Abstraction Layer

Related terms:

Energy Efficiency, Reuse, Device Driver, Header File, Management Process, Middleware

View all Topics

Software Reuse By Design in Embedded Systems

Jim Trudeau, in Software Engineering for Embedded Systems, 2013

Implementing reuse by layers

A hardware abstraction layer (HAL) implements a reusable hardware interface in software. You can think of this as the "hardware section" of an RTOS or code library generalized into a multi-purpose API to access the hardware layer. The RTOS or application (if necessary) can call the HAL without touching hardware. The HAL acts as a buffer that insulates all the code above it from knowledge of or dependency on any hardware details. Figure 9.3 shows what this looks like as a block diagram.

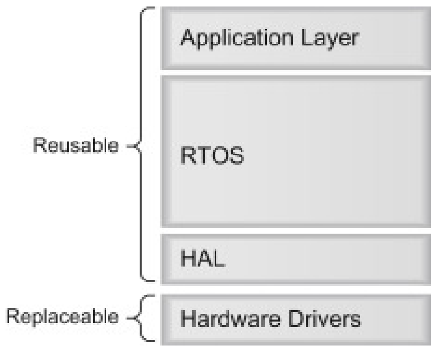


Figure 9.3. A hardware abstraction layer increases reusability.

The HAL itself, in theory, is fully reusable and does not need rewriting when you port to new hardware. In practice, the HAL typically needs tweaking to accommodate some idiosyncrasy of the new platform, but as a stable API it remains unchanged from the perspective of the higher levels of software. What this means from a software reuse perspective is that, when you port the code to new hardware, all access into and out of the hardware-sensitive code remains unchanged. In *theory*, you can swap in any hardware layer and it still works.

software reuse perspective is that, when you port the code to new hardware, all access into and out of the hardware-sensitive code remains unchanged. In *theory*, you can swap in any hardware layer and it still works.

OK, we see that *in theory*, using a HAL can help us port hardware. What about our RTOS dependency? Remember that one of the fundamental barriers to reuse at the software level is how code is locked into an RTOS. What can we do about that?

The answer is, it's the same trick: an RTOS abstraction layer. The fundamental technique is indirection: instead of calling the OS directly, you call a similar function in the abstraction layer (called the porting layer). The porting layer calls the RTOS (Figure 9.4).

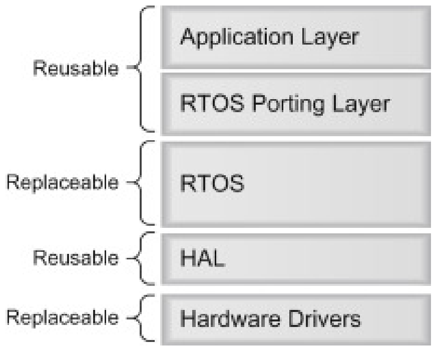


Figure 9.4. Implementing a porting layer allows RTOS independence.

Even if you have not implemented an RTOS porting layer in your software initially, you can still take advantage of this technique after the fact. Like a magician's sleight of hand, you substitute a header file that mimics the RTOS's API. Where your code makes an RTOS call, the new header file redirects the call to your own function. The new function may perform some setup or "wrapper" work if necessary, and then call the new operating system's parallel function to do the work.

This is perhaps the simplest porting trick in the world. In this way you do not need to substantially rewrite code. Just link in the appropriate header files, provide the necessary glue code, and you get to use a new RTOS.

You probably don't have to invent this from scratch. There are commercial endeavors that follow this strategy. One example is MapuSoft, which provides tools to port from multiple OSes such as VxWorks, pSOS, Nucleus, POSIX, micro-ITRON, and others.

The advantage is obvious. Suddenly you aren't locked to that RTOS anymore, and you did not have to change a single line of code. Wow, talk about a free lunch!

The disadvantages are perhaps not quite so obvious, but depending on your circumstances may be worth the price of admission.

A porting layer adds a layer of indirection, hence some inefficiency. If nothing else there's a function call jump in between. There may be an increase in code complexity and size. There may be some performance hit as a result. However, a port to a new

OS is often happening at the same time as a port to new hardware, and the new processor may accommodate the performance requirements quite nicely. Also, the new RTOS may be inherently more efficient. In fact you may see things get better!

OS is often happening at the same time as a port to new hardware, and the new processor may accommodate the performance requirements quite nicely. Also, the new RTOS may be inherently more efficient. In fact you may see things get better!

More importantly, all operating systems are not created equal. Functions will not be precisely parallel, so the translation will not be precisely 1:1. For example, the interrupt-handling scheme may be different; how the new RTOS handles priority queues could be subtly changed. You will need to test carefully for functionality and quality.

Nonetheless, this is one strategy for freeing yourself from the tyranny of your code being intricately tied to an operating system. When the only alternative is a complete rewrite, this may have great appeal and could get you to a design goal more quickly.

> Read full chapter

Real-Time Framework Implementation

Miro Samek, in Practical UML Statecharts in C/C++ (Second Edition), 2009

7.3.3 Internal QF Macros for Interrupt Locking/Unlocking

The QF platform abstraction layer (PAL) uses the interrupt locking/unlocking macros QF_INT_LOCK(), QF_INT_UNLOCK(), and QF_INT_KEY_TYPE in a slightly modified form. The PAL defines internally the parameterless macros, shown in Listing 7.6. Please note the trailing underscores in the internal macros' names.

Listing 7.6

Internal macros for interrupt locking/unlocking (file <qp>\qpc\qf\source\qf_pkg.h)

```
#ifndef QF_INT_KEY_TYPE /* simple unconditional interrupt locking/unlocking */
#define QF_INT_LOCK_KEY_
#define QF_INT_LOCK_() QF_INT_LOCK(ignore)
#define QF_INT_UNLOCK_() QF_INT_UNLOCK(ignore)
#else /* policy of saving and restoring interrupt status */
#define QF_INT_LOCK_KEY_ QF_INT_KEY_TYPE intLockKey_;
#define QF_INT_LOCK_() QF_INT_LOCK(intLockKey_)
#define QF_INT_UNLOCK_() QF_INT_UNLOCK(intLockKey_)
#endif
```

The internal macros QF_INT_LOCK_KEY_, QF_INT_LOCK_(), and QF_INT_UNLOCK_() enable writing the same code for the case when the interrupt key is defined and when it is not. The following code snippet shows the usage of the internal QF macros. Convince yourself that this code works correctly for both interrupt-locking policies.

The internal macros QF_INT_LOCK_KEY_, QF_INT_LOCK_(), and QF_INT_UNLOCK_() enable writing the same code for the case when the interrupt key is defined and when it is not. The following code snippet shows the usage of the internal QF macros. Convince yourself that this code works correctly for both interrupt-locking policies.

```
void QF_service_xyz(arguments) {
    QF_INT_LOCK_KEY_
    . . .
    QF_INT_LOCK_();
    . . . /* critical section of code */
    . . .
    QF_INT_UNLOCK_();
}
```

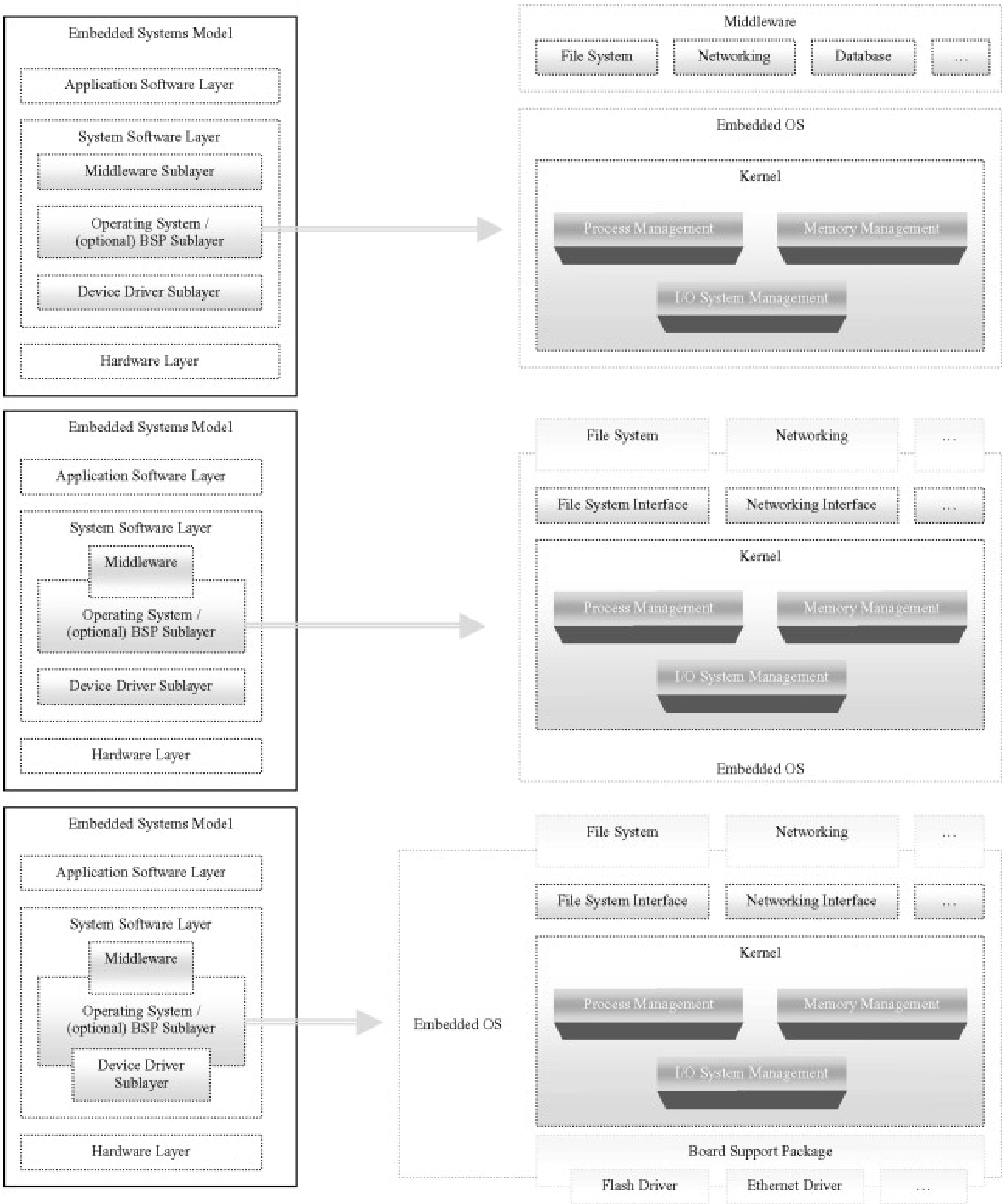
> Read full chapter

The Fundamentals in Understanding Networking Middleware

Tammy Noergaard, in Demystifying Embedded Systems Middleware, 2010

4.4 An Embedded OS and Networking I/O APIs

A common method of providing an abstraction layer to simplify software development, managing an embedded device's hardware and software resources, as well as insuring efficient and reliable operation, is the utilization of an embedded operating system (OS) within a design. In addition to process, memory, and I/O system management components within its *kernel*, an embedded OS may also provide additional I/O system management functionality for networking protocol libraries (see Figures 4.19a and 4.19b).


Embedded Systems Model
Application Software Layer
System Software Layer
Middleware Sublayer
Operating System / (optional) BSP Sublayer
Device Driver Sublayer
Hardware Layer
Middleware
File System
Networking
Database
...
Embedded OS
Kernel
Process Management
Memory Management
I/O System Management
Embedded Systems Model
Application Software Layer
System Software Layer
Middleware
Operating System / (optional) BSP Sublayer
Device Driver Sublayer
Hardware Layer
File System
Networking
...
File System Interface
Networking Interface
...
Kernel
Process Management
Memory Management
I/O System Management
Embedded OS
Embedded Systems Model
Application Software Layer
System Software Layer
Middleware
Operating System / (optional) BSP Sublayer
Device Driver Sublayer
Hardware Layer
File System
Networking
...
File System Interface
Networking Interface
...
Embedded OS
Kernel
Process Management
Memory Management
I/O System Management
Board Support Package
Flash Driver
Ethernet Driver
...

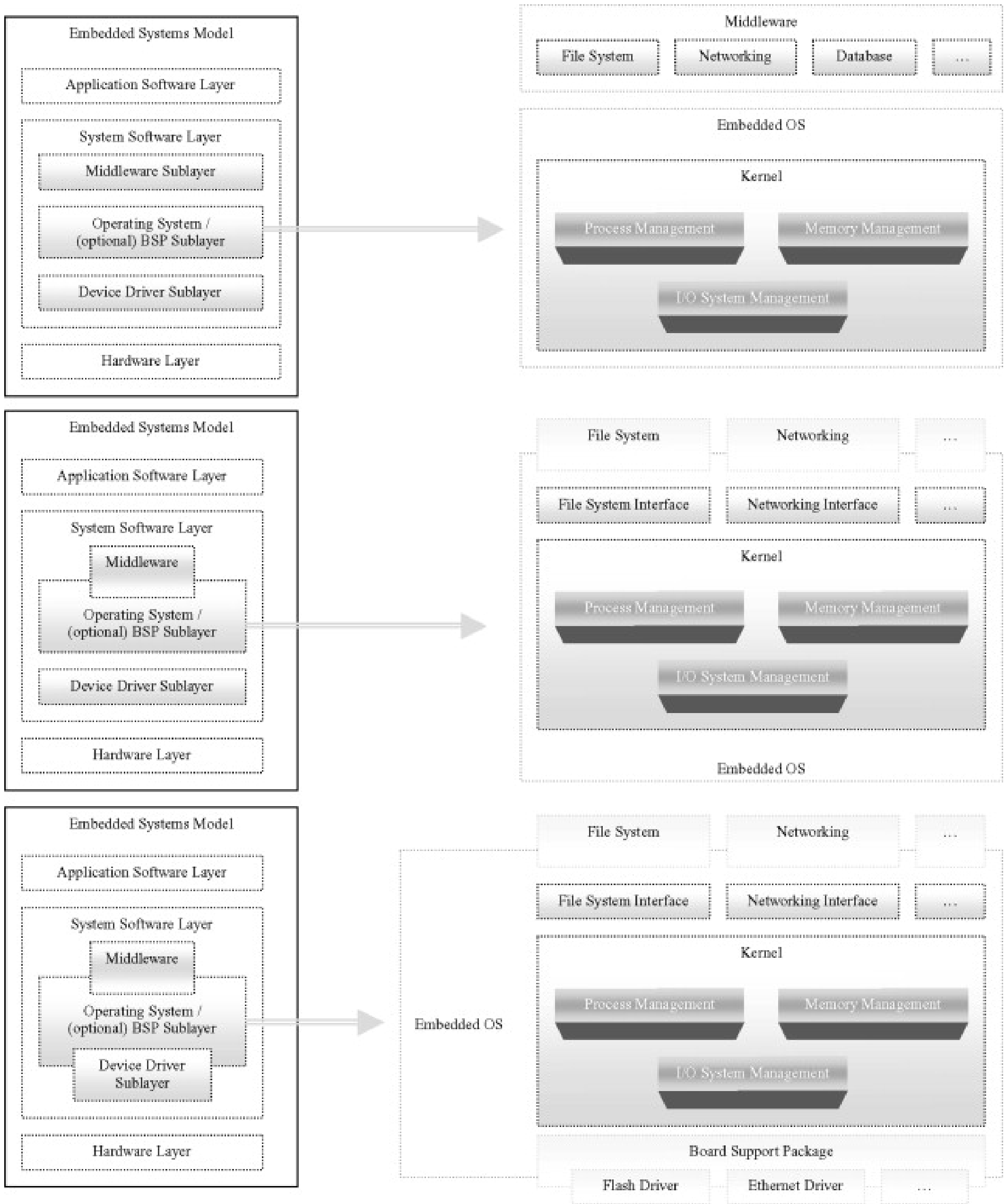


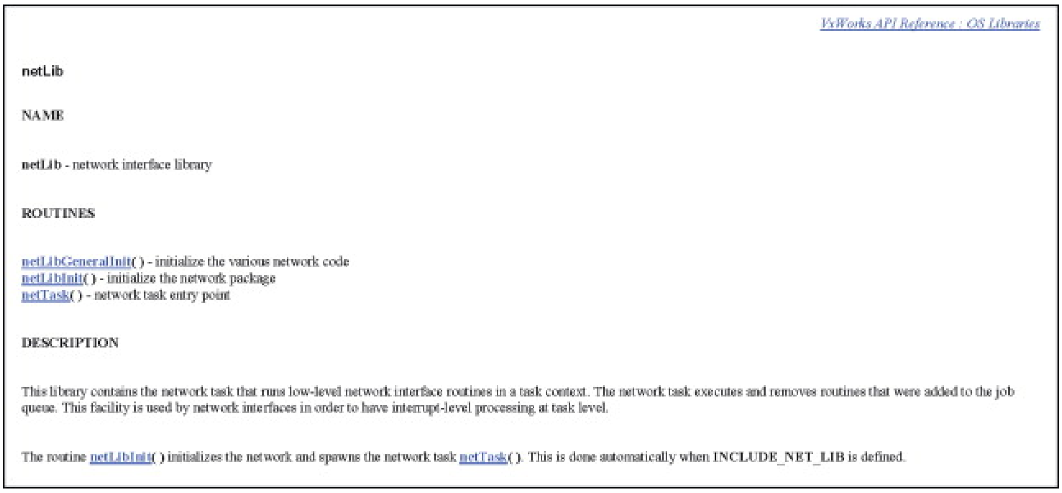
Figure 4.19. (a)Example OS Permutations; (b) Example OS Components

While networking middleware can of course be written to access device driver functionality directly, an embedded OS can also include an abstraction layer API that allows for device driver access by middleware. When providing device access, or any type of I/O access to overlying networking libraries, many OS APIs categorize and abstract the underlying device drivers as some combination of:

- **Character**, a driver that allows hardware access via a (character) byte stream

- **Block**, a driver that allows hardware access via some smallest addressable set of bytes at any given time
- **Network**, a driver that allows hardware access to data in the form of networking packets
- **Virtual**, a driver that allows I/O access to virtual (software) devices
- **Miscellaneous Monitor and Control**, a driver that allows I/O access to hardware that is not accessible via the other categories above.

Figure 4.20 shows an example of a VxWorks network device interface library available to middleware for usage—this example is a subset of VxWorks available functionality for network interfacing, buffering, and monitoring. Overlying middleware software layers then have the option of utilizing functions, such as these types of functions provided by the OS layer, to process and manage incoming and outgoing networking data.



VxWorks API Reference : OS Libraries

netLib

NAME

netLib - network interface library

ROUTINES

netLibGeneralInit( ) - initialize the various network code
netLibInit( ) - initialize the network package
netTask( ) - network task entry point

DESCRIPTION

This library contains the network task that runs low-level network interface routines in a task context. The network task executes and removes routines that were added to the job queue. This facility is used by network interfaces in order to have interrupt-level processing at task level.

The routine netLibInit( ) initializes the network and spawns the network task netTask( ). This is done automatically when INCLUDE_NET_LIB is defined.

The routine netHelp( ) in usrLib displays a summary of the network facilities available from the VxWorks shell.

INCLUDE FILES

netLib.h

SEE ALSO

routeLib, hostLib, netDrv, netHelp( ).

OS Libraries : Routines

netLibGeneralInit( )

NAME

netLibGeneralInit( ) - initialize the various network code

SYNOPSIS

```
STATUS netLibGeneralInit (void)
```

DESCRIPTION

This code use to be in netLibInit. With virtual stacks, we need these specific routines to be executed on a per virtual stack bases.

RETURNS

OK/ERROR

SEE ALSO

netLib

OS Libraries : Routines

netLibInit( )

NAME

netLibInit( ) - initialize the network package

SYNOPSIS

```
STATUS netLibInit (void)
```

DESCRIPTION

This creates the network task job queue, and spawns the network task netTask( ). It should be called once to initialize the network. This is done automatically when **INCLUDE_NET_LIB** is defined.

PROTECTION DOMAINS

This function can only be called from within the kernel protection domain.

RETURNS

OK, or ERROR if network support cannot be initialized.

The routine netHelp( ) in usrLib displays a summary of the network facilities available from the VxWorks shell.

INCLUDE FILES

netLib.h

SEE ALSO

routeLib, hostLib, netDrv, netHelp( ).

OS Libraries : Routines

netLibGeneralInit( )

NAME

netLibGeneralInit( ) - initialize the various network code

SYNOPSIS

```
STATUS netLibGeneralInit (void)
```

DESCRIPTION

This code use to be in netLibInit. With virtual stacks, we need these specific routines to be executed on a per virtual stack bases.

RETURNS

OK/ERROR

SEE ALSO

netLib

OS Libraries : Routines

netLibInit( )

NAME

netLibInit( ) - initialize the network package

SYNOPSIS

```
STATUS netLibInit (void)
```

DESCRIPTION

This creates the network task job queue, and spawns the network task netTask( ). It should be called once to initialize the network. This is done automatically when **INCLUDE_NET_LIB** is defined.

PROTECTION DOMAINS

This function can only be called from within the kernel protection domain.

RETURNS

OK, or ERROR if network support cannot be initialized.

SEE ALSO

netLib, usrConfig, netTask( )

---

netTask( )

NAME

netTask( ) - network task entry point

SYNOPSIS

```
void netTask (void)
```

DESCRIPTION

This routine is the VxWorks network support task. Most of the VxWorks network runs in this task's context.

NOTE

To prevent an application task from monopolizing the CPU if it is in an infinite loop or is never blocked, the priority of netTask( ) relative to an application may need to be adjusted. Network communication may be lost if netTask( ) is "starved" of CPU time. The default task priority of netTask( ) is 50. Use taskPrioritySet( ) to change the priority of a task.

This task is spawned by netLibInit( ).

PROTECTION DOMAINS

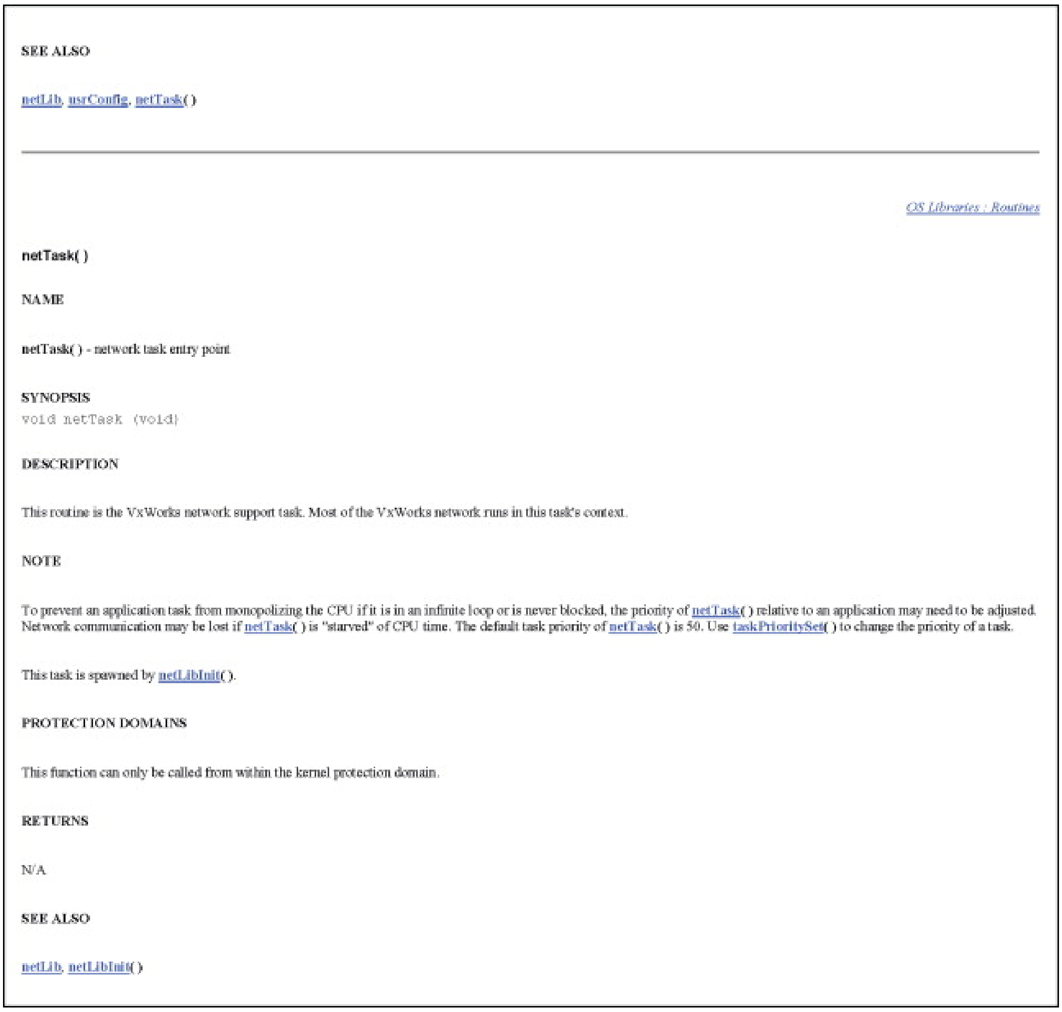
This function can only be called from within the kernel protection domain.

RETURNS

N/A

SEE ALSO

netLib, netLibInit( )



SEE ALSO

netLib, usrConfig, netTask( )

OS Libraries : Routines

netTask( )

NAME

netTask( ) - network task entry point

SYNOPSIS

```
void netTask (void)
```

DESCRIPTION

This routine is the VxWorks network support task. Most of the VxWorks network runs in this task's context.

NOTE

To prevent an application task from monopolizing the CPU if it is in an infinite loop or is never blocked, the priority of netTask( ) relative to an application may need to be adjusted. Network communication may be lost if netTask( ) is "starved" of CPU time. The default task priority of netTask( ) is 50. Use taskPrioritySet( ) to change the priority of a task.

This task is spawned by netLibInit( ).

PROTECTION DOMAINS

This function can only be called from within the kernel protection domain.

RETURNS

N/A

SEE ALSO

netLib, netLibInit( )

Figure 4.20. Example of Ethernet Device Driver Public Library under VxWorks7

> Read full chapter

Communicating Pictures: Delivery Across Networks

David R. Bull, in Communicating Pictures, 2014

11.5.1 Network abstraction

Standards since H.264/AVC have adopted the principle of a Network Abstraction Layer (NAL) that provides a common syntax, applicable to a wide range of networks. A high level parameter set is also used to decouple the information relevant to more than one slice from the media bitstream. H.264/AVC, for example, describes both a Sequence Parameter Set (SPS) and a Picture Parameter Set (PPS). The SPS applies to a whole sequence and the PPS applies to a whole frame. These describe parameters such as frame size, coding modes, and slice structuring. Further details on the structure of standards such as H.264 are provided in Chapter 12.

both a Sequence Parameter Set (SPS) and a Picture Parameter Set (PPS). The SPS applies to a whole sequence and the PPS applies to a whole frame. These describe parameters such as frame size, coding modes, and slice structuring. Further details on the structure of standards such as H.264 are provided in Chapter 12.

> Read full chapter

Software Engineering for Embedded and Real-Time Systems

Rob Oshana, in Software Engineering for Embedded Systems (Second Edition), 2019

10 Hardware Abstraction Layers for Embedded Systems

Embedded system development is about programming at the hardware level. But hardware abstraction layers (HALs) are a way to provide an interface between hardware and software so applications can be device independent. This is becoming more common in embedded systems. Basically, embedded applications access hardware through the HAL. The HAL encapsulates the peripherals of a microcontroller, and several API implementations can be provided at different levels of abstraction. An example HAL for an automotive application is shown in Fig. 19.

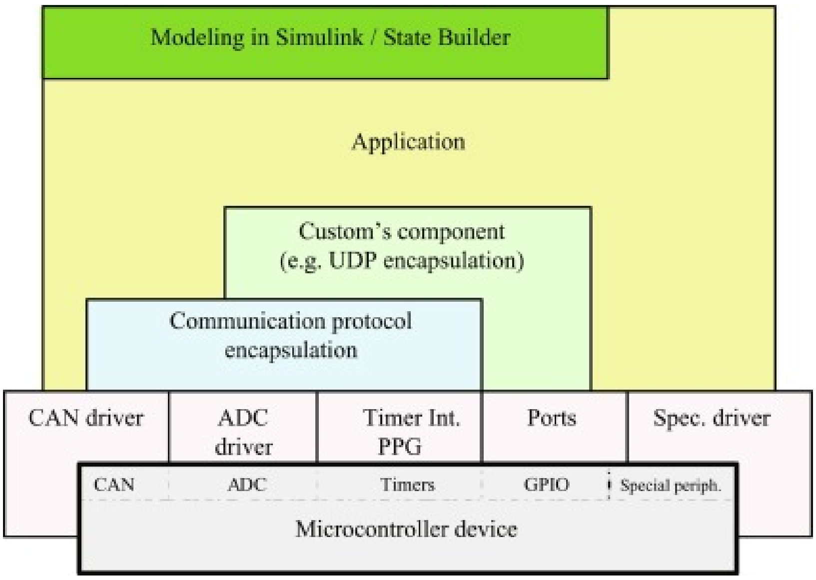


Fig. 19. Hardware abstraction layer.

There are a few problems that a HAL attempts to address:

- Complexity of peripherals and processors; this is hard for a real-time operating system (RTOS) to support out of the box, most RTOSs cover 20%–30% of the peripherals out of the box.
-

Packaging of the chip-making function—how does the RTOS work as you move from a standard device to a custom device?

- The RTOS is basically the lowest common denominator, a HAL can support the largest number of processors. However, some peripherals, like an analog-to-digital converter (ADC), require custom support (peripherals work in either DMA mode or direct mode, and we need to support both).

The benefits of a HAL include:

- Allowing easy migration between embedded processors.
- Leveraging existing processor knowledgebase.
- Creating code compliant with a defined programming interface, such as a standard application programming interface (API), a CAN driver source code, or an extension to a standard API, such as a higher level protocol over SCI communication (like a UDP) or even your own API.

As an example of this more advanced software architecture and a precursor to more detailed material to follow later, consider the case of an automobile “front light management” system as shown in Fig. 20. In this system, what happens if software components are running on different processors? Keep in mind that this automobile system must be a deterministic network environment. The CAN bus inside the car is not necessarily all the same CPU.

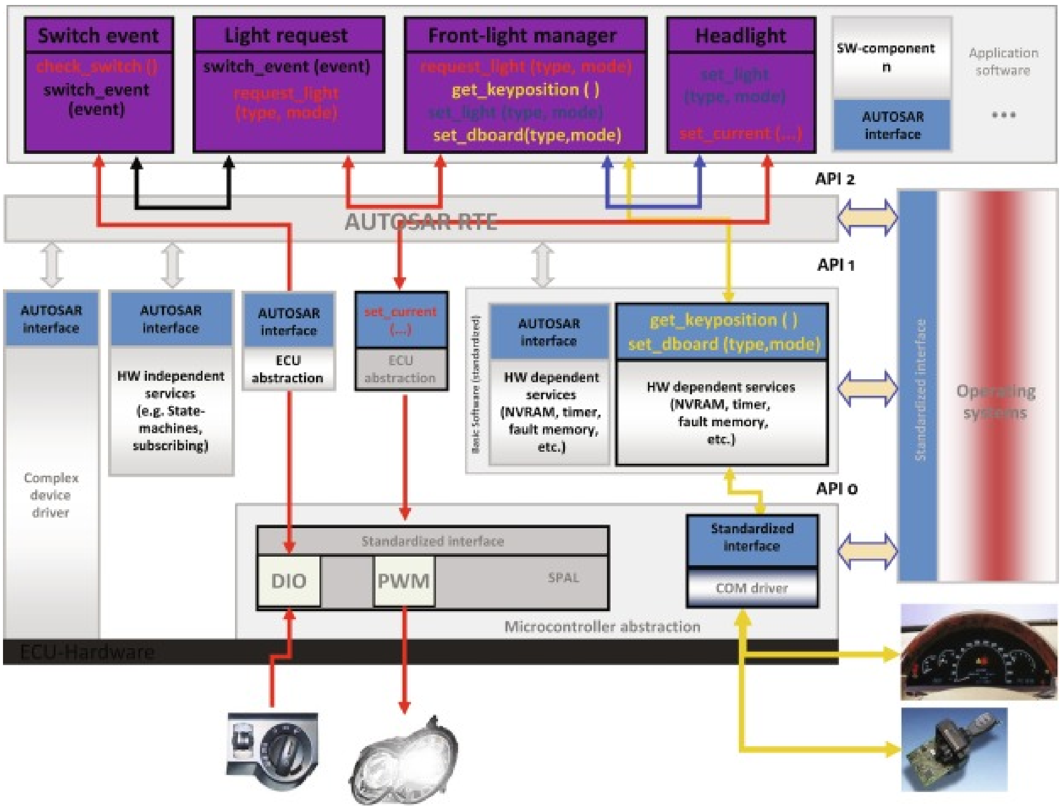


Fig. 20. Use case example of a “front light management” system.

Fig. 20. Use case example of a “front light management” system.

As shown in Fig. 21, we want to minimize the changes to the software architecture if we need to make a small change, like replacing a headlight. We want to be able to change the peripheral (changing the headlight or offering optional components as shown in Fig. 22) but not have to change anything else.

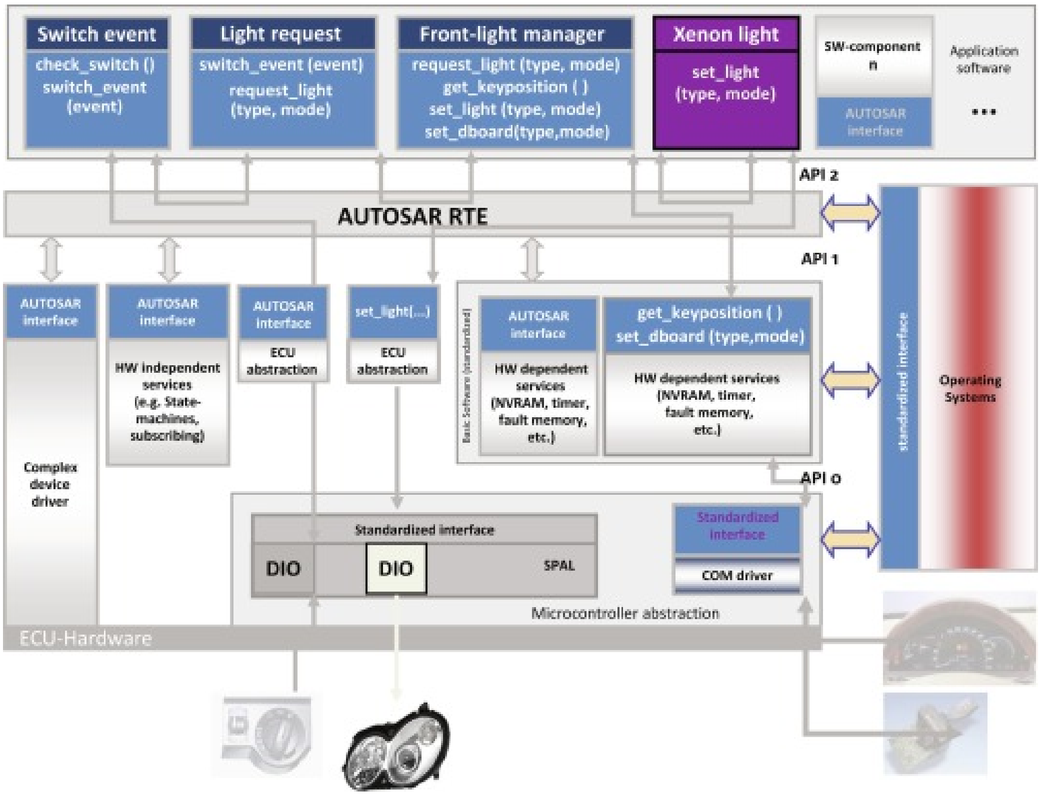
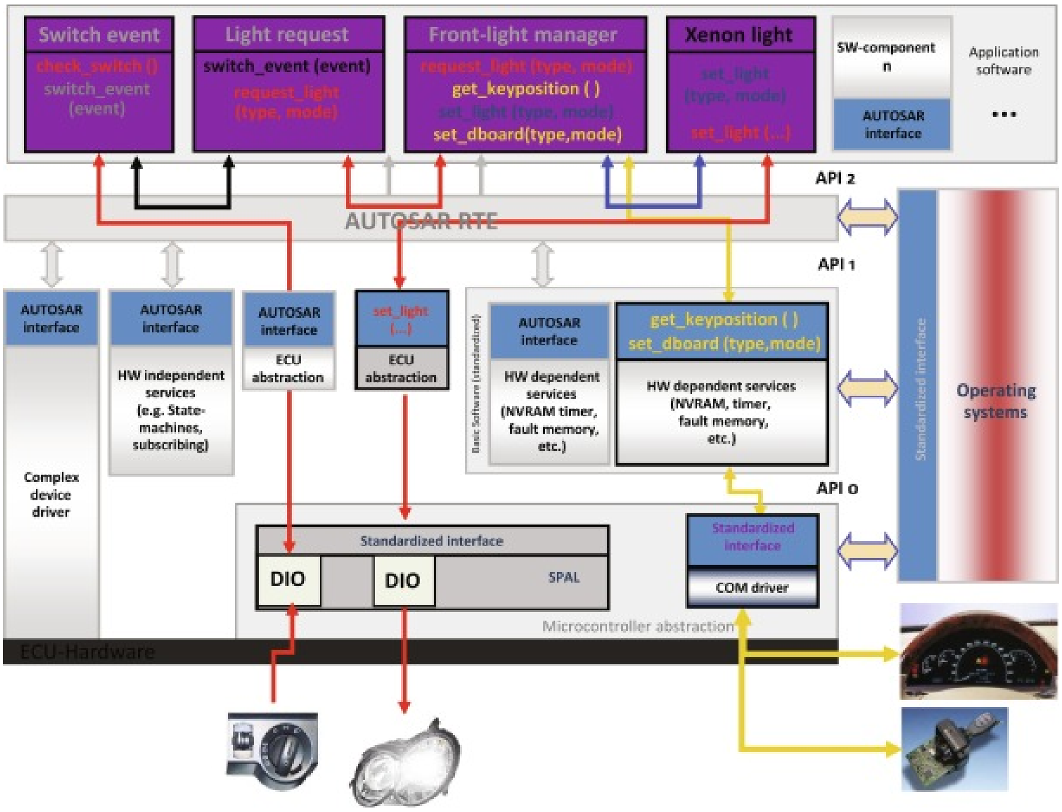


Fig. 21. “Front light management” system headlight components.



Switch event
check_switch ()
switch_event (event)
Light request
switch_event (event)
request_light (type, mode)
Front-light manager
request_light (type, mode)
get_keyposition ( )
set_light (type, mode)
set_dboard(type,mode)
Xenon light
set_light (type, mode)
set_light (...)
SW-component n
AUTOSAR interface
Application software
API 2
AUTOSAR RTE
API 1
AUTOSAR interface
Complex device driver
AUTOSAR interface
HW independent services (e.g. State-machines, subscribing)
AUTOSAR interface
ECU abstraction
set_light (...)
ECU abstraction
Basic Software (standardized)
AUTOSAR interface
HW dependent services (NVRAM timer, fault memory, etc.)
get_keyposition ( )
set_dboard (type,mode)
HW dependent services (NVRAM, timer, fault memory, etc.)
Standardized interface
Operating systems
API 0
Standardized interface
DIO
DIO
SPAL
Standardized interface
COM driver
Microcontroller abstraction
ECU-Hardware

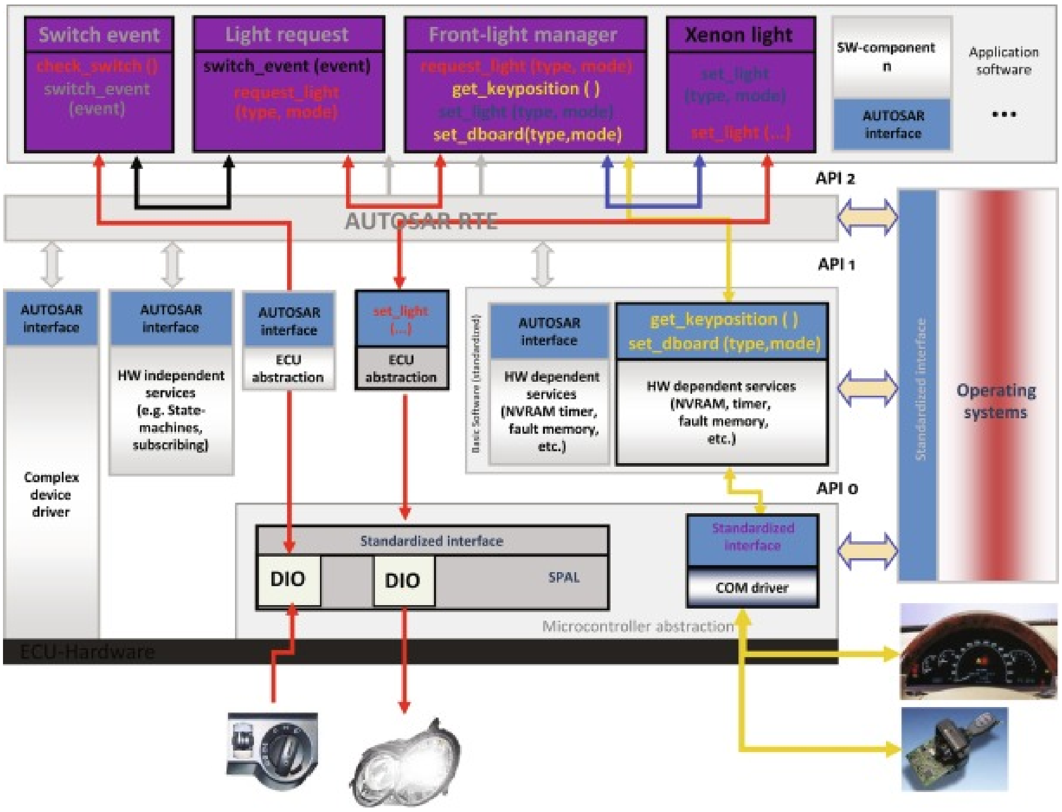


Fig. 22. "Front light management" system peripheral components.

Finally, the embedded systems development flow follows a model similar to that shown in Fig. 23. Research is performed early in the process followed by a proof of concept and a hardware and software codesign and test. System integration follows this phase, where all of the hardware and software components are integrated together. This leads to a prototype system that is iterated until eventually a production system is deployed. We will look into the details of this flow as we begin to dive deeper into the important phases of software engineering for embedded systems.

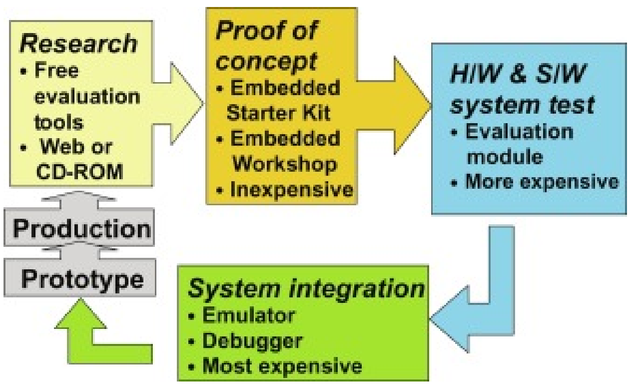


Fig. 23. Embedded system development flow.

> Read full chapter

Serial communications

Serial communications

B.R. Mehta, Y.J. Reddy, in Industrial Process Automation Systems, 2015

9.6.3 MODBUS and OSI model

9.6.3.1 OSI model

The Open Systems Interconnection (OSI) model is a product of the OSI effort at the International Organization for Standardization (ISO).

It entails characterizing and standardizing the functions of a communications system in terms of abstraction layers. Similar communication functions are grouped into logical layers. An instance of a layer provides services to its upper layer instances while receiving services from the layer below.

The MODBUS standard defines an application layer messaging protocol, positioned at level 7 of the OSI model that provides "client/server" communications between devices connected on different types of buses or networks. It standardizes a specific protocol on serial line to exchange MODBUS requests between a master and one or several slaves.

At the physical level, MODBUS over serial line systems may use different physical interfaces (RS485, RS232). TIA/EIA-485 (RS485) two-wire interface is the most common. As an add-on option, RS485 four-wire interface may also be implemented. A TIA/EIA-232-E (RS232) serial interface may be used as an interface, when only short point-to-point communication is required.

Figure 9.20 gives a general representation of MODBUS serial communication stack compared to the seven layers of the OSI model.

Layer	ISO/OSI model	Modbus layers
7	Application layer	Modbus application protocol
6	Presentation layer	Empty
5	Session layer	Empty
4	Transport layer	Empty
3	Network layer	Empty
2	Data link layer	MODBUS serial line protocol
1	Physical layer	EIA/TIA -485 (EIA/TIA -232)

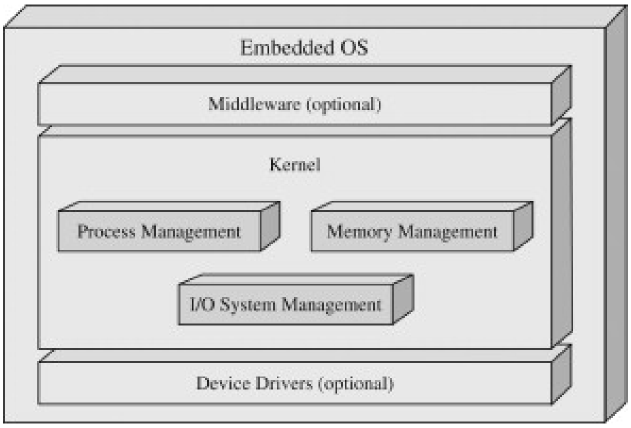
Figure 9.20. MOdbus OSI Layers

> Read full chapter

Software platforms for integrating robots and virtual environments

Angelo Basteris, Sara Contu, in Rehabilitation Robotics, 2018

helps new users to start to use Cortex-M microcontrollers and aids software portability.



Embedded OS
Middleware (optional)
Kernel
Process Management
Memory Management
I/O System Management
Device Drivers (optional)

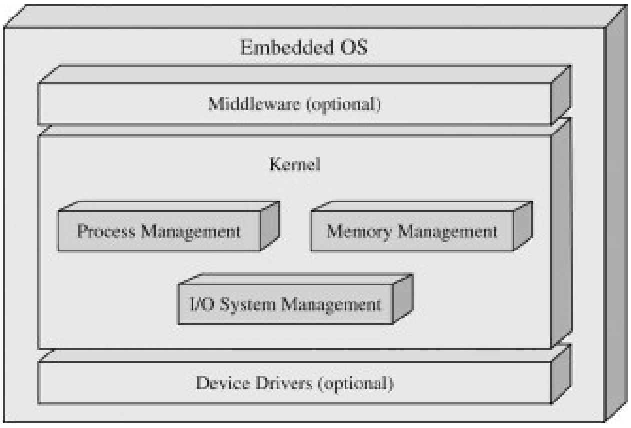


Figure 9-2a. General OS model.

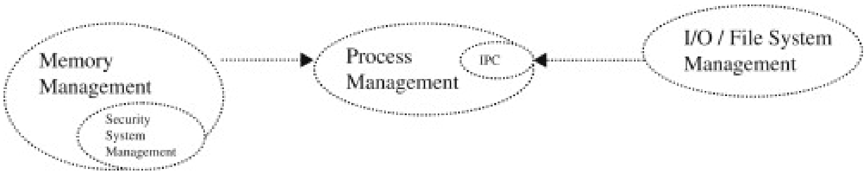


Figure 9-2b. Kernel subsystem dependencies.

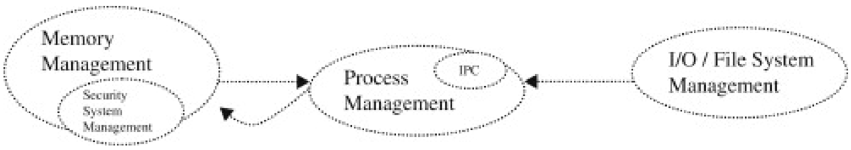


Figure 9-2c. Kernel subsystem dependencies.

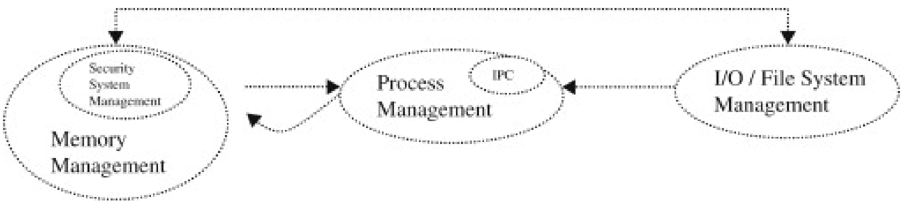


Figure 9-2d. Kernel subsystem dependencies.

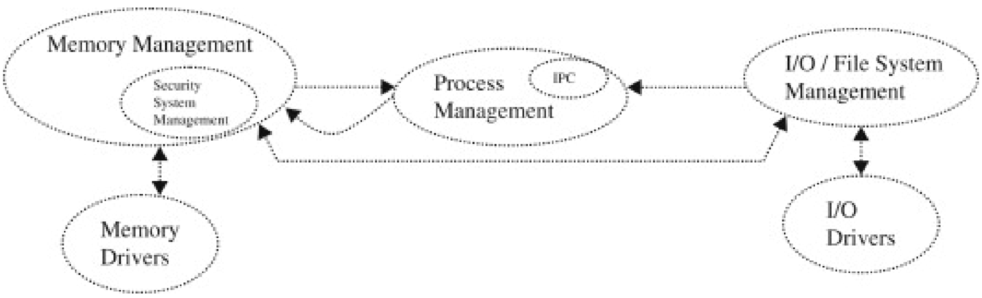


Figure 9-2e. Kernel subsystem dependencies.

- *Process Management*: how the OS manages and views other software in the embedded system (via *processes*—more in Section 9.2, Multitasking and Process Management). A subfunction typically found within process management is *interrupt and error detection management*. The multiple interrupts and/or traps generated by the various processes need to be managed efficiently, so that they are handled correctly and the processes that triggered them are properly tracked.

they are handled correctly and the processes that triggered them are properly tracked.

Memory Management: the embedded system's memory space is shared by all the different processes, so that access and allocation of portions of the memory space need to be managed (more in Section 9.3, Memory Management). Within memory management, other subfunctions such as *security system management* allow for portions of the embedded system sensitive to disruptions that can result in the disabling of the system, to remain secure from unfriendly, or badly written, higher-layer software.

I/O System Management: I/O devices also need to be shared among the various processes and so, just as with memory, access and allocation of an I/O device need to be managed (more in Section 9.4, I/O and File System Management). Through I/O system management, *file system management* can also be provided as a method of storing and managing data in the forms of files.

> Read full chapter