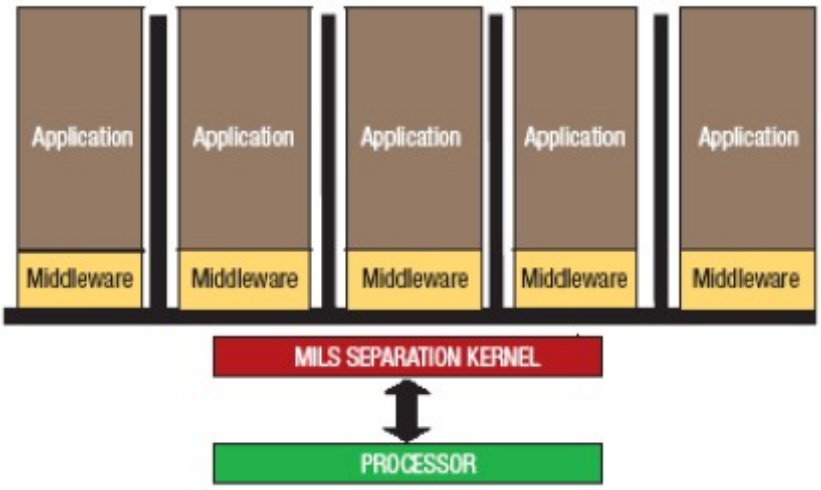

Figure 4.9 MILS Architecture [44]

The second characteristic is that because the separation kernel is so simple it can be very small, approximately 4,000 lines of C language source code. That is small enough to be mathematically modeled so that we can trust this code to rigorously enforce the

four MILS policies. This code will have been proven to be correct under all conditions. Mathematical proof by formal methods is arduous and expensive, but it only needs to be done once for each separation kernel. That investment can be leveraged over again many times.
As examples of MILS compliant OS we have LynxSecure [45] and PikeOS [46]. They have been used especially in military and avionics industries. Another system, RTEMS [47] which is not MILS compliant is also used for military and avionics projects.

LynxSecure addresses this issue on all fronts by providing a robust environment within which multiple secure and non-secure operating systems can perform simultaneously—with no compromise of security, reliability or data. LynxSecure expands on the proven real-time capabilities of the LynxOS® real-time operating system (RTOS) with time-space partitioning and operating-system virtualization. It conforms to the Multiple MILS architecture, with strict adherence to data isolation, damage limitation and information flow policies identified in this architecture. Unlike a traditional security kernel that performs all trusted functions for a secure operating system, a separation kernel's primary security function is to partition data and resources of a system and to control information flow between partitions. Partitions and information-flow policies are defined by the kernel's configuration. This provides a robust foundation for the creation of multi-level secure systems. To fulfill the separation kernel concept of MILS architecture, it utilizes virtualization. LynxSecure uses a hypervisor to create a virtualization layer that maps physical system resources to each guest operating system. Each guest operating system is assigned certain dedicated resources, such as memory, CPU time and I/O peripherals. Figure 4.10 shows the architecture of LynxSecure.

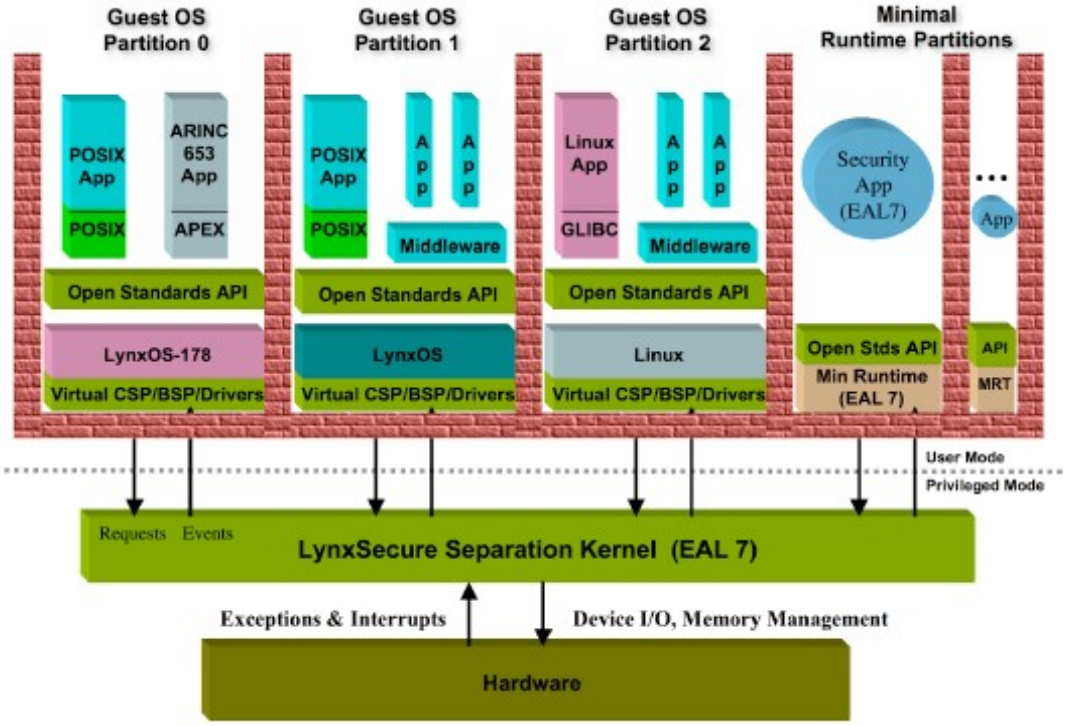

Figure 4.10 LinxSecure Architecture with Guest Operating Systems [45]

PikeOS is a microkernel-based real-time operating system made by SYSGO AG. It is targeted at safety and security critical embedded systems. It provides a partitioned environment for multiple operating systems with different design goals, safety requirements, or security requirements to coexist in a single machine. It was initially modeled after the L4 microkernel [48] and has gradually evolved over the years of its application to real-time, embedded systems space. The goal of Pike OS is to provide partitions that comprise a subset of the system's resources. Processing time is one of those resources. It is expected the partitions to host a variety of guest operating systems with different requirements regarding timely execution. PikeOS combines resource partitioning and virtualization, in order to fulfill the MILS separation kernel concept. Its virtual machine environments are able to host entire operating systems, which need to be adapted in order to run in one of its virtual machine environments, along with their applications that can run unmodified. A number of different operating systems have been adapted to run in a PikeOS virtual machine. Among them are Linux, several popular real-time operating system APIs including POSIX and ARINC-653. PikeOS provides a build-in Health Monitoring Feature which implements all features described in the ARINC-653 standard. Failures like address- and timing violations, illegal instruction will be intercepted by the OS and handled as specified in the system configuration. This adds another layer of determinism without additional application code. The PikeOS system can be configured using PIK, the graphical configuration editor within CODEO. PIK includes a powerful integrity checker that makes it almost impossible to create an invalid configuration. PikeOS supports PowerPC, x86, and MIPS platforms. Examples of projects using PikeOS are DIANA[49] and Airbus for the FSA-NG (Fly Smart with Airbus New Generation). Figure 4.11 shows the architecture of PikeOS.

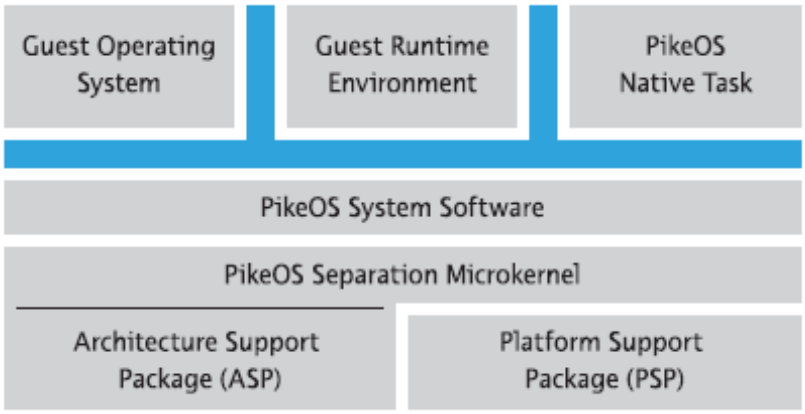

Figure 4.11 PikeOS Architecture

RTEMS (Real-Time Executive for Multiprocessor Systems) is a free open source real-time operating system (RTOS) designed for embedded systems. The acronym RTEMS initially stood for *Real-Time Executive for Missile Systems*, then became *Real-Time Executive for Military Systems* before changing to its current meaning. RTEMS development began in the late 1980s with early versions of RTEMS available via ftp as early as 1993. OAR Corporation is currently managing the RTEMS project in cooperation with a Steering Committee which includes user representatives. Space and
On a conceptual level RTEMS can be characterized by three layers: hardware support, kernel and APIs. The user then develops his application by using the available APIs.

The hardware support layer encompasses the processor and board dependent files as well as a common hardware library. RTEMS provides a notion of executive which encapsulates the API layer and the kernel. The kernel layer is the heart of RTEMS and encompasses the super core, the super API and several portable support libraries. The super core is organized into handlers and provides a common infrastructure and a high degree of interoperability between APIs. It is worth reminding that there is a small part of the super core that is target dependent. The super API contains the code for services that are beyond any standardization, such as API initialization and extensions support. The API layer makes the bridge between the kernel and the application. APIs are implemented in terms of super core services. The Classic API, provides basic features such as multitasking, priority-based pre-emptive scheduling and optional rate-monotonic scheduling, inter-task communication and synchronization, priority inheritance, etc. POSIX and ITRON APIs are also supported. There is also an interface for ARINC 653 specification [50]. Aviation projects supported by RTEMS include Dawn [51] and THEMIS [52] . Figure 4.12 shows the architecture of RTEMS.

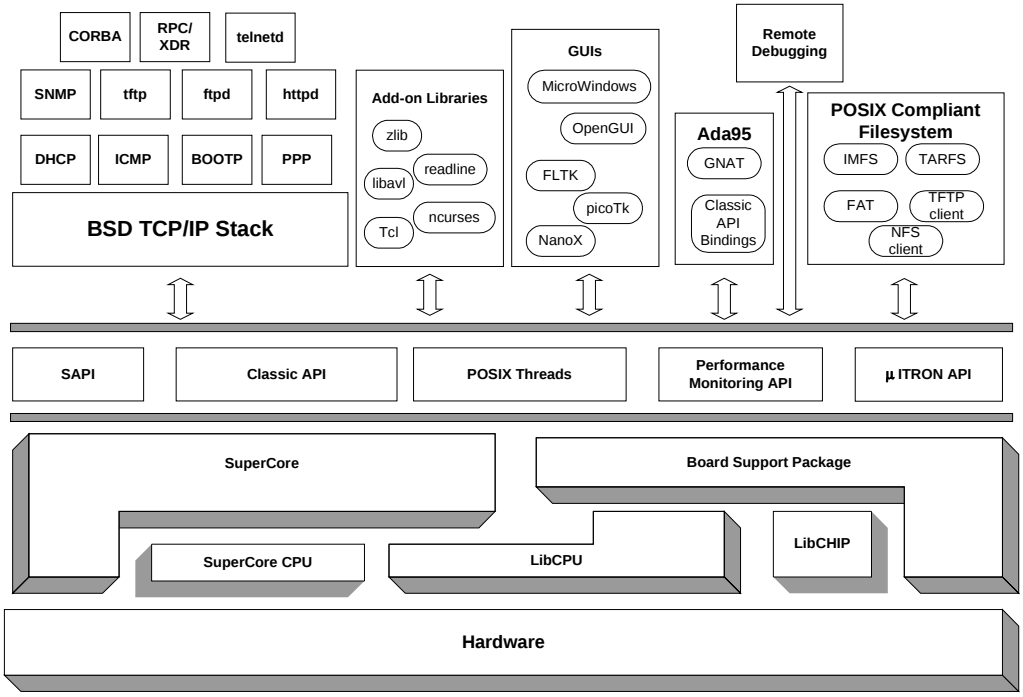

Figure 4.12 RTEMS Architecture

Table 4.2 summarizes some operating systems for avionics domain embedded systems and what properties (requirements) they provide. The table refers to the same set of requirements we have introduce earlier at chapter 3. In this table we want to give you a view of how the systems provide and enforce a certain property: *adequately* (A) in case the system provides and enforces the property, *poorly* (P) when the system provides the property but not as required (for example, when the system is said to be real-time but do not support hard real-time behavior), and *non* (N) when the system does not provide the requirement.

OS/Req.	(*i*)	(*ii*)	(*iii*)	(*iv*)	(*a*)	(*b*)	(*c*)	(*d*)	(*e*)	(*f*)	(*g*)
LynxSecure	P	P	A	A	A	P	A	A	A	A	A
PikeOS	P	P	A	A	A	P	A	A	A	A	P
RTEMS	P	A	A	A	A	P	A	A	P	P	P

Table 4.2 Embedded Systems Requirements provided by OSes for Avionics Domain

All systems tend to be compliant to ARINC and MILS specifications, where the big issue is Security and Safety. However, some basic requirements for embedded systems such as resource constrains and real-time are not adequately handled. Also, it seems that failure handling is not an important concern for these systems.

Automotive

In the automotive domain, the embedded systems are distributed; hence the communications play a key role in the development process all the way from the design, to implementation and integration.

OSEK/VDX [53], which is a collection of widely used standards for automotive systems, specifies a scalable real-time operating system OSEK/VDX OS [53], communications with transparent communication services OSEK/VDX COM [54], and a network manager OSEK/VDX NM [55] allowing for easy integration of subsystems developed by different OEMs. OSEK/VDX provides reusability and portability of software by using abstract high level interfaces. OSEK/VDX COM allows for communications on a high level abstraction, without detailed knowledge on communication transmitters and recipients locations.

The latest automotive software standard is AUTOSAR [56], by the AUTOSAR consortia. The goal of AUTOSAR is to create a global standard for basic software functions such as communications and diagnostics. From an integration point of view, AUTOSAR provides a Run-Time Environment (RTE) routing communications between software components regardless of their locations, both within a node and over networks. Tools allows for easy mapping of software onto the existing architecture of nodes (Electronic Control Units - ECUs). AUTOSAR is working towards integration of standardized tools relying on, e.g., operating system standards such as, e.g., OSEK/VDX OS, and various communication standards as, e.g., OSEK/VDX COM, FlexRay, CAN, LIN and MOST.

OSEK/VDX OS [53], describes the concept of a real-time operating system, capable of multitasking, which can be used for motor vehicles. It also specifies the OSEK operating system API. Figure 4.13 shows the architecture of an OSEK OS.

The objective of the standard is to describe an environment which supports efficient utilization of resources for automotive control unit application software. This standard can be viewed as a set of API for real-time operating system (OSEK) integrated on a network management system (VDX) that together describes the characteristics of a distributed environment that can be used for developing automotive applications. Typical automotive applications are characterized by stringent real-time requirements and high criticality (for example, a power-train application). In addition, these applications have to be made in a huge number of units, therefore there is a need to reduce the memory footprint to a minimum enhancing as possible the OS performance.

The following are some features that help to better characterize the philosophy that drove the main architectural choices of the OSEK Operating System:

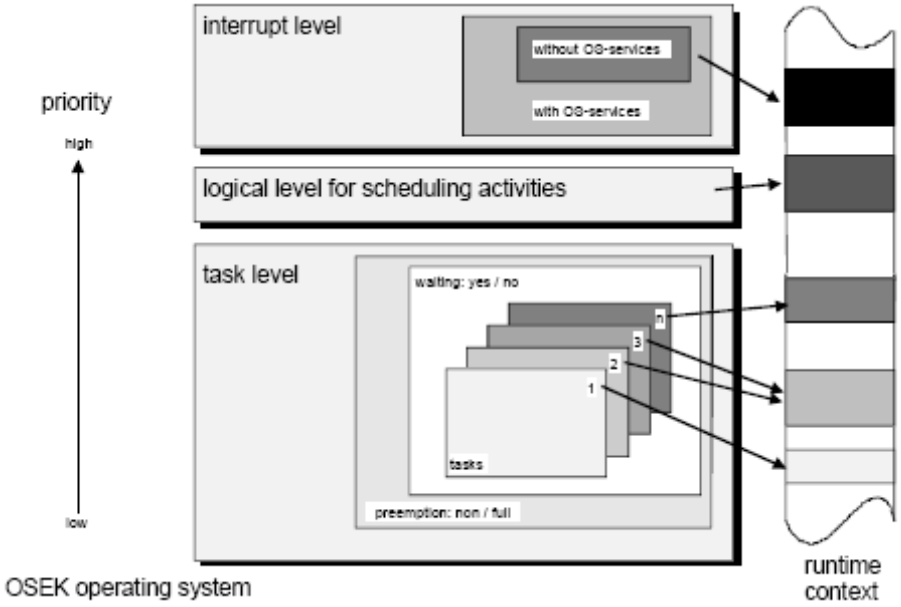

Figure 4.13 OSEK/VDX OS Architecture

Scalability - the operating system is intended for use on a wide range control units (either system require only a minimum of hardware resources RAM, ROM, CPU time and runs even on 8 bit microcontrollers). To support a wide range of systems the standard defines four conformance classes that tightly specifies the main features of an OS. Different conformance classes, various scheduling mechanisms and the configuration features make the OSEK operating system feasible for a broad spectrum of applications and hardware.
Portability of software - the standard specifies an ISO/ANSI-C interface between the application and the operating system that is identical in all the implementations of the OS. The aim of this interface is to give the ability to relocate the application software from one ECU to another ECU without bigger changes inside the application. Due to the wide variety of hardware where the OS has to work in, the standard does not specify any interface for the Input/Output subsystem. Note that this fact reduces (if not prohibits) the portability of the application source code, since the I/O system is one of the main software part that impacts on the architecture of the software. We can say that the prime focus is not to achieve 100% compatibility between the application modules, but to ease their direct portability between compliant operating systems.

Configurability - another requirement needed to adapt the OS to a wide range of hardware is a high degree of modularity and configurability. This configurability is reflected by the tool chain proposed by the OSEK standard, where some configuration tools help the designer in tuning the system services and the system footprint. Moreover, a language called OIL (OSEK Implementation Language) is proposed to help the definition of a standardized configuration information.
Reliability - the OSEK operating system is configured and scaled statically. The user statically specifies the number of tasks, resources, and services required. This approach ease the implementation of an OS capable of running on ROM, and moreover it is

completely different from a dynamic approach followed in other OS standards like for example POSIX.
Real-Time capability - the specification of the OSEK operating system provides a predictable and documented behavior to enable operating system implementations, which meet automotive real-time requirements.
Support for time triggered architectures - the OSEK Standard provides the specification of OSEKTime OS, a time triggered OS that can be fully integrated in the OSEK/VDX framework.

OSEKtime operating system (OSEKtime OS) [57] is the specification of a time-triggered operating system with a fault-tolerant communication layer as a standardized run-time environment for highly dependable real-time software in automotive electronic control units. The operating system must implement the following properties: predictability (deterministic, a priori known behavior even under defined peak load and fault conditions), clear, modular concept as a basis for certification, dependability (reliable operation through fault detection and fault tolerance), support for modular development and integration without side-effects (composability), and compatibility to the OSEK/VDX.

The OSEKtime operating system supports static scheduling and offers all basic services for real-time applications, i.e., interrupt handling, dispatching, system time and clock synchronization, local message handling, and error detection mechanisms. All services of OSEKtime are hidden behind a well-defined API. The application interfaces to the OS and the communication layer only via this API. For a particular application the OSEKtime operating system can be configured such that it only comprises the services required for this application. Thus the resource requirements of the operating system are as small as possible. OSEKtime also comprises a fault-tolerant communication layer that supports real-time communication protocols and systems and is described in FTCom [58] specification.

The following systems are examples of operating systems OSEK/VDX OS compliant. Most of them are from small companies.

Erika Enterprise [59] is a minimal real-time kernel for single and multicore embedded systems. It is a free, open-source implementation of the OSEK/VDX API, implementing conformance classes BCC1, BCC2, ECC1, ECC2, with an OSEK OIL compiler integrated into Eclipse.

PICOS18 [60], is an operating system based on OSEK/VDX standard. It is designed by Pragmatec Inc. for the PICmicro microcontrollers from the Microchip PIC18 family and is totally free and distributed under the GPL license.

Trampoline [61] is an open source RTOS which, once certified, could be compliant with the OSEK/VDX specification. Currently it is not the case, so while Trampoline has the same API as OSEK/VDX, it is not officially compliant. Trampoline is available under the GNU Lesser General Public License V2.

OSEKturbo [62] is a small, scalable Real-Time Operating System (RTOS) that provides a set of RTOS services that can be leveraged by your embedded application. Developed in accordance with the Software Engineering Institute's (SEI) highest Capability

Maturity Model (CMM) rating and fully compliant to the latest OSEK/VDX specifications, the operating system is designed to occupy very little memory, provide fast context switching times and increase reusability of your embedded application.

RTA-OSEK [63] provides a production real-time operating system suitable for applications in all areas of automotive ECU design. It implements the AUTOSAR-OS V1.0 (SC-1) and OSEK/VDX OS V2.2.3 standard and is fully MISRA compliant. An extremely small and fast runtime kernel is supplied for more than 20 popular microcontrollers, together with the Planner and Builder tools that are used to configure and analyze the operating system.

The Systems listed here are all OSEK/VDX OS compliant and so they tend to be very concern with the features listed above. As the information on these systems are very poor we are assuming they provide and enforce the features we have listed above.

Embedded Sensor Networks

Recently, the availability of cheap and small tiny sensors and low power wireless communication allowed the large-scaled deployment of sensor nodes in Embedded Sensor Networks (ESN). An embedded sensor network is a network of embedded computers placed in the physical world that interacts with the environment. These embedded computers, or sensor nodes, are often physically small, relatively inexpensive computers, each with some set of sensors or actuators [64]. Their nodes communicate wirelessly and each node consists of: processing capability (one or more microcontrollers, CPUs, or DSP chips), may contain multiple types of memory (program, data, and flash memories), have an RF transceiver, have a power source (batteries and solar cells), and accommodate various sensors and actuators. Sensor nodes have evolved into two broad categories. The first one consist of small devices with 8-bit microcontrollers CPUs, 10/100KB of working memory, and 100/1000KB of flash secondary storage, such as motes [65]. The second one consist of larger devices with 32-bit CPUs and megabytes each of working memory and secondary storage, such as Cerfcube[66]. The nodes often self-organize after being deployed in an *ad hoc* fashion. Systems of 1,000s or even 10,000 nodes are anticipated. Such systems can revolutionize the way we live and work, it is not irrational to expect that in a decade the world will be covered with wireless sensor networks with access to them via the Internet [5].
To be usable a sensor networking system must provide several services, further than the lower-level networking primitives. When sensor network programmers make a program for sensor network applications, without any middleware or operating system, the development of application is very difficult. Many services, such as operating systems, are also found in traditional wired and wireless networks. The sensor networking community typically uses embedded (and, possibly, real-time) versions of existing operating systems such as Linux for the larger devices. These embedded versions provide largely the same programming support as their regular counterparts, but with additional device-level support for embedded controllers, flash memory, and other peripherals specific to these devices. As such, not much research has been required on new operating systems support for these larger devices. On the other hand, the smaller devices (such as the motes) have required novel directions in operating system design.

In order to attend these novel directions the following requirements generally shape the design of network sensor systems [67], affecting directly the design of operating systems support for network sensor systems:

Small physical size and low power consumption: At any point in technological evolution, size and power constrain the processing, storage, and interconnect capability of the basic device. The current trend of low-end embedded processors is toward larger ROM sizes (64KB to 128 KB) and smaller RAM sizes (2KB to 8KB). The OS architecture should be compliant with this trend by optimizing for RAM with a higher priority than ROM and optimizing for runtime efficiency. If limits on the usage of energy can be enforced, lifetime guarantee requirements of the system as a whole can likely be provided (under reasonable assumptions about operating conditions such as network connectivity). The OS can also ensure that the system energy is apportioned in a manner corresponding to the importance of the tasks so that critical tasks are guaranteed their energy budget. True preemptive multitasking becomes necessary in a system where multiple inputs to the system must be serviced at different rates within a required period.
Concurrency-intensive operation: What is crucial in mode of operation for these devices is to flow information from place to place having a modest amount of processing on-the-fly, in order to accept a command, stop, analize, and respond. For example, it is possible to simultaneously capture information from sensors, manipulate them, and put then onto a network. The OS should provide a simple and intuitive programming paradigm for easy use by application developers. It is desirable to retain the traditional multitasking paradigm familiar to both desktop and embedded system programmers. Application developers should be able to concentrate on application logic rather than low-level system issues such as scheduling, and networking.
Diversity in Design and Usage: Typically networked sensor devices are application specific, rather than general purpose. They carry only the available hardware support actually needed for the application. There is a wide range of potential applications, so the variation in physical devices is likely to be large. In order to provide functionality on any particular device, it is important to easily assemble just the software components required to synthesize the application from the hardware components. As a result, we need an unusual degree of software modularity that must also be very efficient to provide what these devices require. Providing a unified and simple abstraction for accessing sensor readings and actuating responses would greatly benefit the end-user. In particular, low-level details associated with sensor/actuator configurations should be abstracted away from the user. Sensors should be supported using device drivers that can return real-world units as well as raw ADC values. Moreover, it should be natural to migrate components across the hardware/software boundary as technology evolves.
Robust Operation: These devices will be numerous, largely unattended, and expected to form an application which will be operational a large percentage of the time. The application of traditional redundancy techniques to enhance the reliability of individual units is limited by space and power. Since sensor nodes are resource-constrained, precious CPU cycles, network buffers and bandwidth should be apportioned to application needs. OS support for guaranteed, timely and limited access to system resources is necessary for supporting application deadlines and balanced apportioning of system slack (residual unused resources). This mechanism can also be used to place some limits on the impact of faulty or malicious tasks on system operation.
Timeliness and Schedulability: Most sensor applications such as surveillance tend to be time sensitive in nature where packets must be relayed and forwarded on a timely basis.

While outing and network link scheduling are important components in ensuring that packets meet their end-to-end delay bounds, timing support on each node in the network is also essential. In order to honor end-to-end deadlines, local tasks on each node have deadlines associated with the completion of their local data relaying and processing. Managing the deadlines of these tasks requires support of a *real-time* operating system.

One such direction has been the development of a number of OSes for embedded sensor networks and networked low-power systems. TinyOS [67], an operating system for the motes and widely used by many research groups as well as in some segments of industry, deviates significantly from the traditional multi-threaded model of modern operating systems. MantisOS [68] and Contiki [69] are two recent projects providing multithread support. Other OSes like Nano-RK [70], and Pixie [71] , provide different approaches in order to support embedded sensor networks.

TinyOS is a tiny, flexible operating system built from a set of reusable components that are assembled into an application-specific system. It supports an event-driven concurrency model based on split-phase interfaces, asynchronous *events*, and deferred computation called *tasks*. TinyOS has a component-based programming model, codified by the NesC language, a dialect of C, which supports the TinyOS component and concurrency model as well as extensive cross-component optimizations and compile-time race detection. TinyOS is not an OS in the traditional sense; it is a programming framework for embedded systems and set of components that enable building an application-specific OS into each application. A typical application is about 15K in size, of which the base OS is about 400 bytes; the largest application, a database-like query system, is about 64K bytes.
A TinyOS program is a graph of components, where each of the components is an independent computational entity that exposes one or more *interfaces*. The components provide three abstractions: *commands*, which is typically a request to a component to perform some service (such as initiating a sensor reading); *events*, which are typically used to signal the completion of that service, and *tasks*, which is a function executed by the TinyOS scheduler at a later time. While commands and events are mechanisms for inter-component communication, tasks are used to express intra-component concurrency.
TinyOS uses a two level scheduling hierarchy that lets high-priority events pre-empt low priority tasks. It supports a cyclic-executive model wherein interrupts can register events, which can then be acted upon by other non-blocking functions. Events are invoked because of external input such as incoming data or sensor input. Events can post tasks for later processing. Both events and tasks must run to completion after being invoked. This precludes the use of blocking statements. Events are implemented using hardware interrupts, and tasks are implemented using a linear FIFO dispatcher. The dispatcher has a queue of tasks, where each task is represented by a pointer to a function.
The current version of TinyOS provides a large number of components to application developers, including abstractions for sensors, single-hop networking, ad-hoc routing, power management, timers, and non-volatile storage. A developer composes an application by writing components and wiring them to TinyOS components that provide implementations of the required services.
TinyOS is open-source software, published under a 3-clause BSD license. It has been under development for several years and is currently in its third generation involving

several iterations of hardware, radio stacks, and programming tools. Over one hundred groups worldwide use it, including several companies within their products.

Contiki is an operating system designed for networked and memory-constrained systems. Contiki, like TinyOS, is based around an event-driven kernel but has additional support for dynamically loadable programs. Unlike TinyOS, Contiki includes the uIP[8] stack for TCP/IP communication. It also allows applications to be written in a multi-threaded fashion. Multi-threading is implemented as a library that is optionally linked with those applications that specifically requires a threaded model of execution. The event-driven nature of the kernel makes the system compact and responsive, whereas the multi-threading makes it possible to run programs that perform long-running computations without completely blocking the system. Additionally, Contiki provides a third execution model called protothreads. A protothread is an extremely lightweight stack-less thread-like construct that provides linear execution on top of the event-driven kernel.
A Contiki system is composed by the kernel, libraries, the program loader, and a set of processes. It is partitioned into two parts: the *core* and the *loaded programs* as shown in Figure 4.14, and partitioning is made at compile time and 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.

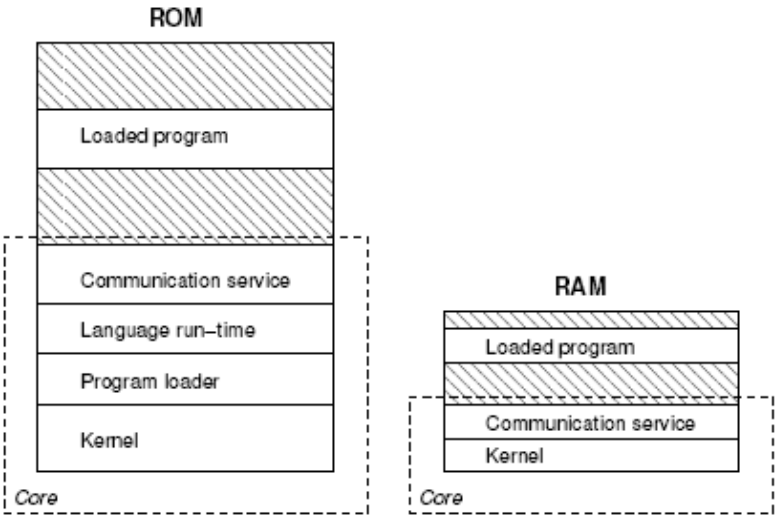

Figure 4.14 Contiki OS Core

A process may be either an application program or a *service* which is the implementation of functionality used by more than one application process. All processes, both application programs and services, can be dynamically replaced at run-time. In addition, processes share the same address space and do not run in different protection domains. The definition of a process contains an event handler function and an optional poll handler function. Communication between processes always goes through the kernel, and it is done by posting events. The kernel does not provide a hardware abstraction layer, but lets device drivers and applications communicate

directly with the hardware. A process is defined by an event handler function and an optional poll handler function.

The *MANTIS* open source RTOS (*MOS*) as being adapted to the additional requirements imposed by sensor networks, e.g. the development of a power-efficient scheduler to reduce energy consumption and the implementation of advanced sensor specific features like remote dynamic reprogramming of micro sensor nodes. MANTIS OS provides lightweight memory footprint as well as energy-efficient operation. At present, the MOS kernel is able to achieve multithreaded preemptively scheduled execution with standard I/O synchronization and a network protocol stack, all for less than 500 bytes of RAM, not including individual thread stack sizes.
MOS is also designed to provide advanced remote management capabilities for in-situ sensor networks. Towards this end, the goals of MOS are to support useful yet sophisticated features, including dynamic reprogramming of sensor nodes via wireless, remote debugging of sensor nodes, and multimodal prototyping of virtual and deployed sensor nodes. The goal of the MOS kernel design is to implement familiar services such as POSIX threads, binary (mutex) and counting semaphores in a manner efficient enough for the resource-constrained environment of a sensor node. The design of the MOS kernel resembles classical, UNIX-style schedulers, most notably priority-based thread scheduling with round-robin semantics within a priority level. The scheduler receives a timer interrupt from the hardware to trigger context switches; switches may also be triggered by system calls or semaphore operations. The timer interrupt is the only one handled by the kernel–other hardware interrupts are sent directly to the associated device drivers. Upon an interrupt, a device driver typically posts a semaphore in order to activate a waiting thread, and this thread handles whatever event caused the interrupt. The time slice is configurable, and is currently set to about 10 ms. They claim that automatic preemption, provided by time-sliced multithreading, is important in sensor systems, since blocking certain time-critical tasks from executing, such as network packet processing, can result in overflow of network buffers when tasks are sufficiently long-lived and a sensor node's RAM buffers are sufficiently small.
The MANTIS multithreaded OS provide a structure, figure 4.15, where the main components of MOS are the MOS Kernel, COM Layer, DEV Layer, and NET Layer.

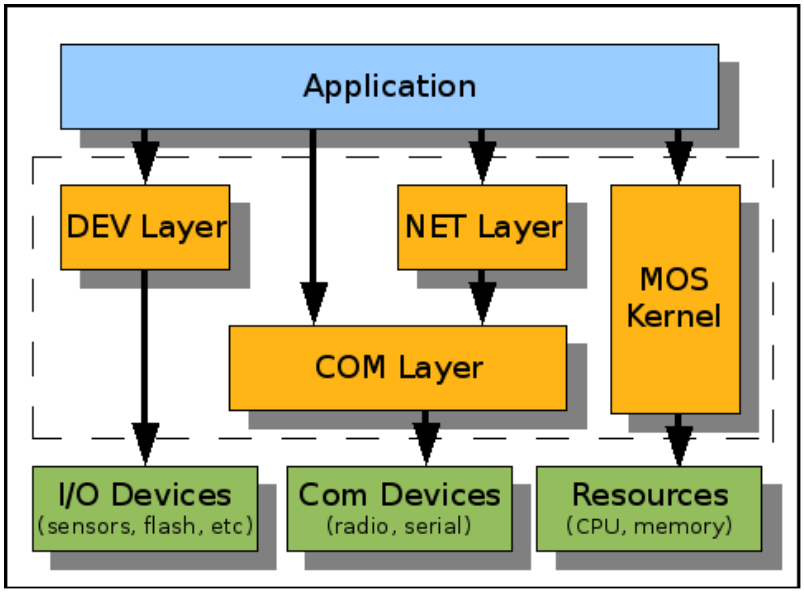

Figure 4.15 MOS architecture (http://mantis.cs.colorado.edu/tikiwiki/tiki-index.php)

The MOS Kernel, which is responsible for system management, provides a preemptive multi-threaded environment including power modes and timing. The DEV layer provides a consistent interface for accessing peripheral devices such as sensors, flash, and so on. The COM Layer provides a consistent interface to communications devices, such as the the radio and serial lines. The NET Layer allows network protocols to be abstracted away from program logic; multiple routing protocols can be registered and implemented using the NET interface.

Nano-RK is a fully preemptive reservation-based real-time operating system (RTOS) from Carnegie Mellon University with multi-hop networking support for use in wireless sensor networks. It includes a light-weight embedded resource kernel (RK) with rich functionality and timing support using less than 2KB of RAM and 18KB of ROM. Nano-RK supports fixed-priority preemptive multitasking for ensuring that task deadlines are met, along with support for CPU, network, as well as, sensor and actuator reservations. It provides explicit support for periodic task scheduling with support for real-time task sets that have deadlines associated with their data delivery.

It uses the novel mechanisms of CPU and network reservations to enforce limits on the resource usage of individual tasks. With respect to networking it is provided a rich API set for socket-like abstractions, and a generic system support for network scheduling and routing. NanoRK is power-management and provides several power-aware APIs that can be used by the system. Tasks can specify their resource demands and the operating system provides timely, guaranteed and controlled access to CPU cycles and network packets. Together these resources form virtual energy reservations that allows the OS to enforce system and task level energy budgets.

Nano-RK uses a static design-time framework, consistent with sensor networking assumptions, where the OS and the applications are co-located in a single address space. In order to guarantee timeliness and enforce temporal isolation Nano-RK uses the reservation paradigm, allowing applications to specify timeliness and resource requirements, and the OS enforces guaranteed access to system resources and schedules tasks so that the application timeliness requirements are satisfied.

The system uses priority-based preemptive scheduling and while providing explicit support for periodic tasks, it also supports aperiodic and sporadic tasks in its framework. The highest priority task that is eligible to run in the system is always scheduled by the operating system. A periodic task can suspend itself after the completion of its current instance using a system call. Real-time synchronization is supported in Nano-RK, through priority ceiling protocol emulation (Highest Locker Priority protocol).

Figure 4.16 shows the architecture of Nano-RK. It provides an API which includes task management functions, signals and semaphores functions, general device drivers management functions and resource reservation functions.

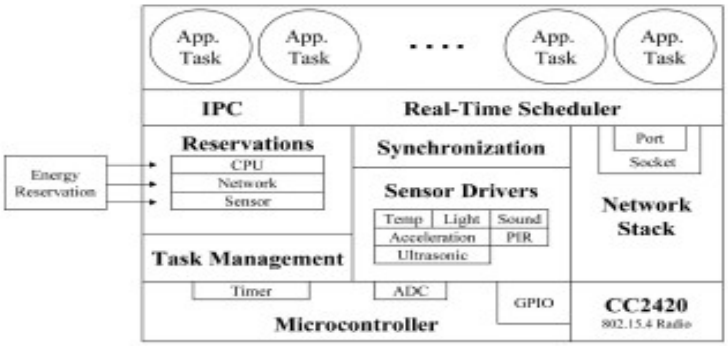

Figure 4.16 Nano-RK Architecture [83]

Pixie is an operating system for sensor networks that enables *resource-aware programming*, a model in which applications receive feedback on, and have explicit control over, resource usage. Pixie is designed to support the needs of data-intensive applications. These applications, which include high-resolution monitoring of acoustic, seismic, acceleration, and other signals, involve high data rates and extensive in-network processing. Given the fundamentally resource-limited nature of sensor networks, a pressing concern for such applications is their ability to receive feedback on, and adapt their behavior to, fluctuations in both resource availability and load. The Pixie architecture consists of three components, as showed in Figure 4.17. The Application Dataflow layer implements application logic, the Resource Manager allocates and manages resources for the application, and the Scheduler schedules and executes application stages.

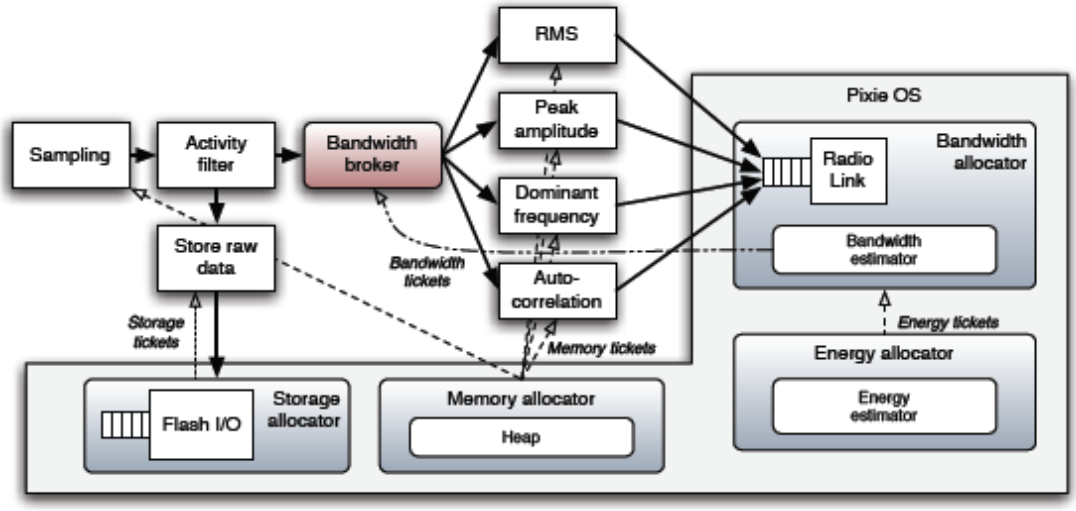

Figure 4.17 An example Pixie application for limb monitoring using wearable sensors. Resource allocators and brokers are components of the Pixie OS [71].

The Pixie OS is based on a dataflow programming model and is based on the concept of *resource tickets*, a core abstraction for representing resource availability and reservations which means a right to consume a given amount of a resource type within a time window. To shield application programmers from the burden of managing these

details, Pixie provides a suite of *resource brokers*, which mediate between low-level physical resources and higher-level application demands. For example, Pixie's energy broker implements a policy for the system to achieve a target lifetime. In this case, the energy broker will trickle the amount of energy the application can use at a specific rate. Pixie is implemented in NesC and supports limited backwards compatibility with TinyOS.

Table 4.3 summarizes some operating systems for sensor networks domain embedded systems and what properties (requirements) they provide. The table refers to the same set of requirements we have introduce earlier at chapter 3. In this table we want to give you a view of how the systems provide and enforce a certain property: *adequately* (A) in case the system provides and enforces the property, *poorly* (P) when the system provides the property but not as required (for example, when the system is said to be real-time but do not support hard real-time behavior), and *non* (N) when the system does not provide the requirement.

OS/Req.	**(*i*)**	**(*ii*)**	**(*iii*)**	**(*iv*)**	**(*a*)**	**(*b*)**	**(*c*)**	**(*d*)**	**(*e*)**	**(*f*)**	**(*g*)**
TyniOS	A	P	A	P	P	N	N	N	A	A	N
Contiki	A	P	A	P	P	N	N	N	A	A	P
MantisOS	A	P	A	P	P	N	N	N	P	A	A
Nano-RK	A	A	A	P	P	N	N	N	P	P	P
Pixie	A	P	A	P	P	N	N	N	P	P	A

Table 4.3 Embedded Systems Requirements provided by OSes for Sensor Networks Domain

One of the characteristics of OSes for sensor networks is to be adequately able to deal with resource constraints such as memory and low power. They all provide and ensure this particular requirement. However, most of the systems does not provide an adequately real-time behavior. Regarding some of the important requirements for distributed embedded systems these OSes perform very poorly. Therefore, it seems that those requirements have to be supported in a different level than operating system.

Conclusion

We have presented various operating systems that are supposed to be tailored for different types of embedded systems, from stand alone embedded systems (SES) such as a digital camera to distributed real-time embedded systems (DRE) such as a military defense system. It is possible to say that for SES systems the requirements that are important for these systems are basically supported by the existing OSes. However, regarding most of networked embedded systems requirements there is no OS that provide and enforce all the requirements.

For instance, the many types of DRE systems all have one thing in common: to deliver the right answer at the right time. Providing the right answer at the right time is crucial for, life-critical military DRE systems, such as those defending ships against missile attacks or controlling unmanned combat air vehicles through wireless links as well as for safety-critical civilian DRE systems, such as those regulating the temperature of coolant in nuclear reactors and maintaining the safe operation of steel manufacturing machinery. It is hard to design DRE systems that implement their required quality of

service (QoS) capabilities, are dependable and predictable, and are cost-conscious in their use of computing resources, being supported by only operating system software infrastructure. It is even harder to build them on time and within budget.

As a result, distributed real-time and embedded systems are built using a common layer of software infrastructure, called middleware, which serves two purposes. The first goal is to ease the development of applications by abstracting away the particular details of the hardware and operating system that executes in each computational site. The second purpose is to provide a family of services that are common to many applications, simplifying component design and increasing reusability while allowing specific optimizations for a particular deployment.

Network Embedded Systems, especially DRE systems can span a variety of network types, topologies and scales (e.g., ranging from next-generation local sensor/actuator networks, to large scale traffic or power grid management systems). A single real-time and embedded system could also span multiple networks with significantly different characteristics. A general theme of these systems is that independent of the characteristics of the networks, the constraints of stringent application-specific must be enforced by all infrastructure software service including OS, middleware and the application as much as enforced end-to-end by the network. Also, applications with different constraints will share the networking and other physical and logical resources. How to provide different service classes and ensure the proper allocation and protection of shared resources across all the layers, including operating systems, consistently is another important challenge. Still, creating the proper interfaces between the network infrastructure and the applications is an open research issues for operating systems.

References

[1] Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers. Committee on Networked Systems of Embedded Computers, National Research Council, 2001

[2] DARPA IXO, "Networked Embedded Software Technology (NEST)." http://www.darpa.mil/ixo/.

[3] Report of NSF Workshop on Distributed Real-time and Embedded Systems Research in the Context of GENI (October 2005 and February 2006) – (http://www.geni.net/GDD/GDD-06-32.pdf).

[4] FP6 IP "RUNES" - D5.1 Survey of Middleware for Networked Embedded Systems, January 2005 – (http://www.ist-runes.org/docs/deliverables/D5_01.pdf).

[5] Handbook of real-time and embedded systems / Insup Lee, Joseph Y-T. Leung and Sang H. Son. Chapman & Hall/CRC computer & information science series, 2008.

[6] Alfons Crespo, Ismael Ripoll, Michael González-Harbour, and Giuseppe Lipari, Operating System Support for Embedded Real-Time Applications, EURASIP Journal on Embedded Systems, Volume 2008 , 2 pages.

[7] Distributed Systems: Concepts and Design, second edition, George Coulouris, Jean Dollimore, and Tim Kindberg, Addison-Wesley 1994.

[8] A. Dunkels. The uIP TCP/IP Stack for Embedded Microcontroller. http://www.sics.se/~adam/uip/

[9] A. Dunkels. lwIP, a Lightweight TCP/IP Stack. http://www.sics.se/~adam/lwip/, http://savannah.nongnu.org/projects/lwip/

[10] Viola Systems Ltd. OpenTCP. http://www.violasystems.com/opentcp.php

[11] G. Denys, F. Piessens, and F. Matthijs, A Survey of Customizability in Operating Systems Research, ACM Computing Surveys, Vol. 34, No. 4, December 2002, pp. 450-468.

[12] Frank Engel, Gernot Heiser, Ihor Kuz, Stefan M. Petters and Sergio Ruocco Operating systems on SoCs: a good idea? , Embedded Real-Time Systems Implementation (ERTSI 2004) Workshop, Lisbon, Portugal, December, 2004

[13] VxWorks, http://www.windriver.com.

[14] Microsoft Corporation. Microsoft Windows CE. http://www.microsoft.com/windowsce/embedded/

[15] Dan Hildebrand. An Architectural Overview of QNX. Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures, 113–126, 1992.

[16] QNX Software Systems Ltd. QNX Neutrino Realtime OS. http://www.qnx.com/products/os/neutrino.html.

[17] Palm, Inc. PalmOS Software 3.5 Overview. http://www.palm.com/devzone/docs/palmos35.html.

[18] Microware. Microware OS-9. http://www.microware.com/ProductsServices/Technologies/os-91.html.

[19] LynuxWorks. LynxOS 4.0 Real-Time Operating System. http://www.lynuxworks.com/.

[20] Symbian. Symbian OS - the mobile operating system. http://www.symbian.com/.

[21] Jim Turley, Embedded systems survey: Operating systems up for grabs, Embedded Systems Design, 2005. http://www.embedded.com/columns/surveys/163700590?_requestid=50912

[22] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. J.W. Haskins. A survey of configurable component-based operating systems for embedded applications. IEEE Micro, May 2001.

[23] eCos - Embedded Operating System. http://ecos.sourceware.org/

[24] A. Massa. Embedded Software Development with eCos. Prentice Hall, 2003.

[25] Integrated Chipware, Inc. Integrated Chipware icWORKSHOP. http://www.chipware.com/.

[26] J. A. Stankovic, H. Wang, M. Humphrey, R. Zhu, R. Poornalingam, and C. Lu. VEST: Virginia Embedded Systems Toolkit. In IEEE/IEE Real-Time Embedded Systems Workshop, London, December 2001.

[27] B. Kauler. CREEM Concurrent Realitme Embedded Executive for Microcontrollers. http://www.goofee.com/creem.htm.

[28] Wind River Systems, Inc. pSOSystem Datasheet. http://www.windriver.com/products/html/psosystem_ds.html.

[29] Wind River Systems, Inc. OSEKWorks 4.0. http://www.windriver.com/products/osekworks/osekworks.pdf.

[30] Microware. Microware Ariel RTOS. http://findarticles.com/p/articles/mi_m0EIN/is_/ai_20862996.

[31] L. Dozio, P. Mantegazza, Real Time Distributed Control Systems Using RTAI, Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Hakodate, Hokkaido, Japan, 14-16 May 2003.

[32] V. Yodaiken, M. Barabanov, A Real-Time Linux.

http://www.soe.ucsc.edu/~sbrandt/courses/Winter00/290S/rtlinux.pdf

[33] Embedded Linux - http://www.linuxdevices.com/.

[34] uClinux – Embedded Linux Microcontroller Project – http://www.uclinux.org/.

[35] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos, Can we Make Operating Systems Reliable and Secure?, IEEE Computer, May 2006, pp. 44-51.

[36] Gernot Heiser, Kevin Elphinstone, Ihor Kuz, Gerwin Klein, and Stefan M. Petters. Towards trustworthy computing systems: taking microkernels to the next level , ACM SIGOPS Operating System Review, 41(4), July 2007, pp. 3-11.

[37] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Construction of a Highly Dependable Operating System, Proceedings of the Sixth European Dependable Computing Conference (EDCC'06), 2006.

[38] Gernot Heiser, Secure Embedded Systems Need Microkernels. The USENIX Magazine, 30(6), December 2005, pp. 9-13.

[39] ARINC. *ARINC 653:* Avionics Application Software Standard Interface (Draft 15). Airlines Electronic Engineering Committee (AEEC), June 17th, 1996.

[40] R. A. Edwards. ASAAC phase I harmonized concept summary. In Proceedings ERA Avionics Conference and Exhibition, London, UK, 1994.

[41] D. Field and J. Kemp. The ASAAC Programme, NATO HQ, Brussels, July 20th, 2000. Available from: http://www.safsec.com/safsec files/resources/nato p 1.ppt.

[42] Airlines electronic engineering committee (AEEC), avionics application software standard interface - ARINC specification 653 - part 1 (supplement 2 - required services). ARINC, Inc., 2006.

[43] Airlines electronic engineering committee (AEEC), avionics application software standard interface - ARINC specification 653 - part 2 (extended services). ARINC, Inc., June 2007.

[44] Jim Alves-Foss, W. Scott Harrison, Paul Oman, and Carol Taylor, The MILS Architecture for High-Assurance Embedded Systems. International Journal of Embedded Systems, 2006, 2, pp. 239-247.

[45] Rance J. DeLong, LynxSecure Separation Kernel – a High-Assurance Security RTOS, LynuxWorks, San Jose, CA, 2007. http://www.lynuxworks.com.

[46] Robert Kaiser and Stephan Wagner, The PikeOS Concept History and Design, SYSGO, 2007. http://www.**sysgo**.com.

[47] RTEMS - Real-Time Operating System for Multiprocessor Systems, http://www.rtems.com/.

[48] Sergio Ruocco, A Real-Time Programmer's Tour of General-Purpose L4 Microkernels. EURASIP Journal on Embedded Systems, Volume 2008 , 14 pages.

[49] DIANA Project - http://diana.skysoft.pt/

[50] J. Rufino, S. Filipe, M. Coutinho, S. Santos, and J. Windsor. ARINC 653 interface in RTEMS. In Proceedings of the DASIA 2007 .DAta Systems In Aerospace. Conference, Naples, Italy, June 2007. EUROSPACE.

[51] Dawn Mission - http://dawn.jpl.nasa.gov/

[52] THEMIS Mission - http://www.nasa.gov/mission_pages/themis/main/index.html

[53] OSEK/VDX. Open Systems and the Corresponding Interfaces for Automotive Electronics. http://www.osekvdx.org/.

[53] OSEK/VDX-Operating System. Version 2.2.2, July 2004. http://www.osek-vdx.org/mirror/os222.pdf.

[54] OSEK/VDX-Communication. Version 3.0.3, July 2004. http://www.osek-vdx.org/mirror/OSEKCOM303.pdf.

[55] OSEK/VDX-Network Management. Version 2.5.3, July 2004. http://www.osek-vdx.org/mirror/nm253.pdf.

[56] AUTOSAR. Specification of Operating System — Version 2.0.1. Technical report, AUTOSAR GbR, 2006.

[57] OSEKtime OS. http://portal.osekvdx.org/files/pdf/specs/ttos10.pdf.

[58] OSEKFT Com. http://portal.osek-vdx.org/files/pdf/specs/ftcom10.pdf.

[59] Erika Enterprise. http://www.evidence.eu.com/content/view/27/254/ .

[60] PICOS18. http://www.picos18.com/index_us.htm .

[61] Trampoline. http://trampoline.rts-software.org/ .

[62] OSEKturbo. http://www.metrowerks.com .

[63] RTA-OSEK. http://www.etas.com/en/products/rta_osek.php .

[64] John Heidemann and Ramesh Govindan, An Overview of Embedded Sensor Networks, in Handbook of Networked and Embedded Control Systems, D. Hristu-Varsakelis and W.S. Levine, editors, Springer-Verlag, 2004.

[65] J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. IEEE Micro, 22(6):12{24, Nov/Dec 2002.

[66] Intrinsyc. http://www.intrinsyc.com/.

[67] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System architecture directions for networked sensors. In Architectural Support for Programming Languages and Operating Systems, pages 93–104, Boston, MA, USA, Nov. 2000.

[68] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, R. Han, MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms, ACM/Kluwer Mobile Networks & Applications (MONET), Special Issue on Wireless Sensor Networks, vol. 10, no. 4, August 2005, guest co-editors P. Ramanathan, R. Govindan and K. Sivalingam, pp. 563-579.

[69] A. Dunkels, B. Grönvall and T. Voigt. Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors. In Proceedings of the First IEEE Workshop on Embedded Networked Sensos, Tampa, Florida, USA, November 2004

[70] A. Eswaran, A. Rowe and R. Rajkumar, Nano-RK: An Energy-Aware Resource-Centric Operating System for Sensor Networks, IEEE Real-Time Systems Symposium, December 2005.

[71] Konrad Lorincz, Bor-rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh, Pixie: An Operating System for Resource-Aware Programming of Embedded Sensors, Fifth Workshop on Embedded Networked Sensors (HotEmNets'08), June, 2008

[72] J. Liedke, On µ-Kernel Construction, In Proceedings of 15th ACM Symposium on Operating Systems Principles, December 1995, pp. 237-250.

[74] H. Härtig, M. Hohmuth, J. Liedke, S. Schonberg, and J. Wolter. The Performance of On µ-Kernel based Systems, Proceedings of 16th ACM Symposium on Operating Systems Principles, October 1997, pp. 66-77.