Programming real-time systems with C/C++ and POSIX

Michael González Harbour

1. Introduction

The C language [1], developed in 1972 by Dennis Ritchie at the Bell Telephone Laboratories, is the most widely used high level programming language in the embedded systems community, both for systems programming as for the development of applications. Although as a general purpose language it is being surpassed by Java [7], C is still favourite among embedded system developers because of the efficiency of the generated code, the simplicity and wide availability of the compilers, and the availability of numerous development tools, even though C does little to promote robustness and reliability of the developed code.

The C++ language, as a superset of C with object-oriented facilities, is a natural candidate for those developers moving from applications developed in C into a world of more complex software in which object-oriented abstractions help in handling the complexity and increasing reuse. Although more programmers continue to use C, the popularity of C++ for developing embedded systems is increasing [11].

Both C and C++ lack support for developing real-time systems, mainly because they don't offer services for programming concurrent software. This limitation implies that for developing real-time systems it is necessary to rely on services offered by some additional software package that provides a means to build concurrent programs with predictable timing behaviour. These services can rely on in-house developed kernels or third-party real-time operating systems (RTOS).

If an application developer wants to preserve the investment in developing software, it is important that the services offered by the RTOS follow a standard API that allows the application to be ported to different platforms, even from different vendors. One of these APIs that has support for real-time systems, is supported by major RTOS vendors and is approved as an international standard is POSIX [1].

POSIX stands for "Portable Operating Systems Interface" and is the standardisation or the widespread UNIX operating system. The title of the standard tells us two important things about it. The main objective of the standard is portability of applications. The way this is achieved is by defining the interfaces or APIs between the operating system (OS) and the application. The standard describes the services that the OS must provide, and the syntax and semantics of their interfaces, in terms of data types and function prototypes. The actual implementation of those services is not specified by the standard, and is left to the open competition among OS vendors. Since the standard defines the interfaces at the source code level, the portability achieved is also for the source code. Binary level portability is outside the scope of the standard because it requires the use of a unified processor (or virtual processor) architecture.

UNIX systems were typically non real-time and large, quite far away from the traditional RTOS that was usually small, adapted to embedded systems with limited resources and with real-time requirements. POSIX itself, as the standardisation of UNIX is not primarily intended for real-time systems. However, the OS community made a large effort to bring real-time into the POSIX standard by adding those services that are required to ensure that the timing behaviour of the system can be made predictable. It also made a big effort to develop standard subsets of the OS interfaces that would be suitable for building a small RTOS for embedded systems. These subsets are called profiles, and the POSIX.13 standard [3] defines four such profiles developed for different sizes of real-time embedded systems. For instance, the smallest of those profiles, called the "Minimal real-time systems profile", removes processes and the file system from the OS services and describes a small set of OS services that can be implemented in small platforms.

Even for the smallest profiles the OS services are quite powerful. They resemble those of a larger OS, and are upward compatible. The target of the standard is not the very simple 8 bit platforms requiring a memory footprint of a few hundred bytes, for which there are other standards that may be more appropriate (like ITRON or OSEK/VDX), but platforms that can hold a kernel with a footprint above 10 kilobytes.

The POSIX standard was developed by the IEEE and the first version was released in 1988 and contained interfaces in the C language for the most common system services found in UNIX. Other extensions were approved later, for instance the real-time extensions (POSIX.1b:1993) and the threads extensions (POSIX.1c:1995). Interfaces in other programming languages were developed as bindings (Fortran, Ada), and profiles or subsets were developed (POSIX.13:1998).

In addition to IEEE, The Open Group, an international consortium of OS vendors and users, was developing additional operating system services in rapid growing areas were the international standard process was slower. This set of standards received the name of "The Single Unix specification". As the interfaces matured, there was an effort to join the IEEE/ISO/IEC POSIX standards and the single UNIX specification. This resulted in incorporating the interfaces into a single unified standard that is now published jointly by IEEE [1] and The Open Group [2]. The last published version is from 2003. The POSIX real-time profiles were also updated to reflect this unification [3] but were kept as a separate standard. Figure 1 shows the history of some of the POSIX standards.

As a consequence of all these developments in the C, C++, and POSIX standards we can say that it is possible to develop portable real-time embedded applications using the C language and an RTOS that follows one of the POSIX real-time systems profiles. In the following sections we will try to describe the main services offered by the union of the POSIX OS and the language, with emphasis on those services required to provide predictable timing behaviour.

2. Description

In this section we describe the most relevant POSIX services from the point of view of developing real-time applications. Two fundamental requirements must be met for this purpose: the ability to create concurrent programs with multiple cooperative tasks, and the ability to bound the response time of the OS services and to predict the timing behaviour of the application.

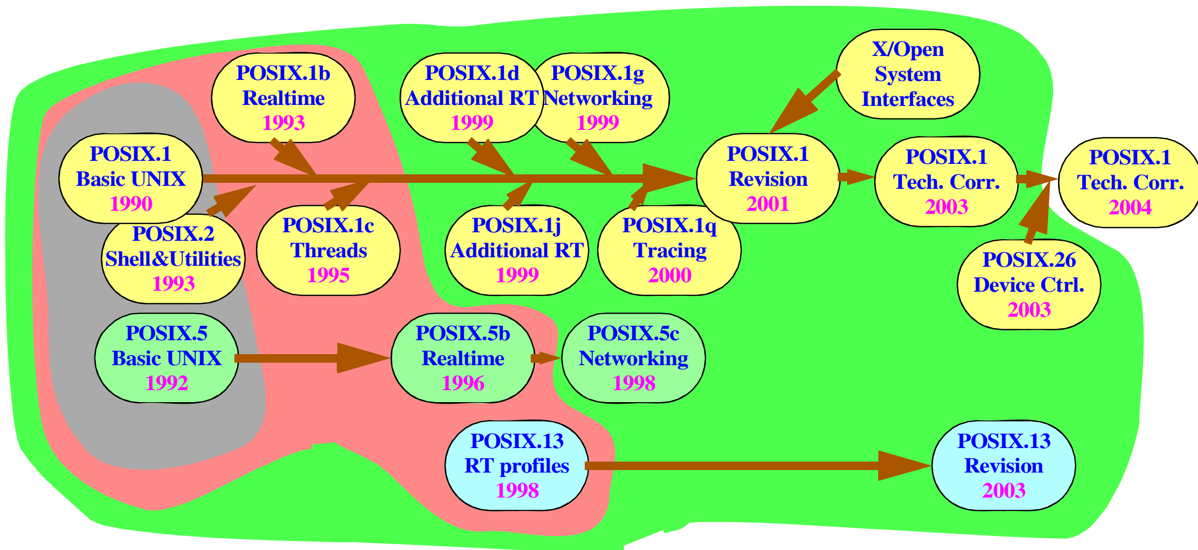

Figure 1. History of (some of) the POSIX standards

2.1 Concurrency

The POSIX standard offers two mechanisms to achieve the goal of programming concurrent tasks that can execute under logical parallelism with independent threads of control: processes and threads.

Processes are useful when developing independent application components because they define independent memory address spaces. It is possible to program applications composed of concurrent processes that can cooperate among themselves but whose memory and system resources are protected against accesses from other processes. This allows a spatial partitioning of the application and introduces a protection boundary against mistakes or intrusion. The smallest real-time profiles do not allow processes and therefore the only service providing concurrency is threads.

Threads allow programming concurrent activities that can share the same address space, and thus are more interesting for implementing cooperative tasks. Usually their management has less overhead than that of processes, as it doesn't involve manipulating the memory management unit to provide address space separation. When multiple processes are supported, each process has one or more threads that share the address space of their process.

2.2 Fixed-priority scheduling

POSIX defines three scheduling policies that can be used to schedule real-time applications. All three policies are based on preemptive fixed priorities. Each schedulable entity (process or thread) is assigned a priority and the system chooses for execution, from the entities that are ready to execute, the one with the highest priority. The policies are:

- SCHED_FIFO: FIFO order among entities of the same priority

- SCHED_RR: Round robin order among entities of the same priority
- SCHED_SS: Sporadic server scheduling, useful for scheduling aperiodic tasks

All these policies are compatible, so it is possible to assign them individually to each entity, in a mixed fashion.

When both processes and threads are present, three scheduling models may be supported by the implementation:

- System contention scope, in which the the only schedulable entity is the thread. All threads in the system compete among them, regardless of the process to which they belong.
- Process contention scope, in which the scheduler works at two levels. It first chooses a process according to its priority, and then chooses the highest priority thread of that process.
- Mixed contention scopes are also possible, in which some threads have a “system” scope, and other threads have a “process” scope.

For real-time applications the system scope leads to a higher degree of predictability.

2.3 Mutual exclusion synchronisation

Cooperative scheduling requires some synchronisation mechanism for accessing shared resources, most commonly based on mutual exclusion. Real-time systems require in addition that the mutual exclusion mechanisms avoid priority inversion that could destroy the predictability of the response times. POSIX provides a synchronisation object called the mutex, and describes two mechanisms for avoiding priority inversion: The immediate priority ceiling (called Priority Protection) useful for static systems in which it is possible to assign a priority ceiling, and priority inheritance, that generally produces more blocking but is useful in dynamic systems where it is difficult to assign such ceiling.

2.4 Signal/wait synchronisation

In addition to sharing resources, cooperative tasks also require some form of signalling and conditional waiting so that one task can wait for another one to complete some action. POSIX defines two mechanisms for this purpose counting semaphores, and condition variables.

A Condition variable is a powerful primitive that allows a task to wait until a condition is true. This condition is usually set true by some other thread. The condition itself is arbitrary state, possibly implying multiple variables or complex data structures, which can be checked and updated under the protection of a mutex. For this purpose, the condition variable is associated with a mutex that is atomically locked when a thread that was waiting on the condition is awakened. In summary, condition variables allow for signal & wait synchronisation based on complex conditions, and have a very efficient behaviour.

Counting semaphores are a classic low-level synchronisation primitive and can be used in POSIX to synchronise an application thread with an interrupt service routine or a signal handler.

2.5 Asynchronous notification

POSIX uses signals as a notification mechanism for events like certain exceptions, the expiration of a timer, or the end of an asynchronous I/O operation. Signals can also be generated explicitly from the application. In response to a signal, a process can execute a signal handler. It is also possible for a thread to explicitly wait for a signal from a certain set. In multithreaded processes signals may be delivered to any thread that expresses interest in it, so careful programming discipline is needed to achieve predictable results. The usual recommendation is to have only one thread within the process to express interest in a given signal.

2.6 Message passing

Message passing services are provided in POSIX, that allow variable-size messages to be exchanged among threads, through message queues. It is possible to poll for the availability of a message or to block waiting until one is available. This wait operation can be programmed with a timeout. Message passing can be used to exchange information among processes or threads, and to synchronise their work.

2.7 Timing services

Logical clocks are defined as objects that can measure the passage of time. They are given an identifier than can be used later in all the time-related services. Several clocks are defined by the standard:

- CLOCK_REALTIME: Measures “system” time, and may be subject to changes, for instance to adapt to the official time.
- CLOCK_MONOTONIC: Measures time in a monotonic fashion, at a constant rate. Usually real-time timing should be based on this clock, to make it immune to changes to the official calendar.
- Execution time clocks: they measure the CPU time consumed by a given process or thread

Based upon these clocks different services can be invoked. It is possible to sleep until a given clock reaches some absolute time, or until some relative interval elapses. It is also possible to create a timer, which is a software object that is capable of notifying the application when a given clock has reached a time, or an interval has elapsed. Timers may be programmed to expire periodically.

Execution time clocks are very useful in combination with timers to monitor the CPU usage from a given thread, and to be able to detect the consumption of excessive time. This is key in real-time systems, because the schedulability analysis is based on the assumption that worst-case execution times are never exceeded. This assumption can be enforced by the OS.

2.8 Memory management

The C language defines basic primitives for explicit allocation or deallocation of memory. Since in many implementations these services have unbounded response times, it is usually consid-

ered good practice in real-time systems to restrict these services to initialisation or to the non-real time parts of the application.

Also crucial to the real-time behaviour is the use of virtual memory. This is a widely used mechanism to make the logical memory used by an application independent of the physical memory, but it can introduce large delays in the memory access times that would ruin predictability of the response times. Therefore it is very important to lock into physical memory those parts of the application that have real-time requirements. POSIX provides services for this purpose.

2.9 Other services

In addition to the services mentioned above, which have important implications in the real-time properties of an application, there are other general-purpose services that are required from the OS, such as I/O, file system, networking, ... As a standard that targets both real-time and general purpose systems, POSIX defines all these services and many more.

2.10 Interoperability with other languages

The POSIX family of standards contains language bindings for interfacing the OS services from languages other than C. In particular, there exist bindings for Fortran and Ada, although the real-time extensions are defined only for the latter.

Ada already has concurrency integrated in the language, but it can still get benefit from POSIX from two perspectives. On the one hand, POSIX defines processes with separate address spaces, so it is possible to program applications composed of multiple Ada programs or partitions (each being implemented as a process) that cooperate among them by using the OS services, and get the benefit of the spatial protection provided by the processes.

On the other hand, the POSIX services provide a means to achieve interoperability between Ada tasks and threads written in C. Consistent scheduling, timing and synchronisation may be achieved through the OS services.

2.11 Profiles

As it an be seen in Figure 2, each of the four real-time profiles defined in POSIX.13 is a subset of the next profile, and of the overall POSIX standard:.

- Minimal Real-Time System (PSE51): The main services are threads, fixed-priority scheduling, mutexes with priority inheritance, condition variables, semaphores, timing services including CPU-time clocks, simple device I/O, signals, and memory management. See Figure 3.
- Real-Time Controller (PSE52): Modelled after industrial controllers, this profile adds a simple file system, message queues, and tracing facilities.
- Dedicated Real-Time system (PSE53): Intended for large embedded systems, this profile adds multiple processes, networking and asynchronous I/O.

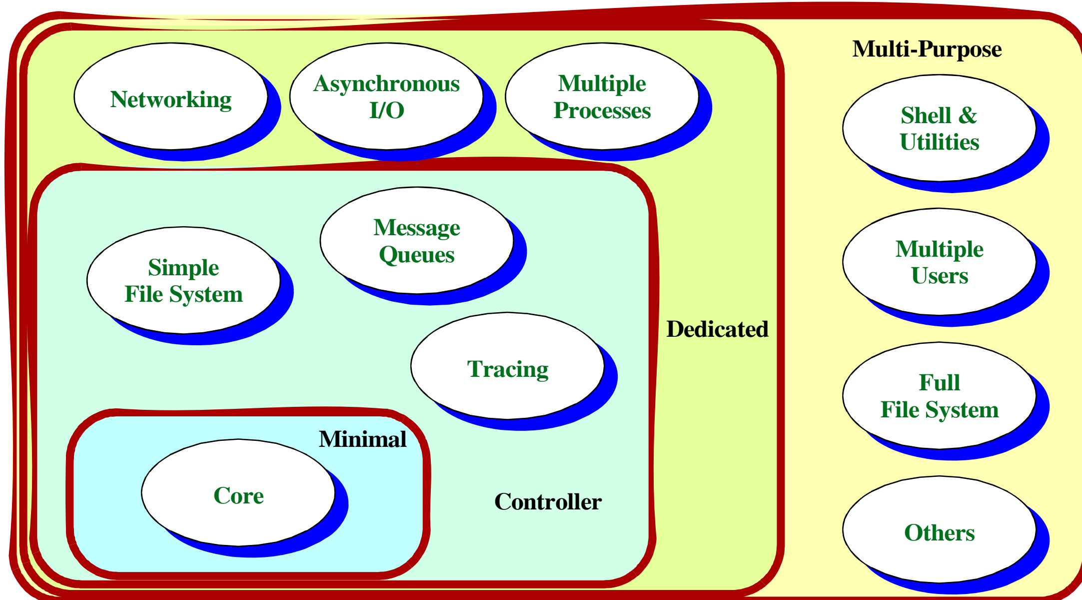

Figure 2. Main units of functionality defined for the larger profiles (PSE52-PSE54)

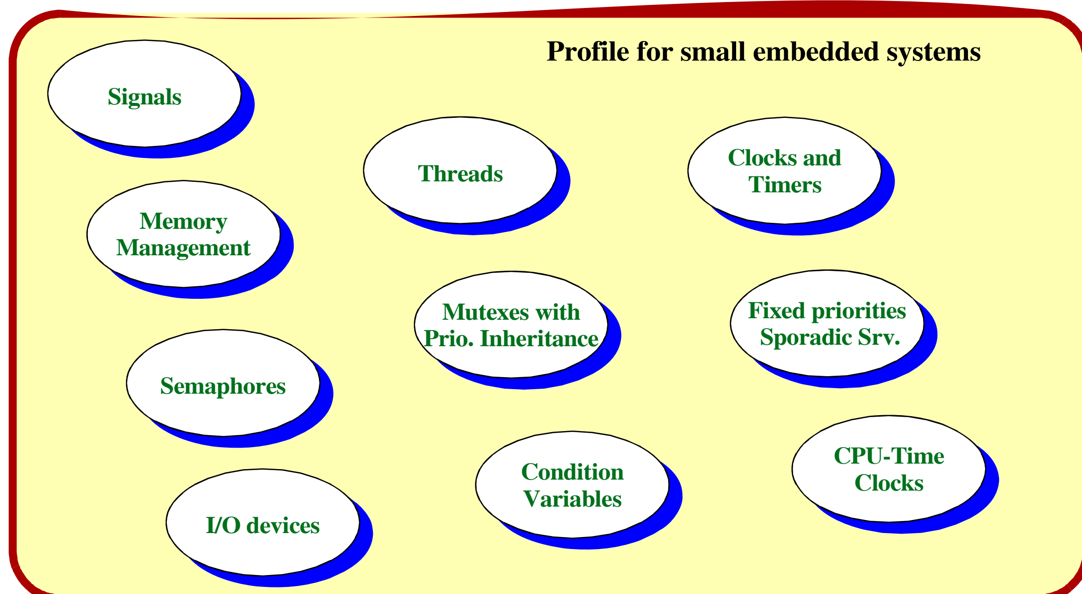

Figure 3. Main services included in the minimal real-time system profile (PSE51)

- Multipurpose Real-Time System (PSE54: Intended for general-purpose systems that also have real-time requirements. This profile adds a full file system, multiple users capability, a shell and common utilities, and many other features.

3. Future

The POSIX standard is a mature technology that receives maintenance in the form of technical corrigenda, but has not had major additions in the past few years.

There are some important features that would be interesting for real-time applications and that are not included in the POSIX standard. They constitute candidates for a future extension of the real-time capabilities of the standard:

- Interrupt management: services to allow the application to manage the processor interrupts, install handlers, and synchronise with them. Although these services should probably not be offered in the OSs addressed to general-purpose systems, they are important in small embedded systems.
- Multiprocessor allocation: it is an important issue with the current trend towards multicore and multiprocessor architectures. Real-time applications need to be able to control the allocation of threads to processors, to achieve predictable timing behaviour.
- Application-defined scheduling. Many high-end applications are running into the need of using scheduling policies that are capable of better exploiting the available resources. Dynamic priority scheduling would thus be desirable or, even better, the ability for the application to define its own scheduling policy.

4. Links and References

[1] ISO/IEC 9945:2003, Information Technology--Portable Operating System Interface (POSIX®). The Institute of Electrical and Electronics Engineers, 2003.

[2] The Open Group. Single UNIX Specification, Version 3 http://www.unix.org/version3/iso_std.html

[3] IEEE Standard 1003.13:2003, "Standard for Information Technology -Standardized Application Environment Profile (AEP)- POSIX Realtime and Embedded Application Support". The Institute of Electrical and Electronics Engineers, 2004.

[4] ISO/IEC 9899:1999 - Programming languages - C http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf

[5] ISO/IEC 14882:2003 Programming Language C++

[6] ISO/IEC 8526:AMD1:2007. Ada 2005 Language Reference Manual (LRM) http://www.ada-ic.org/standards/05rm/html/RM-TTL.html

[7] TIOBE Programming Community Index http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

[8] The RTOS Buyer's Guide http://www.dedicated-systems.com/encyc/buyersguide/products/Dir1048.html

[9] Greg Hawley. "Selecting a Real-Time Operating System". Embedded.com http://i.cmpnet.com/embedded/1999/9903/9903sr.pdf http://www.embedded.com/1999/9903/9903sr.htm

[10] Michael Barr. "Special Report: Choosing an RTOS". Embedded.com http://www.embedded.com/story/OEG20021212S0061

[11] Colin Walls. “Guidelines for using C++ as an alternative to C in embedded designs: Part 1. Why is C++ not more widely used?”. Embedded.com http://www.embedded.com/columns/technicalinsights/207200004?_requestid=51499

[12] B.O. Gallmeister. “Programming for the Real World, POSIX.4”. Sebastopol, CA: O’Reilly & Associates, 1995.

[13] A. Burns and A. Wellings. “Real-Time Systems and Programming Languages (Third Edition): Ada 95, Real-Time Java and Real-Time POSIX Addison Wesley Longman, 2001. ISBN: 0201729881

[14] B. Nichols, D. Buttlar and J. Proulx Farrell. “Pthreads Programming”. O’Reilly & associates, Inc., 1996

[15] S. Kleiman, et al., “Programming with Threads”. Prentice Hall, 1996.