CHAPTER 1

Demystifying Middleware in Embedded Systems

Chapter Points

- Middleware is introduced in reference to the Embedded Systems Model
- Outline why understanding middleware is important
- Identifying common types of middleware in the embedded space

1.1 What is the Middleware of an Embedded System?

With the increase in the types and profitability of complex, distributed embedded systems, an approach common in the industry is designing and customizing these types of embedded systems in some manner that is independent of the underlying low-level system software and hardware components. To successfully achieve desired results within cost, schedule, and complexity goals many engineering teams base their approach on architecting various higher-level *middleware* software components into their embedded systems designs.

Currently within the embedded systems industry, there is no formal consensus on how embedded systems middleware should be defined. Thus, until such time as there is a consensus, this book takes the pragmatic approach of defining what middleware is and how different types of middleware can be categorized. Simply put, middleware is an abstraction layer that acts as an intermediary. Middleware manages interactions between application software and the underlying system software layers, such as the operating system and device driver layers. Middleware also can manage interactions between multiple applications residing within the embedded device, as well as applications residing across networked devices.

Middleware is simply software, like any other, that in combination with the embedded hardware and other types of embedded software is a means to an end to achieving some combination of the desirable goals shown in Table 1.1.

Demystifying Embedded Systems Middleware. DOI: 10.1016/B978-0-7506-8455-2.00001-7

Table 1.1: Examples of Desirable Requirements for Middleware to Meet

Requirement	Description
Adaptive	Middleware that enables overlying middleware and/or embedded applications to adapt to changing availability of system resources
Flexibility and Scalability	Middleware that allows overlying middleware and/or embedded applications to be configurable and customizable in terms of functionality that can be scaled in or out depending on application requirements, over all device requirements, and underlying system software and hardware limitations
Security	Middleware that insures the overlying middleware and/or embedded applications (and the users using them) have authorized access to resources
Portability	The 'write-once', 'run-anywhere' mantra. Middleware that allows overlying middleware and/or embedded applications to run on different types of embedded devices with different underlying system software and hardware layers. To avoid requiring time-consuming and expensive rewrites of the application code, middleware can mask the differences in underlying layers within different types of embedded systems, programming languages, and even implementations of the same standard produced by different design teams
Connectivity and Inter-Communication	Middleware that provides overlying middleware and/or embedded applications the ability to transparently communicate with other applications on a remote device through some user-friendly, standardized interface. Essentially, communication interfaces abstracted to level of local procedure call or method invocation

As shown in Figure 1.1a, middleware resides in the system software layer of an embedded system and is any software that is not a device driver, an operating system kernel, or an application. Middleware components can exist within various permutations of a real-world software stack: such as directly over device drivers, residing above an operating system, tightly coupled with an operating system package from an off-the-shelf vendor, residing above other middleware components, or some combination of the above, for example.

Keep in mind that what determines if a piece of software is 'middleware' is by where it resides within the embedded system's architecture, and not only because of its inherent purpose within the system alone. For example, as shown in Figure 1.1b, embedded Java virtual machines (JVMs) are currently implemented in an embedded system in one of three ways: in the hardware, in the system software layer, or in the application layer. When a JVM is implemented within the system software layer and resides on an operating system kernel is an example *when* a JVM is classified as *middleware*.

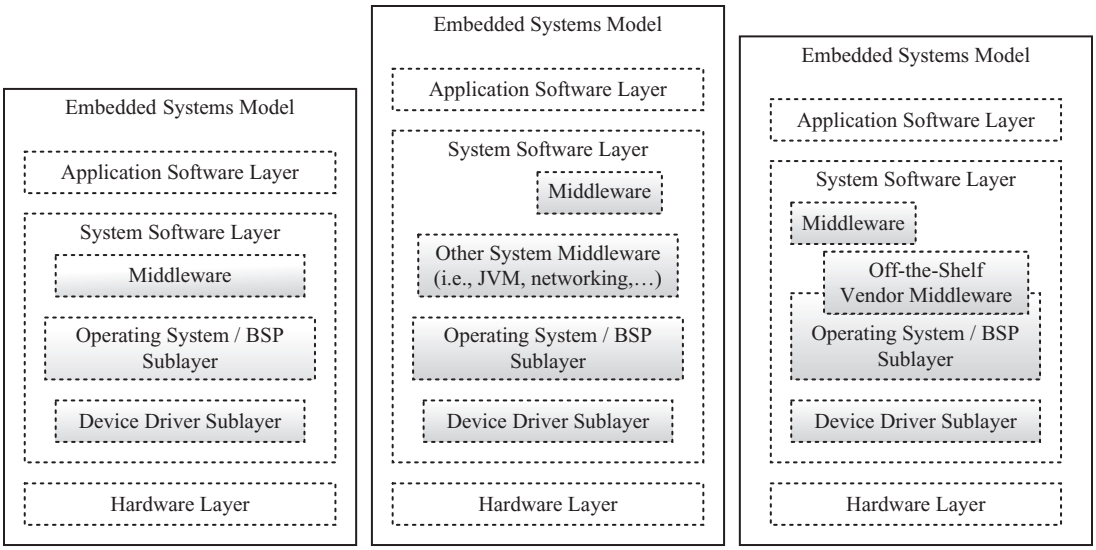

Figure 1.1a: Middleware and the Embedded Systems Model[1]

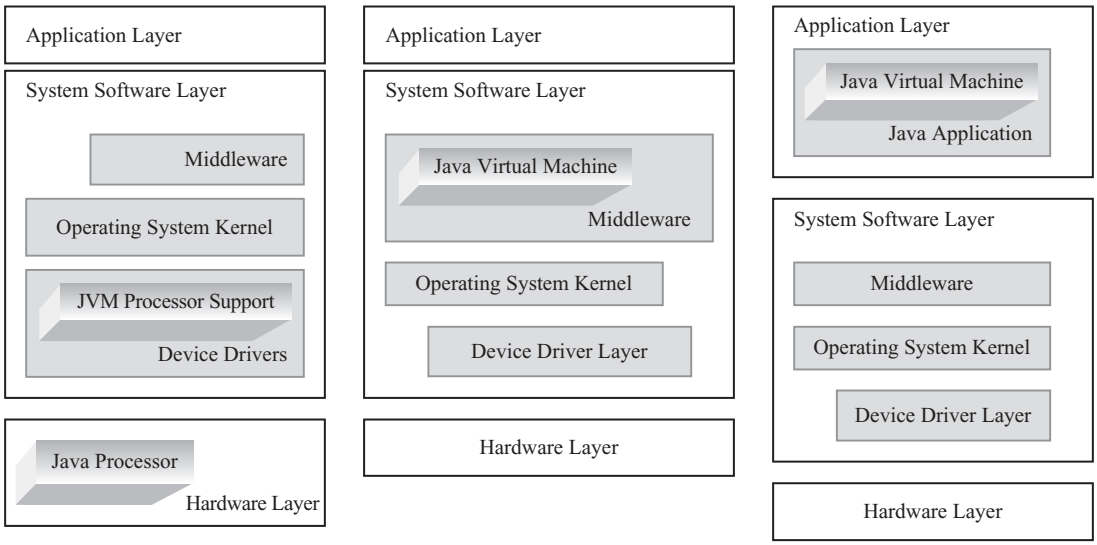

Figure 1.1b: Embedded JVMs in the Architecture[1]

Figure 1.1c shows a high-level block diagram of different types of middleware utilized in embedded devices today. Within the scope of this text, at the most general level, middleware is divided into two categories: ***core*** middleware and middleware that ***builds on*** these ***core*** components. Within each category, middleware can be further broken down into types, such as file systems, networking middleware, databases, and virtual machines to name a few. Open source and

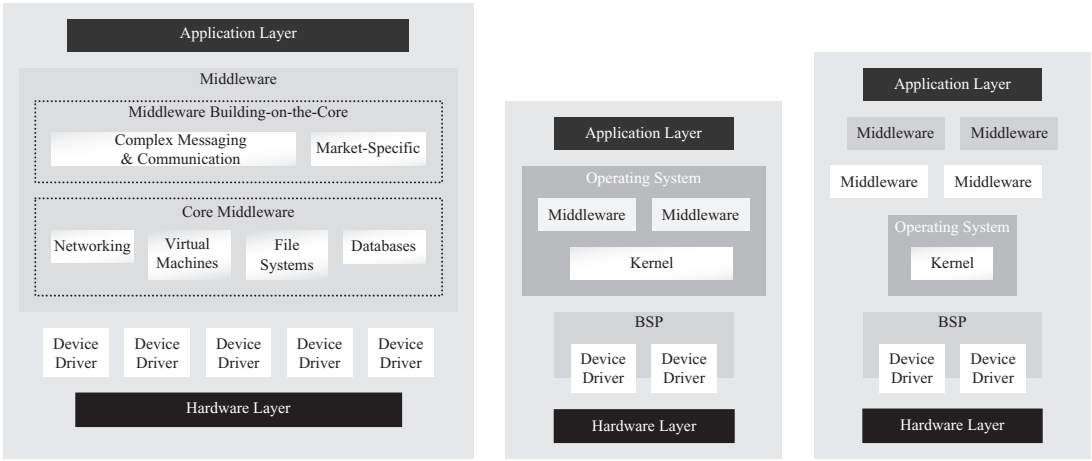

Figure 1.1c: Types of Middleware in Embedded Systems

real-world examples of these types of middleware will be used when possible throughout this book to demonstrate the technical concepts. Examples of building real-world designs based on these types of middleware will be provided, and the challenges and risks to be aware of when utilizing middleware in embedded systems will also be addressed in this text.

Core middleware is software that is most commonly found in embedded systems designs today that do incorporate a middleware layer, and is the type of software that is most commonly used as the foundation for more complex middleware software. By understanding the different types of core middleware, the reader will have a strong foundation to understanding and designing any middleware component successfully. The four types of core middleware discussed in this book are:

- **Chapter 4**. Networking
- **Chapter 5**. File systems
- **Chapter 6**. Virtual machines
- **Chapter 7**. Databases.

Middleware that builds on the core components varies widely from market to market and device to device. In general, this more complex type of middleware falls under some combination of the following:

- Message Oriented and Distributed Messaging, i.e.,
  - Message Oriented Middleware (MOM)
  - Message Queues
  - Java Messaging Service (JMS)
  - Message Brokers
  - Simple Object Access Protocol (SOAP)

- Distributed Transaction, i.e.,
  - Remote Procedure Call (RPC)
  - Remote Method Invocation (RMI)
  - Distributed Component Object Model (DCOM)
  - Distributed Computing Environment (DCE)
- Transaction Processing, i.e.,
  - Java Beans (TP) Monitor
- Object Request Brokers, i.e.,
  - Common Object Request Broker Object (CORBA)
  - Data Access Object (DAO) Frameworks
- Authentication and Security, i.e.,
  - Java Authentication and Authorization Support (JAAS)
- Integration Brokers.

At the highest level, these more complex types of middleware will be subcategorized and discussed under the following two chapters:

- **Chapter 3**. Market-specific Complex Middleware
- **Chapter 8**. Complex Messaging and Communication Middleware.

This book introduces the main concepts of different types of middleware and provides snapshots of open-source to help illustrate the main points. When introducing the fundamentals of various middleware components within the relative chapters, this book takes a multistep approach that includes:

- discussing the importance of understanding the standards, underlying hardware, and system software layers
- defining the purpose of the particular middleware component within the system, and examples of the APIs provided with a particular middleware component
- introducing middleware models and open-source software examples that would make understanding the middleware software architecture much simpler
- providing some examples of how overlying layers utilize various middleware components to apply some of what the reader has read.

The final chapter pulls it all together with pros and cons of utilizing the different types of middleware in embedded systems designs. As this book will demonstrate, there are several different types of embedded systems middleware on the market today, in addition to the countless homegrown solutions. Note that these embedded systems middleware solutions can be further categorized as other types of middleware depending on the field – such as being *proprietary* versus *open-source*, for example. In short, the key is for the reader to pick up on the high-level concepts and the patterns in embedded middleware software – and to recognize that these endless permutations of middleware solutions in the embedded space exist, because there is not 'one' solution that is perfect for all types of embedded designs.

1.2 How to Begin When Building a Complex Middleware-based Solution

For better or worse, successfully building an embedded system with middleware requires more than just solid technology alone. Engineers and programmers who recognize this wisdom from day one are most likely to reach production within quality standards, deadlines, and costs. In fact, the most common mistakes that kill complex embedded systems projects, especially those that utilize middleware components, are unrelated to the middleware technology itself. It is because team members did not recognize that successfully completing complex embedded designs requires:

- **Rule #1**: more than technology
- **Rule #2**: discipline in following development processes and best practices
- **Rule #3**: teamwork
- **Rule #4**: alignment behind leadership
- **Rule #5**: strong ethics and integrity among each and every team member.

So, what does this book mean by Rule 1 – that building an embedded system with middleware successfully requires more than just technology?

It means that many different influences, including technical, business-oriented, political, and social to name a few, will impact the process of architecting an embedded design and taking it to production. The architecture business cycle shown in Figure 1.2 shows a visualization

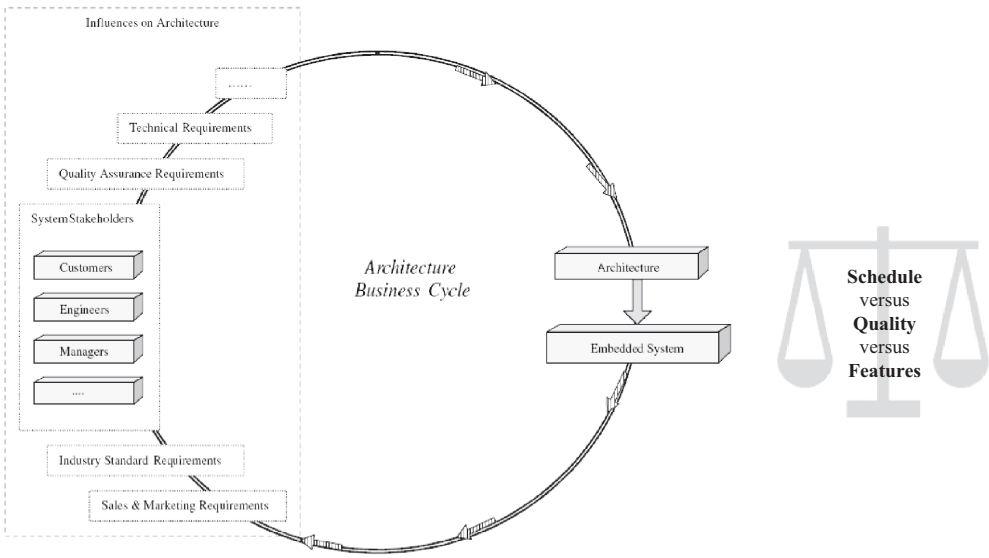

Figure 1.2: Architecture Business Cycle[2]

of this rule in which many different types of influences generate the requirements, the requirements in turn generate the embedded system's architecture, this architecture is then the basis for producing the device, and the resulting embedded system design in turn provides feedback for requirements and capabilities back to the team.

So, out of the architecture business cycle comes a reflection of what challenges real-world development teams building a complex middleware-based system face – balancing quality versus schedule versus features. This is where the other four rules stated at the start of this section come into play for insuring success. Ultimately, the options embedded teams have to choose from when targeting to successfully build a complex design are typically some combination of:

- **X** Option 1: Don't ship
- **X** Option 2: Blindly ship on time, with buggy features
- **X** Option 3: Pressure tired developers to work even longer hours
- **X** Option 4: Throw more resources at the project
- **X** Option 5: Let the schedule slip
- √ Option 6: Healthy Shipping Philosophy: '**Shipping a very *high-quality* system *on time*.**'

Not shipping unfortunately happens too often in the industry, and is obviously the option everyone on the team wants to avoid. 'No' products will ultimately lead to 'no' team, and in some cases 'no' company. So, moving on to the next option – why 'shipping a buggy product' is also to be avoided at all costs is because there are serious liabilities that would result if the organization is sued for a lot of money, and/or employees going to prison if anyone gets hurt as a result of the bugs in the deployed design (see Figure 1.3). When developers are forced to cut corners to meet the schedule relative to design options, are being forced to work overtime to the point of exhaustion, are undisciplined about using best practices when programming, code inspections, testing, and so on – this can then result in serious liabilities for the organization when what is deployed contains serious defects.

Option 3 – 'pressure tired developers to work even longer hours' – is also to be avoided. The key is to 'not' panic. Removing calm from an engineering team and pushing exhausted developers to work even longer overtime hours on a complex system that incorporates middleware software will only result in more serious problems. Tired, afraid, and/or stressed-out engineers and developers will result in mistakes being made during development, which in turn translates to additional costs and delays.

Negative influences on a project, whether financial, political, technical, and/or social in nature, have the unfortunate ability to negatively harm the cohesiveness of an ordinarily healthy team within a company – eventually leading to sustaining these stressed software teams as unprofitable in themselves. Within a team, even a single weak link, such as a team of exhausted and stressed-out engineers, will be debilitating for an entire project and even an

- **Breach of Contract**, i.e.,
  - if bug fixes stated in contract are not forthcoming in timely manner
- **Breach of Warranty and Implied Warranty**, i.e.,
  - delivering system without promised features
- **Strict and Negligence Liability**, i.e.,
  - bug causes damage to property
  - bug causes injury
  - bug causes death
- **Malpractice**, i.e.,
  - customer purchases defective product
- **Misrepresentation and Fraud**, i.e.,
  - product released and sold that doesn't meet advertised claims

- Based on the chapter "Legal Consequences of Defective Software" by Cem Kaner Testing Computer Software. 1999

Figure 1.3: Why Not Blindly Ship? – Programming and Engineering Ethics Matter[3]

entire organization. This is because these types of problems radiate outwards influencing the entire environment, like waves (Figure 1.4).

The key here is to decrease the interruptions (see Figure 1.5) *and* stress for a development team during their most productive programming hours within a **normal** work week, so that there is more focus and fewer mistakes.

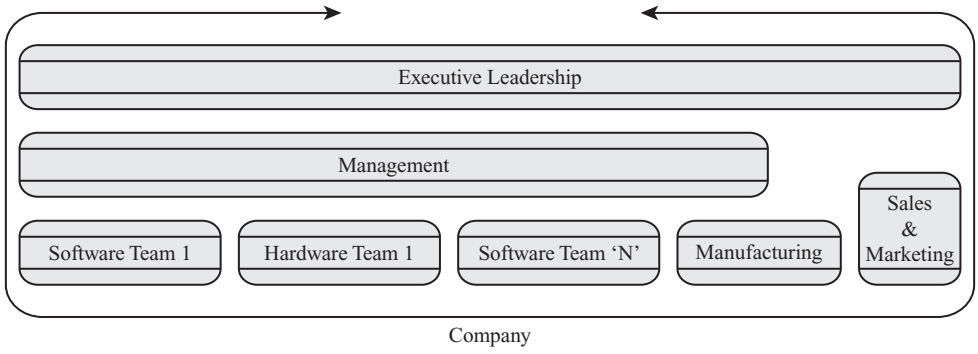

Figure 1.4: Problems Radiate and Impact Environment

"… developers imprisoned in noisy cubicles, those who had no defense against frequent interruptions, did poorly. How poorly? The numbers are breathtaking. The best quartile was 300% more productive than the lowest 25%. Yet privacy was the only difference between the groups.

Think about it – would you like 3× faster development?

It takes your developers 15 minutes, on average, to move from active perception of the office busyness to being totally and productively engaged in the cyberworld of coding. Yet a mere 11 minutes passes between interruptions for the average developer. Ever wonder why firmware costs so much? …"

- Jack Ganssle. *A Boss's Quick-Start to Firmware Engineering*.
- DeMarco and Lister. *Peopleware*.

Figure 1.5: Real World Tidbit, Underpinnings of Software Productivity

Another approach in the industry to avoid a schedule from slipping has been to throw more and more resources at a project. Throwing more resources ad-hoc at project tasks without proper planning, training, and team building is the surest way to hurt a team and guarantee a missed deadline. As indicated in Figure 1.6, productivity crashes with the more people there are on a project. A limit in the number of communication channels can happen through more than one (>1) smaller sub-teams, as long as:

- it makes sense for the embedded systems product being designed, i.e.,
  - not dozens of developers and several line/project managers for a few MB of code
  - not when few have embedded systems experience and/or experience building the product
  - not for corporate empire-building! – which results in costly project problems and delays = bad for business!

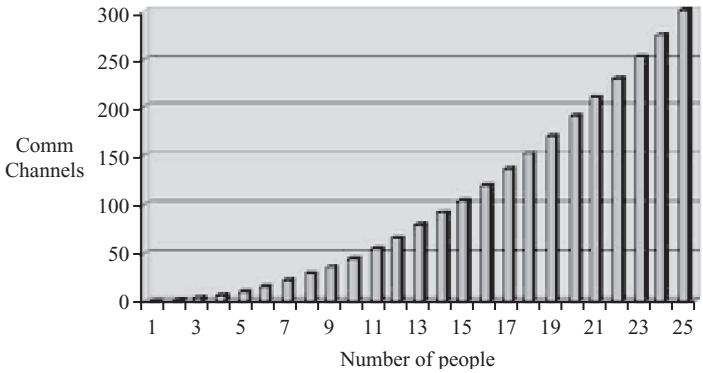

Figure 1.6: Too Many People[4]

- in a healthy team environment
- no secretiveness
- no hackers
- best practices and processes not ignored
- team members have sense of professional responsibility, alignment, and trust with each other, leadership and the organization.

While more related to this discussion will be covered in the last chapter of this book, ultimately the most powerful way to meet project schedules and successfully take an embedded system middleware-based solution to production is:

- by shipping a very high-quality product on time
- have a strong technical foundation
- sacrificing less essential features in the first release
- start with skeleton, then hang code off skeleton
- Do not overcomplicate the design!
- Systems integration, testing and verification from Day 1.

The rest of this chapter and most of this book are dedicated to supplying the reader with a strong, pragmatic technical foundation relative to embedded systems middleware. The last section of this book will pull it all together to link in what was introduced in this section.

1.3 Why is a Strong Technical Foundation Important in Middleware Design?

One of the biggest myths propagated by inexperienced team members and mistakes made in the industry is assuming that the embedded systems programmers of a middleware layer can afford to think as abstractly as PC developers and/or the application developers using that middleware layer. There are too many examples of stressed-out engineers, millions of dollars in project overruns, and failed ventures in the industry that are a result of team members not understanding the fundamentals relative to utilizing middleware within an embedded system at the start and throughout the design process of the project. When it comes to understanding the underlying hardware and system software when designing middleware software, it is critical that, at the very least, developers understand the entire design at a systems level. In fact, one of the most common mistakes made on an embedded project that makes it much tougher to successfully build a complex design is when engineers and programmers on the team do not investigate or understand the type of embedded system they are trying to build, the components that can make up the device, and/or the impact individual components have on each other.

Thus, this book is a springboard from *'Embedded Systems Architecture: A Practical Guide for Engineers and Programmers'*. This book takes a more detailed and practical

approach of discussing all layers relative to the Embedded Systems Model, shown in Figure 1.1a, when introducing principles and major elements of embedded systems middleware. This is because it is critical to the success of any project team that introduces middleware into the architecture that all team members understand all layers of an embedded system because ***all*** layers of an embedded system are ***impacted*** by middleware and vice versa.

Introducing middleware software to an embedded system introduces an **additional overhead** that will impact everything from memory requirements to performance, reliability, as well as scalability, for instance. The goal of this book is not just about introducing some of the most common types of embedded systems middleware, but **more importantly** *to show the reader the pattern behind different types of embedded middleware designs and to help teach the reader an approach to understanding and applying this knowledge to any embedded system's middleware component encountered in the future*.

The Embedded Systems Model represents the layers in which all components existing within an embedded system design can reside. This model is a powerful tool utilized within the scope of this book because it not only provides a clear visual representation of the various middleware elements of an embedded system, their interrelationships, and functionality – this model also provides a basis for modular architectural representations that commonly are used to successfully structure an embedded systems project. At the highest level, there are three layers:

- **hardware**, which contains all the physical components located on an embedded systems board
- **system software**, which is the device's application-independent software
- **application software**, which is the device's application-specific software.

As shown in Figure 1.7, a middleware component – whether it is a file system, database, or networking protocol – that resides in an embedded system's middleware software layer typically resides on top of 'some' combination of other middleware, an operating system, device drivers, and hardware. This means middleware implemented in the system software layer exists either as:

- middleware that sits on top of the operating system layer, or device driver layer for systems with no operating system
- middleware that sits on top of other middleware components, for example a Java-based database or file system that resides over a Java Virtual Machine (JVM)
- middleware that has been tightly integrated and provided with a particular operating system distribution.

In some embedded systems, there may even be more than one different middleware component, as well as more than one of the same type of middleware in the embedded device (see Figure 1.8). In short, whatever the combination of middleware – in co-operation with

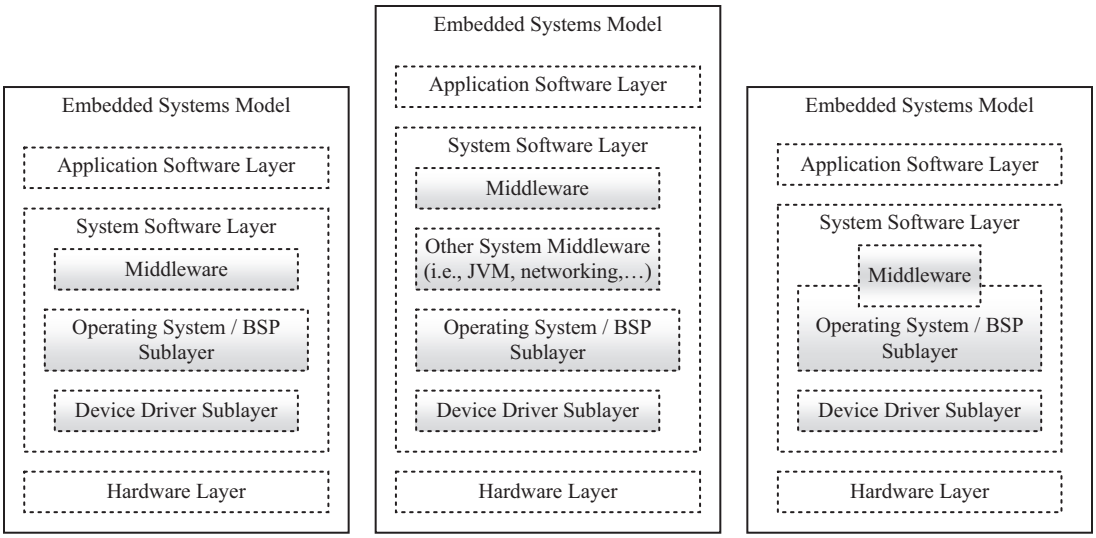

Figure 1.7: System Components and the Embedded Systems Model

the underlying embedded software and hardware – these components act as an abstraction layer that provides various data management functions to the other system software layer components, application software layer in the system, and even other computer systems that have remote access to the device.

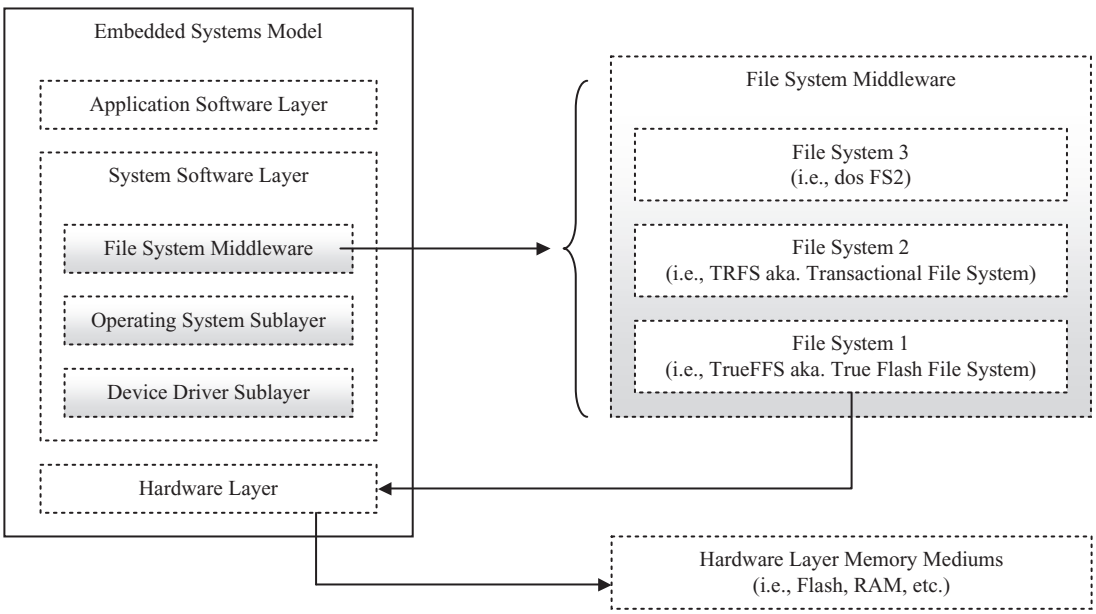

Figure 1.8: Multiple File Systems in an Embedded System Example

1.4 Summary

Middleware is increasingly becoming a required component in embedded systems designs due to the increase in the types of complex, distributed embedded systems, the number of applications found on embedded systems, and the desire for customizable embedded software applications for embedded devices. In this chapter, middleware was defined relative to the Embedded Systems Model, and the types of middleware introduced in this book were also discussed. Finally, some initial guidelines of whether using middleware within an embedded systems design should even be entertained as an option are discussed.

Chapters 4–7 cover core middleware components, specifically file systems, networking, and databases. Chapters 3, 8 and 9 go on to discuss middleware that builds on the core components, as well as pulls all the concepts together in discussing overall design implementations, approaches, and risk mitigation for utilizing middleware in real-world embedded designs.

The next chapter of this book introduces core components that underlie middleware commonly found in embedded systems. Chapter 2, specifically, introduces the hardware and underlying system software required by core middleware.

1.5 End Notes

1 Systems Architecture, Noergaard, 2005. Elsevier.
2 The six stages of creating an architecture outlined and applied to embedded systems in this book are inspired by the Architecture Business Cycle developed by SEI. For more on this brainchild of SEI, read 'Software Architecture in Practice,' by Bass, Clements, and Kazman.
3 Based on the chapter 'Legal Consequences of Defective Software' by Cem Kaner. Testing Computer Software. 1999.
4 'Better Firmware, Faster'. Jack Ganssle. 2007.