Embedded Systems

4th Stage

Lecture Six

Assist. Prof. Dr. Yasir Amer Abbas

Computer Engineering Department

2022

Embedded System Software

Lecture Six

6. Embedded System Software

Introduction

Not all components of embedded systems need to be designed from scratch. Instead, there are standard components that can be reused. These components comprise knowledge from earlier design efforts and constitute intellectual property (IP). IP reuse is one key technique in coping with the increasing complexity of designs. The term "IP reuse" frequently denotes the reuse of hardware. However, reusing hardware is not enough. The software components need to be reused as well. **Therefore, the platform-based design methodology advocated comprises the reuse of hardware and software IP.**

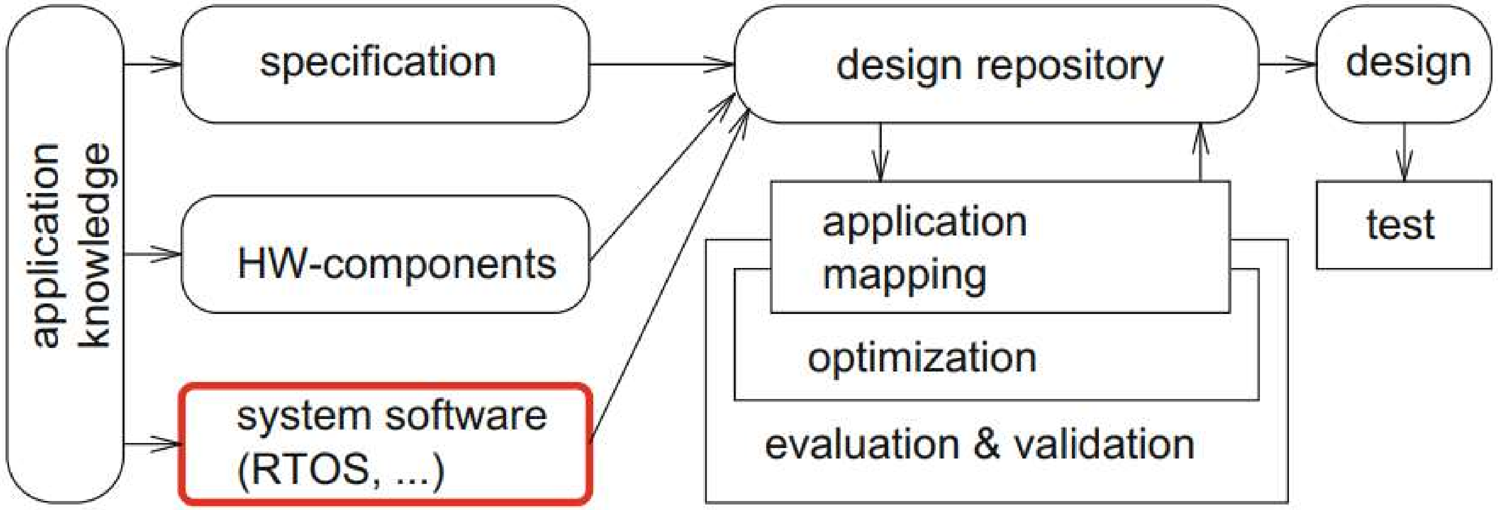


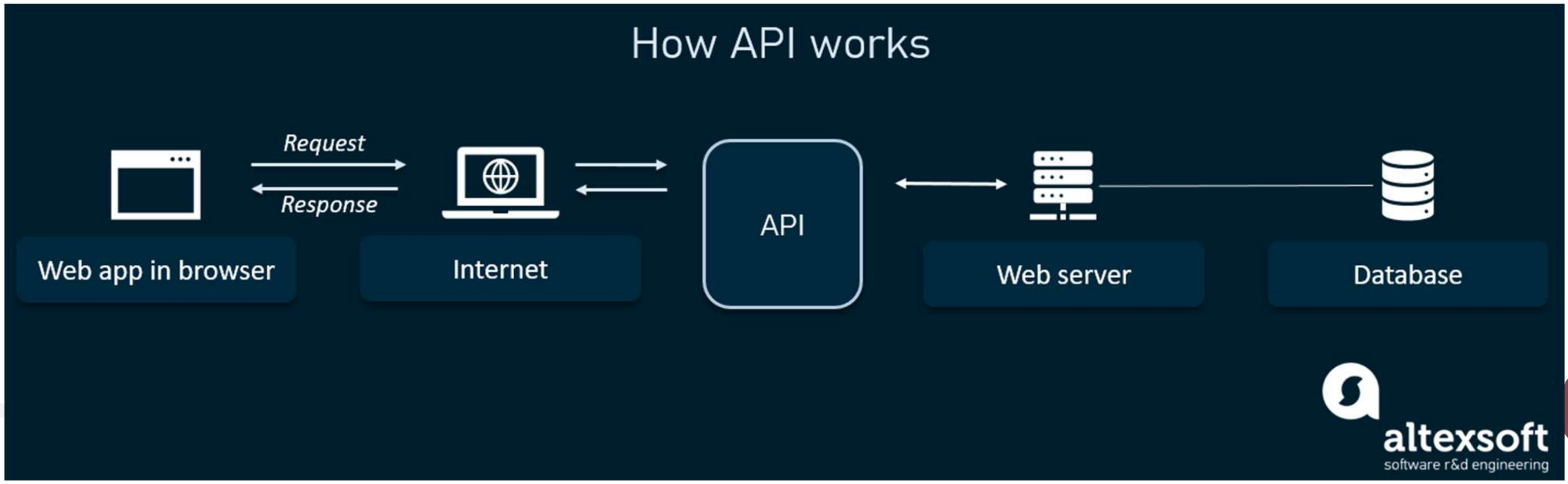
Fig. 1 Simplified design information flow

6. Embedded System Software

Introduction

Standard software components that can be reused include system software components such as embedded operating systems (OSs) and **middleware**.

The last term denotes software that provides an intermediate layer between the OS and application software. We **include libraries for communication** as a special case of middleware. Such libraries extend the basic communication facilities provided by operating systems. Also, we consider **real-time databases to be a second class of middleware**. Calls to standard software components may already need to be included in the specification. Therefore, information about the **application programming interface (API)** of these standard components may already be needed for completing executable specifications.



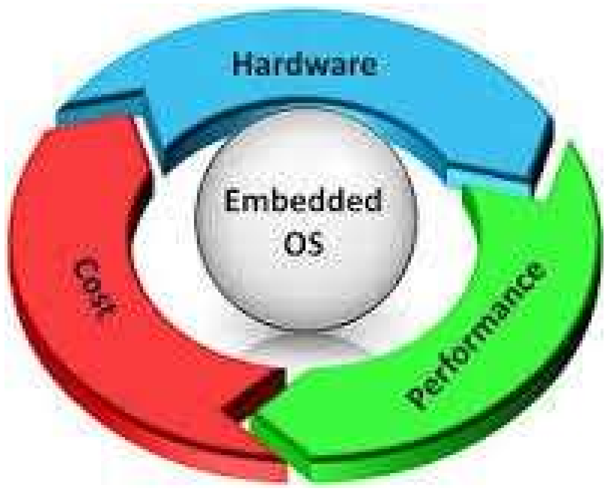
6. Embedded System Software

6.1 Embedded Operating Systems

6.1.1 General Requirements

An embedded operating system (OS**) is a specialized operating system designed to perform a specific task for a device that is not a computer.**

An embedded operating system's main job is to run the code that allows the device to do its job. *The embedded OS also makes the device's hardware accessible to the software that is running on top of the OS.*



6. Embedded System Software

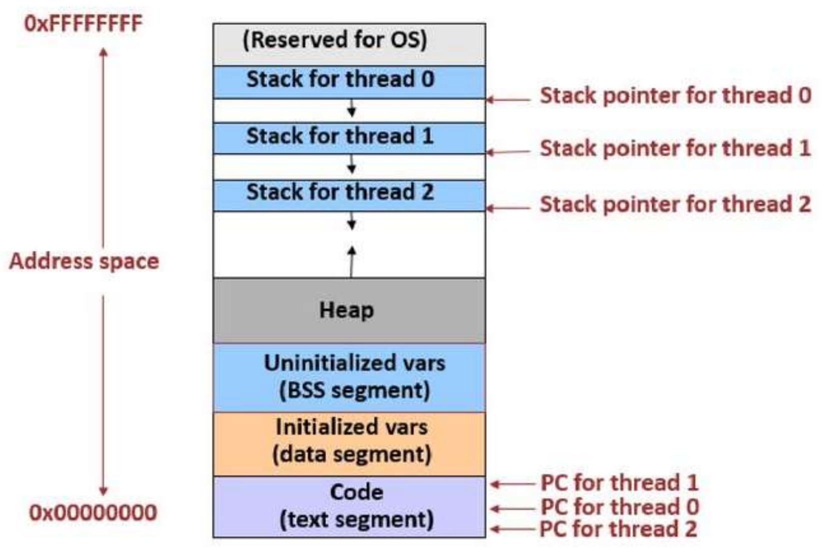
6.1 Embedded Operating Systems

6.1.1 General Requirements

Except for very simple systems, scheduling, context switching, and I/O require the support of an operating system suited for embedded applications. Context switching algorithms multiplex processors such that each process seems to have its own processor.

For systems with virtual addressing, we can distinguish between different address spaces and between processes and threads. **Each process has its own address space, whereas several threads may share an address space.** Context switches which change the address space require more time than those which do not. Threads sharing an address space will typically communicate via shared memory. ***Operating systems must provide communication and synchronization methods for threads and processes.***

Process Address Space w/ Threads



All threads in a single process share the same address space!

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.1 General Requirements

The essential features of embedded operating systems:

Due to the large variety of embedded systems, there is also a large variety of requirements for the functionality of embedded OSs. Due to efficiency requirements, it is not possible to work with OSs which provide the union of all functionalities.

For most applications, the OS must be small. Hence, we need operating systems which can be **flexibly tailored** toward the application at hand. **Configurability** is therefore one of the main characteristics of embedded OSs. There are various techniques of implementing configurability, including

1. Object orientation.
2. Aspect-oriented programming
3. Conditional compilation.
4. Advanced compile-time evaluation.
5. Linker-based removal of unused functions.

4. Embedded System Software

6.1 Embedded Operating Systems

6.1.1 General Requirements

The essential features of embedded operating systems:

There is a **large variety of peripheral devices employed in embedded systems**. Many embedded systems do not have a hard disk, a keyboard, a screen, or a mouse.

There is effectively no device that needs to be supported by all variants of the OS, except maybe the system timer. Frequently, applications are designed to handle particular devices. In such cases, devices are not shared between applications, and hence, there is no need to manage the devices by the OS. Due to the large variety of devices, it would also be difficult to provide all required device drivers together with the OS. Hence, it makes sense to decouple OS and drivers by using special processes instead of integrating their drivers into the kernel of the OS**. Due to the limited speed of many embedded peripheral devices, there is also no need for an integration into the OS in order to meet performance requirements. This may lead to a different stack of software layers.**

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.1 General Requirements

The essential features of embedded operating systems:

For PCs, some drivers, such as disk drivers, network drivers, or audio drivers, are implicitly assumed to be present*. *They are implemented at a very low level of the stack. The application software and middleware are implemented on top of the application programming interface, which is standard for all applications.

For an embedded OS, device drivers are implemented on top of the kernel. Applications and middleware may be implemented on top of appropriate drivers, not on top of a standardized API of the OS (see Fig. 2). Drivers may even be included in the application itself.

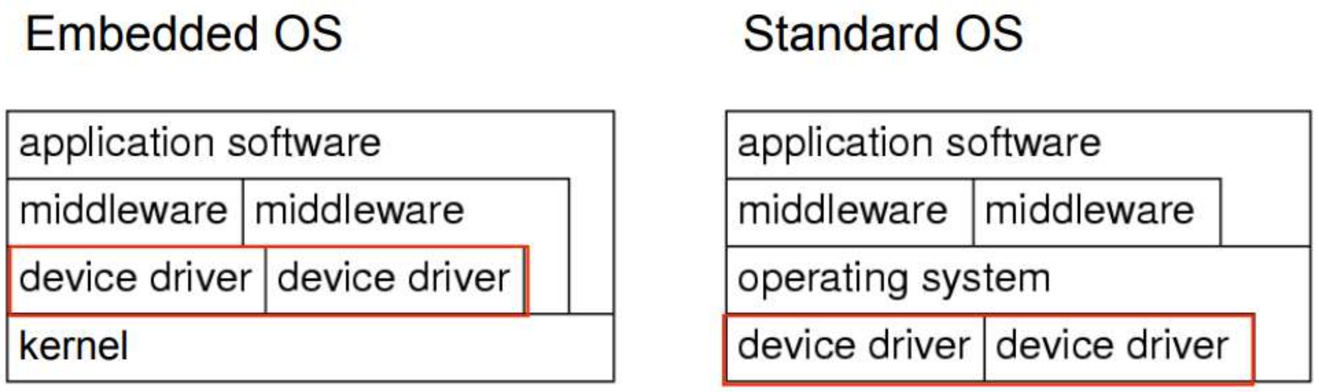


Fig. 2

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.1 General Requirements

The essential features of embedded operating systems:

Protection mechanisms are sometimes not necessary, since embedded systems are sometimes designed for a single purpose (they are not supposed to support so-called multiprogramming). Untested programs have traditionally hardly ever been loaded. After the software has been tested, it could be assumed to be reliable. This also applies to input/output. In contrast to desktop applications, it is possibly not always necessary to implement I/O instructions as privileged instructions and processes can sometimes be allowed to do their own I/O. This matches nicely with the previous item and reduces the overhead of I/O operations.

Interrupts can be connected to any thread or process. Using OS service calls, we can request the OS to start or stop them if certain interrupts happen. We could even store the start address of a thread or process in the interrupt vector address table, but this technique is very dangerous, since the OS would be unaware of the thread or process actually running. Also composability may suffer from this: If a specific thread is directly connected to some interrupt, then it may be difficult to add another thread which also needs to be started by some event. Application specific device drivers (if used) might also establish links between interrupts and threads and processes.

•

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.2 Real-Time Operating Systems

Many embedded systems are real-time (RT) systems, and hence, the OS used in these systems **must be a real-time operating system** (RTOS). Real-time operating system is an operating system that supports the construction of real-time systems.

What does it take to make an OS an RTOS? There are four key requirements:

A. The timing behaviour of the OS must be predictable.

For each service of the OS, an upper bound on the execution time must be guaranteed. In practice, there are various levels of predictability. For example, there may be sets of OS service calls for which an upper bound is known and for which there is not a significant variation of the execution time. Calls like "get me the time of the day" may fall into this class. For other calls, there may be a huge variation. Calls like "get me 4MB of free memory" may fall into this second class. In particular, the scheduling policy of any RTOS must be deterministic.

There may also be times during which interrupts must be disabled to avoid interferences between components of the OS. Less importantly, they can also be disabled to avoid interferences between processes. The periods during which interrupts are disabled must be quite short in order to avoid unpredictable delays in the processing of critical events. For RTOSs implementing file systems still using hard disks, it may be necessary to implement contiguous files (files stored in contiguous disk areas) to avoid unpredictable disk head movements.

4. Embedded System Software

6.1 Embedded Operating Systems

6.1.2 Real-Time Operating Systems

B. Some systems require the OS to manage time.

The OS must manage the scheduling of threads and processes. Scheduling can be defined as mapping from sets of threads or processes to intervals of execution time (including the mapping to start times as a special case) and to processors (in case of multiprocessor systems).

Also, the OS possibly has to be aware of deadlines so that the OS can apply appropriate scheduling techniques. There are, however, cases in which scheduling is done completely off-line, and the OS only needs to provide services to start threads or processes at specific times or priority levels.

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.2 Real-Time Operating Systems

C. The OS must manage the scheduling of threads and processes.

This management is mandatory if internal processing is linked to an absolute time in the physical environment. Physical time is described by real numbers. In computers, discrete time standards are typically used instead. The precise requirements may vary:

1. ***In some systems, synchronization with global time standards is necessary***. In this case, **global clock synchronization** is performed. Two standards are available for this:

– **Universal Time Coordinated (UTC)**: UTC is defined by astronomical standards. Due to variations regarding the movement of the earth, this standard has to be adjusted from time to time.

– **International atomic time:** This standard is free of any artificial artifacts. Some connection to the environment is used to obtain accurate time information. External synchronization is typically based on wireless communication standards such as the global positioning system (GPS) or mobile networks.
2. If embedded systems are used in a network, it is frequently sufficient to synchronize time information within the network. Local clock synchronization can be used for this. In this case, connected embedded systems try to agree on a consistent view of the current time.

3. There may be cases in which provision for precise local delays is all that is needed.

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.2 Real-Time Operating Systems

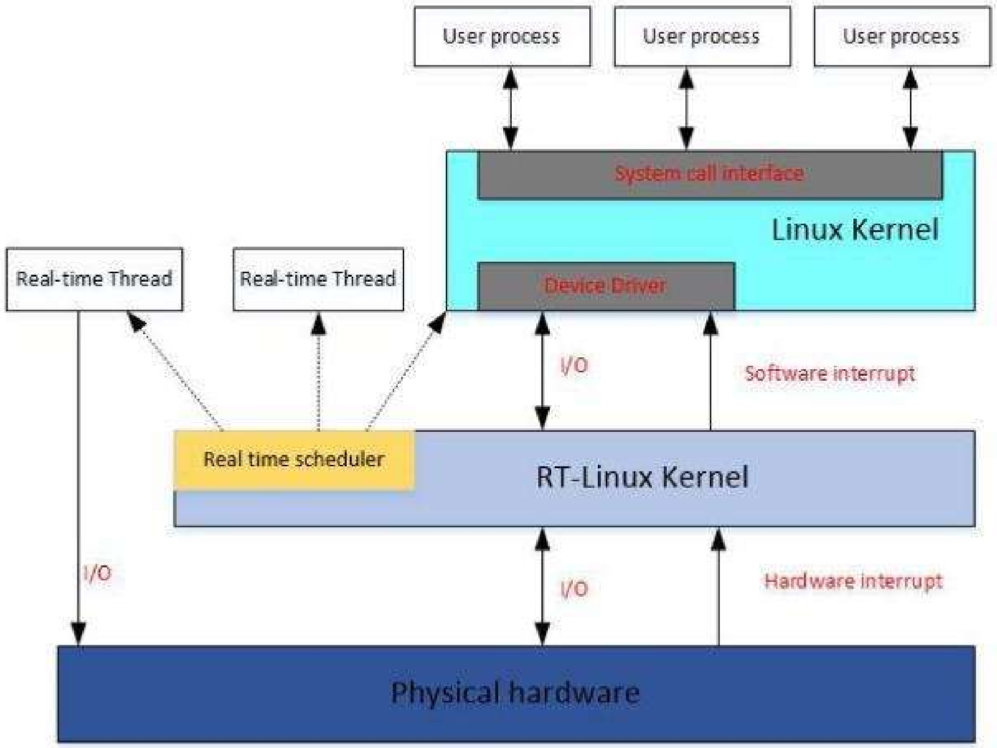
D. The OS must be fast. An operating system meeting all the requirements mentioned so far would be useless if it were very slow. Therefore, the OS must obviously be fast.

Each RTOS includes a so-called real-time OS kernel. This kernel manages the resources which are found in every real-time system, including the processor, the memory, and the system timer. Major functions in the kernel include the process and thread management, inter-process synchronization and communication, time management, and memory management.

E. Fast proprietary kernels: According to Gupta, *"for complex systems, these kernels are inadequate, because they are designed to be fast, rather than to be predictable in every respect."* Examples include QNX, PDOS, VCOS, VTRX32, and VxWorks.

F. Real-time extensions to standard OSs: In order to take advantage of comfortable mainstream operating systems, hybrid systems have been developed. For such systems, there is an RT-kernel running all RT-processes.

RT-kernel running all RT-processes.



6. Embedded System Software

6.1 Embedded Operating Systems

6.1.3 Virtual Machines

In certain environments, it may be useful to emulate several processors on a single real processor. This is possible with **virtual machines** executed on the bare hardware. On top of such a virtual machine, several operating systems can be executed. Obviously, this allows several operating systems to be run on a single processor. For embedded systems, this approach has to be used with care since the temporal behaviour of such an approach may be problematic and timing predictability may be lost. Nevertheless, sometimes this approach may be useful. For example, we may need to integrate several legacy applications using different operating systems on a single hardware processor. A full coverage of virtual machines is beyond the scope of this lecture. PikeOS is an example of a virtualization concept dedicated toward embedded systems. PikeOS allows the system's resources (e.g., memory, I/O devices, CPU-time) to be divided into separate subsets. PikeOS comes with a small micro-kernel. Several operating systems, application programming interfaces (APIs), and run-time environments (RTEs) can be implemented on top of this kernel (see Fig. 3)

6. Embedded System Software

6.1 Embedded Operating Systems

6.1.3 Virtual Machines

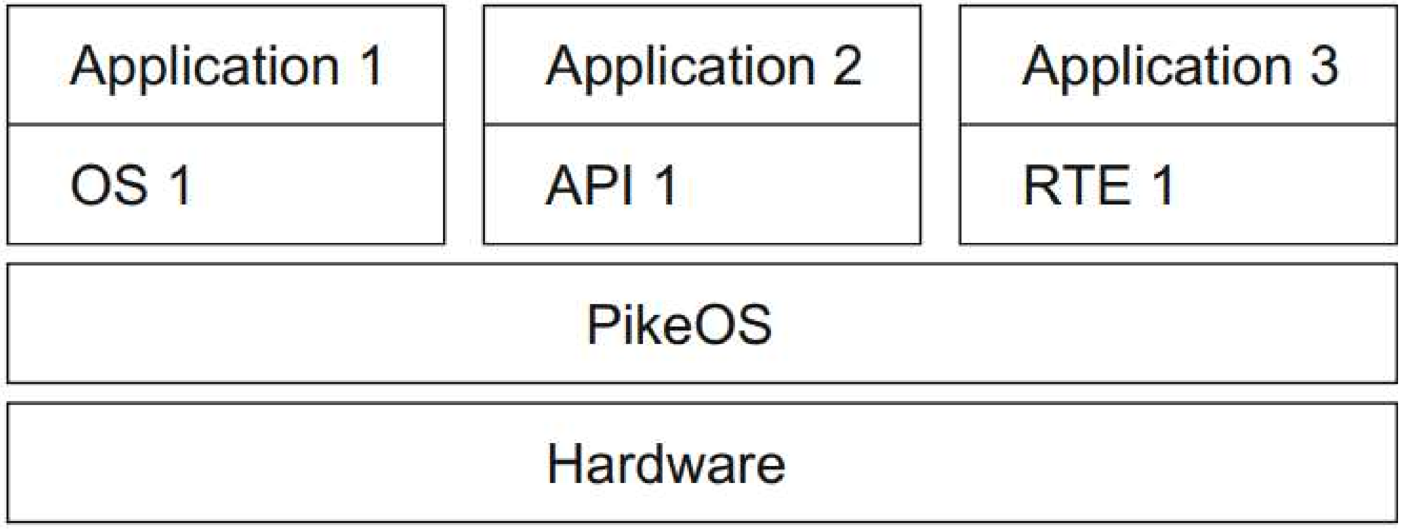
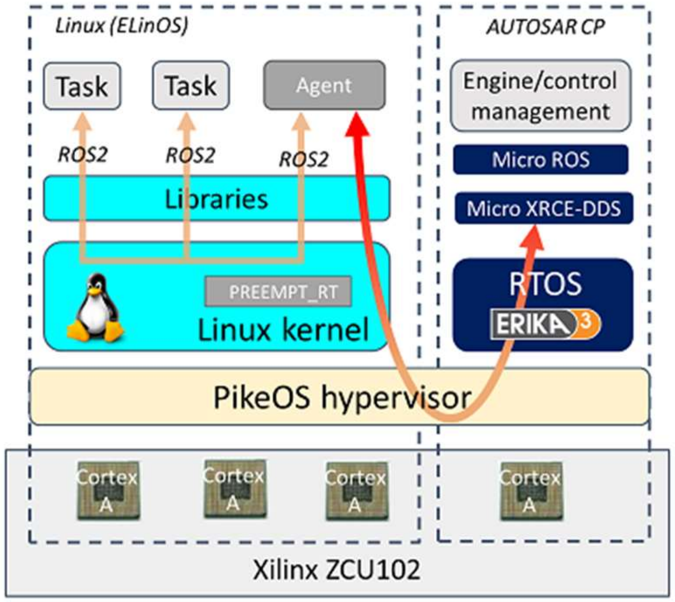


Fig. 3 PikeOS virtualization (©SYSGO)

6.2 Erika

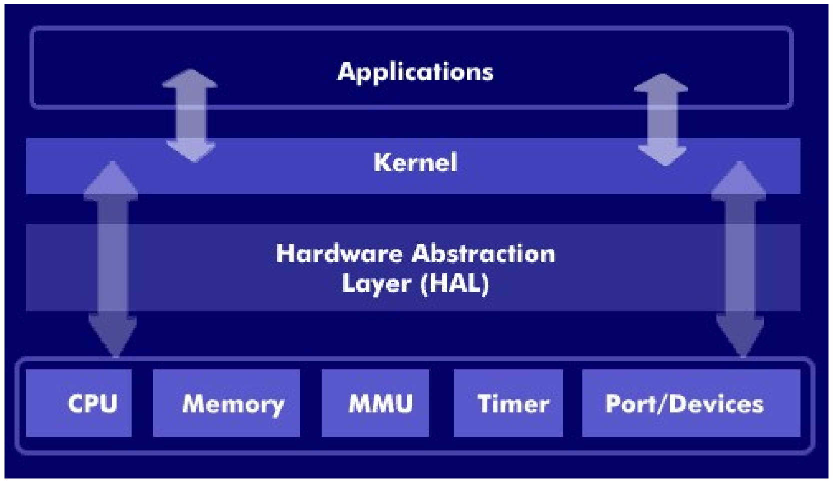
Erika Enterprise is the first open-source Free RTOS that has been certified OSEK/VDX (open systems and the corresponding interfaces for automotive electronics) compliant

Main Features:
Hard Real-Time support with Fixed Priority Scheduling and Immediate Priority Ceiling;
Support for Earliest Deadline First (EDF) and Resource Reservation Schedulers;
1-4 Kb Flash footprint, suitable for 8 to 32 bit microcontrollers;
Support for multi-core platforms;
Support for stack sharing among tasks;
Easy configuration using RT-Druid with Eclipse plugins;



Hardware abstraction layers

- Hardware abstraction layers (HALs) provide a way for accessing hardware through a hardware-independent application programming interface (API). For example, we could come up with a hardware-independent technique for accessing timers, irrespective of the addresses to which timers are mapped. Hardware abstraction layers are used mostly between the hardware and operating system layers. They provide software intellectual property (IP), but they are neither part of operating systems nor can they be classified as middleware

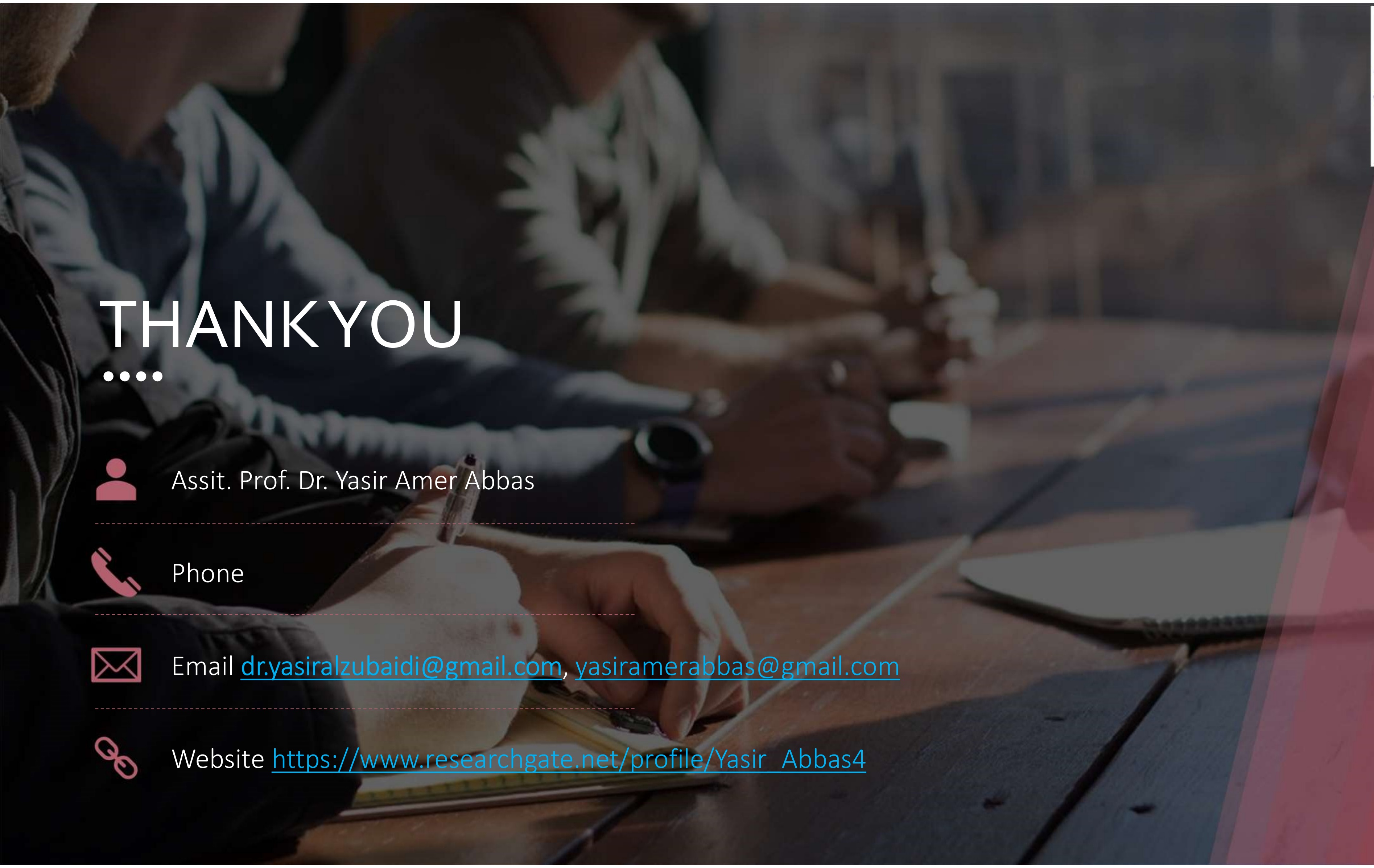


• Middleware

- Communication libraries provide a means for adding communication functionality to languages lacking this feature. They add communication functionality on top of the basic functionality provided by operating systems. Due to being added on top of the OS, they can be independent of the OS (and obviously also of the underlying processor hardware). As a result, we will obtain communication-oriented cyberphysical systems. Such communication is needed for the Internet of Things (IoT). There is a trend toward supporting communication within some local system as well as communication over longer distances. The use of Internet protocols in general is becoming more popular. Frequently, such protocols enable secure communication, based on en- and decryption . The corresponding algorithms are a special case of middleware.

Real-Time Databases

Databases provide a convenient and structured way of storing and accessing information. Accordingly, data bases provide an API for writing and reading information. A sequence of read and write operations is called a transaction. Transactions may have to be aborted for a variety of reasons: there could be hardware problems, deadlocks, problems with concurrency control, etc. A frequent requirement is that transactions do not affect the state of the database unless they have been executed to their very end. Hence, changes caused by transactions are normally not considered to be final until they have been committed. Most transactions are required to be atomic. This means that the end result (the new state of the database) generated by some transaction must be the same as if the transaction has been fully completed or not at all. Also, the database state resulting from a transaction must be consistent. Consistency requirements include, for example, that the values from read requests belonging to the same transaction are consistent (do not describe a state which never existed in the environment modeled by the database). Furthermore, to some other user of the database, no intermediate state resulting from a partial execution of a transaction must be visible (the transactions must be performed as if they were executed in isolation). Finally, the results of transactions should be persistent. This property is also called durability of the transactions. Together, the four properties printed in bold are known as ACID properties



THANK YOU

Assit. Prof. Dr. Yasir Amer Abbas

Phone

Email dr.yasiralzubaidi@gmail.com, yasiramerabbas@gmail.com

Website https://www.researchgate.net/profile/Yasir_Abbas4