White Paper

Embedded Infrastructure – A Re-use Opportunity

by Anders Rosvall and Jan-Erik Frey

Embedded Artists

Embedded Artists AB

Västerås Technology Park
Glödgargränd 14
SE-721 30 Västerås
Sweden

Phone/Fax +46 (21) 470 22 00

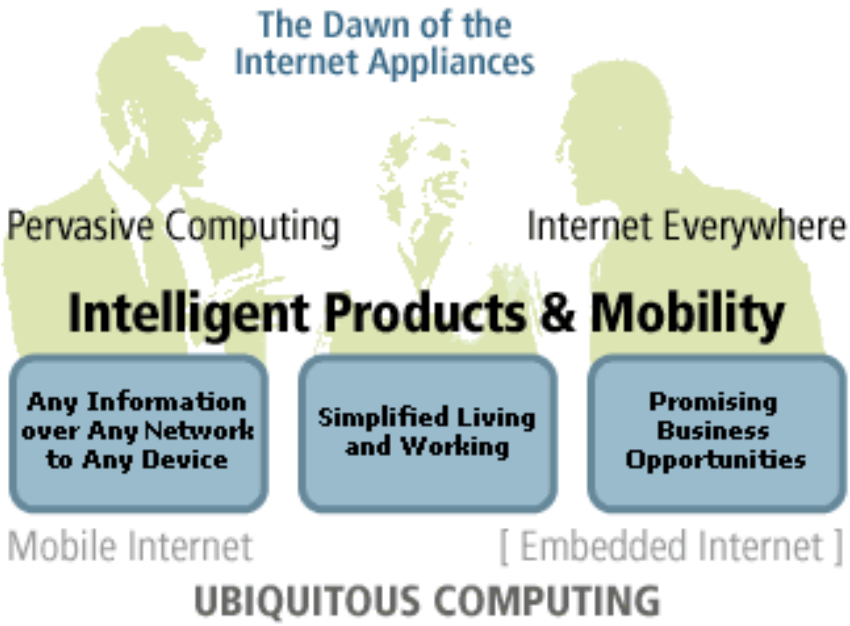
info@EmbeddedArtists.com
http://www.EmbeddedArtists.com

Table of Contents

Abstract

Embedded Internet, Mobile Internet, Internet Everywhere, Ubiquitous Computing, the Dawn of the Internet Appliances… there is no shortage of aphorisms to describe the current boost of embedded applications. While these trends create promising business opportunities, they also create a feature explosion that most embedded product suppliers have difficulties managing. Recent surveys show that 80% of all embedded systems are delivered late. However, a closer analysis shows that the majority of this increase in functionality is contributed to the software infrastructure of the system rather than the applications. The software infrastructure is comprised of functionality common to a large group of application domains; typically 2/3 of the development is devoted to the infrastructure. While the functionality is common, the requirements differ greatly. This has led to a very low level of standardization and reuse in resource-constraint embedded systems.

This white paper explores the field of embedded software infrastructure, provides a number of application analyses, and presents a method to increase the level of reuse and decrease the time spent to get the basic functionality up and running in a new project. Embedded Artists' graphical configuration tool, **InfraBed™**, is presented and put in context with regard to infrastructure software development.



Note: This white paper address infrastructure in embedded systems in a general format, except in *Chapter 4 – The InfraBed™ Concept*. This chapter presents Embedded Artists' solution for embedded software development. Readers not interested in this information can simply skip over *Chapter 4*.

1 Introduction

1.1 Feature Explosion

Embedded internet, mobile internet, internet everywhere, ubiquitous computing, the dawn of the internet appliances… there is no shortage of aphorisms to describe the current boost of embedded applications. While these trends create promising business opportunities, they also create a feature explosion that most embedded product suppliers have difficulties managing. According to Embedded Systems Programming magazine [February 1999, page 98], 80% of all embedded projects are delivered late.

Many of today's embedded systems exhibit behavior normally associated with yesterday's PC-based systems. *Figure 1* below illustrates the evolution of embedded systems during the last decades. The increased demands on functionality along with the ever-decreasing time-to-market requirements make embedded system development a demanding and difficult task.

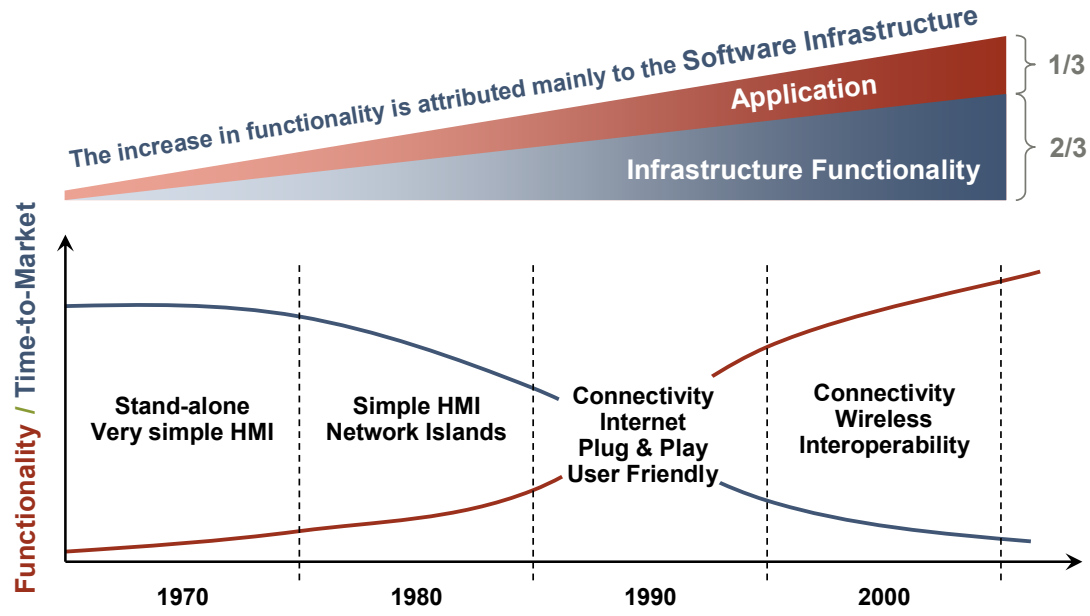


Figure 1: Evolution of Embedded Systems

Now, the figure above would suggest that embedded applications have become increasingly complex over the last decades. However, a closer analysis shows that the bulk of this increase in functionality and complexity is related to infrastructure functionality rather than the actual applications. The software infrastructure is comprised of functionality common to a large group of application domains; typically 2/3 of the software development is devoted to the infrastructure.

While the functionality is common, the requirements on the infrastructure differs greatly (performance, memory consumption, robustness, interfaces, etc.). This has lead to a very low level of standardization and reuse in resource-constraint embedded systems.

1.2 Nothing has Changed, Everything is New

In the same manner as a car hasn't really changed since the emergence of the T-Ford, many embedded applications perform the same tasks today as 5-10 years ago. Today, a large (and rapidly increasing) time of the software development of embedded systems is spent on the development of basic functions and on realizing the selected system architecture. This part of the system is referred to as the **software infrastructure** (or simply infrastructure in this

paper). For example, an application might have used a simple serial interface to communicate with its surroundings. The wireless trend now "forces" this application to accomplish its communication via a wireless link. Although this might have beneficial effects from a business point of view, the complexity of the system has increased by a multitude (while the application might perform the exact same task as before).

To maintain their market shares, embedded product manufacturers struggle to comply with all these new standards and trends. What was high-tech and exiting yesterday is mandatory today and not even considered as added-value tomorrow. This "compliance race" adds even more requirements/demands on the infrastructure. To illustrate this technology spin, let's twirl down the wireless trail for a while.

Wireless communication creates many promising business opportunities, but wireless communication also affects the total system architecture. At some point, the wireless system has to be integrated with the existing network/infrastructure. This requires protocol conversions, e.g., Fieldbus or IP communication over different media, and results in a variety of communication bridges. Wireless communication also affects how systems are used. Generally speaking we are moving from a centralized to a distributed paradigm, which creates an increased need for local data storage (databases, file systems, and web servers). Further, wireless communication has an unreliable and open medium. If sensitive information is exchanged, encryption and authentication functions need to be provided. Local buffering of data is a must to avoid loss of data.

The list can be made much longer, but the point is that wireless systems require a large set of infrastructure functions (other than pure communication protocols such as Bluetooth).

1.2.1 Mobile Phone Address Book Examined

To further illustrate the importance and complexity of the infrastructure in modern embedded systems, let's take a look at a very common embedded system, namely our mobile phones. A central application of a mobile phone is the address book functionality. To implement an address book, a large set of infrastructure functions is required: Flash File System (to persist data), Database (to accomplish structured storage and retrieval), Sorting (to organize views), Search (to locate entries), and Graphical User Interface (user interaction). In fact, the application *Address Book* is nothing other than the integration of these functions, the design of the user interface and the connection to the target hardware (connecting the buttons of your phone to the correct functions). *Figure 2* below illustrates the Address Book example.

Mobile Phone Address Book Function

Infrastructure functions
- FLASH File System (*to persist data*)
- Database (*to accomplish structured storage and retrieval*)
- Sorting (*to organize views*)
- Search (*to locate entries*)
- Graphical User Interface (*to allow user interaction*)

Application
- Integration of functions above
- Design of graphical user interface
- Tying keystrokes to appropriate actions
- Hardware interface (to FLASH, LCD driver, and keypad)

Figure 2: Mobile Phone Address Book Example

1.3 Keep it Simple

Technology and market trends aside, to really be able to compete, cost can never be compromised. So 8- and 16-bit microcontrollers and Internet services does not rime to well, or does it? As we are forced to put more and more functionality into our embedded devices, the requirements on the infrastructure have become quite rigorous. If we are to be able to fulfill the vision of Internet appliances, the infrastructure needs to be highly optimized to minimize the footprint and CPU load. So, if a careful selection/implementation of protocols and applications is made, it is definitely feasible to run a web server on an 8-bit controller with less than 32 Kbytes. There are even examples of 4-8 Kbytes implementations. The smaller the footprint, the more specialized the implementation must be. To pre-compute protocol frames (at compile time) is one example to minimize protocol processing overhead and code size. Limiting generality, given by application specifics, is another commonly used technique to reduce the footprint and processing overhead.

No discussion around embedded Internet working can be left without questioning the future of 8- and 16-bit processors. There are different views on how big the different microcontroller market segments will be in the future. According to industry analysts, sales of embedded 8-bit microcontrollers are estimated to more than double sales of 16- and 32-bit embedded general-purpose microprocessors. Manufacturers in the 8-bit segment strongly believe there will be a future for this segment. Companies producing software for the upper-end of the embedded systems spectrum (for example PDAs) believe in the 32-bit future. This is logical since their products require this type of computational power.

In general, 8/16-bit systems will always be cheaper than 32-bit systems; because of the cost of more external memory chips. Today, few 32-bit MCUs can offer all necessary memory on-chip (this may however change in the future). Also, the program code size is generally larger (typically true for RISC processors, which is why 16-bit instruction subsets have been developed for some of the processor families). The cost of the MCU itself is also higher for high-end processors, primarily due to the inclusion of an on-chip cache and more advanced peripherals.

So if you ask a white goods manufacturer (who typically struggles to save a fraction of a penny per refrigerator or stove), adding a dollar or two is simply not acceptable. Regardless of which segment will win, the fact remains that the leaner the infrastructure, the simpler the MCU, the lower the cost, the more competitive you are.

2 Embedded Software Infrastructure

In the previous chapter we argued about the importance of the infrastructure. Now let's take a closer look at what the infrastructure actually is. We begin by positioning the infrastructure from a systems hierarchy point of view. Second we categorize and exemplify typical infrastructure functionality. Finally we provide an application analysis to help obtain a better understanding of which parts of your own system that can be considered as the software infrastructure.

2.1 Infrastructure Defined

There are many definitions of what a software infrastructure is. Common for most of them is: *"A set of basic functions that are common between applications"*. Much like the roads, railroads, airline systems, power networks, and telecommunication systems are the infrastructure of our society; the software infrastructure provides basic services to an application program. These basic functions present a framework upon which applications can be built, enabling the application program to focus on the core product functionality and user-perceived specifics.

Much of the functions that are implemented in the infrastructure are taken for granted by a typical user. A few examples are: non-volatile storage of product settings, a graphical user interface, responsiveness, and communication with external systems. Most products are soon expected to have Internet and/or Bluetooth connectivity (to be a bit general…), and as argued earlier, the current industry trends often imply a further increase in infrastructure functionality.

Today, infrastructure functionality is largely implemented by the embedded product suppliers themselves, with a few exceptions such as TCP/IP protocol stacks and web servers. Also, infrastructure functionality is often tightly coupled to the application, often due to pressed time schedules that do not permit a proper design where application and infrastructure are clearly separated. This monolithic approach, although often a necessity, greatly hinders software re-use to any larger extent. Why is that?

In *Figure 3* below we take a look at an embedded system from a systems hierarchy point of view. If we break down the software portion of the system, we find that the middle layer of the system (the infrastructure) represents the only part of the system that is somewhat generic (and therefore re-usable).

It's obvious that the hardware and application program are very specific to each product. However, the layer between these parts is rather general and the requirements are more or less the same for a given application domain. In fact, the infrastructure is quite generic also between completely different applications, which will be shown later.

In short, the functionality is *common to a large group of application domains*, but the *requirements differ greatly* in: behavior, memory, speed, interfaces.

As a conclusion; ***One size does NOT fit all applications!***

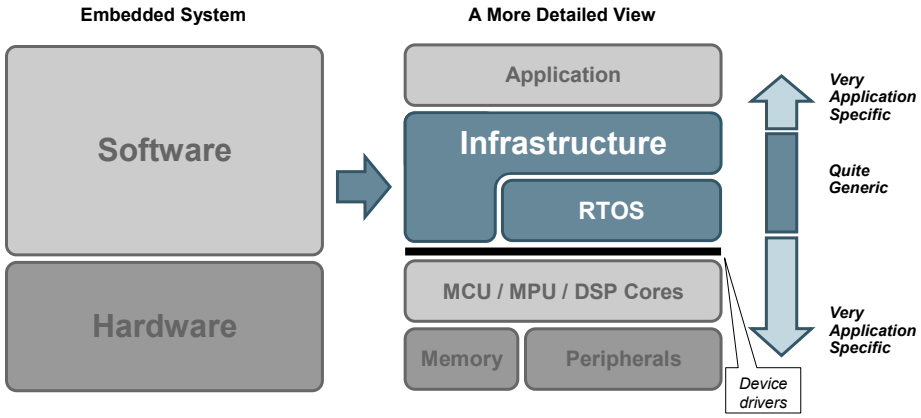


Figure 3: Embedded Systems Hierarchy & Infrastructure

It's worth noting that product differentiation takes place in the upper and lower system layers. The application layer contains the user-perceived functionality of the product. It's mainly in this part of the program that the marketing department has opinions, and it's here a product acquires its unique identity. Optimizing the hardware is very essential for a manufacturer since the per-unit production cost must be kept as low as possible. This sometimes leads to very specific and inflexible designs. Adding specific interfaces (for example an extra serial port) can often be a marketing decision rather than a technical requirement.

The infrastructure constitutes the majority of the development effort today, sometimes as high as 90%, but typically between 40-70%. Take the mobile phone address book example. The majority of the functionality required to implement an address book are infrastructure functions, such as FLASH programming, file system, database, sorting, searching, and graphical user interface. The application consists only of tying keystrokes with appropriate actions.

Since the infrastructure is such a large part of most systems, and the only portion of the system that is somewhat generic in its nature, we should turn our focus towards re-using the infrastructure rather than complete applications. An address book application is difficult to re-use, but a file system, database, or graphical user interface is not!

2.1.1 Oh No, Here comes Another Platform!?

The *infrastructure*, although related, should not be confused with a *platform*. A platform consists mainly of infrastructure functions, which have been selected and tied together based on the requirements of a specific application domain. Typically, the lower end of the application program layer is also included. I.e., a platform is comprised of: *Infrastructure + Hardware Integration + Part of the Application Program*.

In other words, a platform is an infrastructure implementation for a specific application domain.

2.2 Identifying Your Infrastructure

Until this point we have only discussed infrastructure in very general terms, and provided a few illustrative examples. It's time to make a more detailed description and classification of the different functions that are included in an infrastructure.

As illustrated in *Figure 3*, infrastructure is a relative concept, both in terms of generality and in terms of classification. The closer you move towards the two extremes (hardware and application), the less generic the infrastructure becomes. E.g., if we move too close to the hardware we are moving into the area of device drivers, which in our classification are not

considered a part of the infrastructure. This section aims at providing some hints to the question: How do I identify the software infrastructure in my embedded system?

2.2.1 You Say Data Structure, I Say File System

If we stick to the definition of an infrastructure as consisting of functionality common to a large set of applications, we do not have to discuss whether a certain function should be considered a device driver or not. Regardless of what classification is employed, the spectrum of the infrastructure is very broad. *Table 1* below provides some examples of the range of the infrastructure. Note: No interpretation of the extremes in terms of good or bad should be made (depending on where you live, cold/hot are positively or negatively charged words).

One Extreme ("Cold")	⇔	**Another Extreme ("Hot")**
Single, simple functions • Data structure (list and queue) • Error detecting checksum	⇔	**Multiple, complex functions** • File system • Pre-emptive RTOS
Hardware close • Ethernet driver • Serial port communication and buffering	⇔	**Abstract** • Database • Abstract data type (dictionary)
Specific (impl. one specific function) • TFTP communication protocol • Dynamic memory manager	⇔	**General (impl. a general function)** • Script interpreter • Graphical window interface platform

Table 1: Infrastructure Functionality Spectrum

There are many plausible ways of categorizing infrastructure functionality. However, since the spectrum is so broad, ranging from simple data structures to databases, we find that a categorization according to level of abstraction is suitable. In *Figure 4* below the infrastructure is divided into five groups.

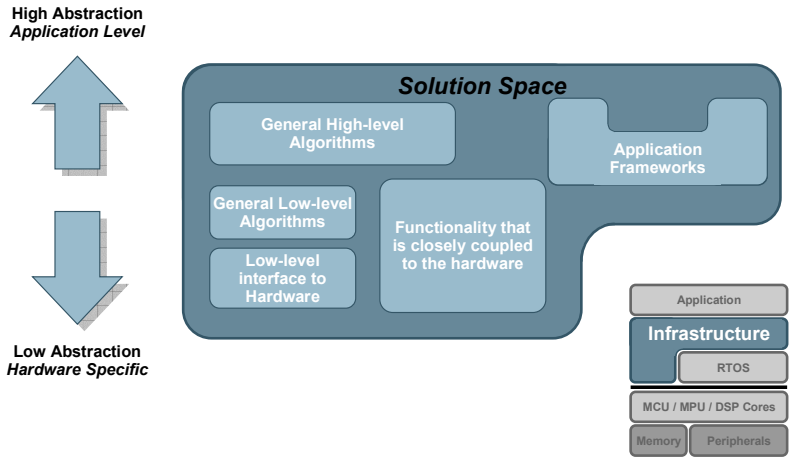


Figure 4: Infrastructure Solution Space

Category	Examples
Low-level Interface to Hardware	These functions communicate mainly with the hardware (on-chip or external peripherals) and act as device drivers: Memory programming (parallel / serial FLASH / E2PROM) Communication (serial port, USB, I2C, Ethernet driver, AT-command modem driver) LCD / graphical video chipset General purpose digital I/O, keyboard scanning
General Low-level Algorithms	These functions mainly implement one basic functionality: Mathematical (fractional / high precision / floating point arithmetic, trigonometric calculations, CORDIC algorithm, curve fitting) Functions (error logger, timer / alarm handler, dynamic memory manager) Algorithms (random numbers, error checksums, error correcting codes) Basic data structures (list, stack, vector, binary tree, B-tree, skip-list, bitmap, hash table, queue, etc.) Advanced data structures; ADT (dictionary, set, graph), balanced trees
General High-level Algorithms	These functions implement higher-level functionality (often built out of low-level functions): Compression Database package / Registry Parser / Interpreter / Script processor File system (RAM, ROM, FLASH) Web-server Event scheduler Graphical user interface Security (encryption, authentication, digital signature)
Functionality that is Closely Coupled to Hardware	These functions operate in close cooperation with the physical hardware: Pre-emptive operating system, incl. synchronization primitives Communication protocols; data link layer / network layer / transport layer (TCP/IP, Bluetooth, HDLC, IrDA, etc.) Communication protocols; Internet related (dynamic IP, NAT, (T)FTP, SMTP, POP3, SNMP, etc.)
Application Frameworks	These functions present application domain frameworks that can be programmed to perform application specific tasks: FSM (Finite State Machine) with Statechart design environment Custom communication protocol design environment Internet connectivity, with remote control / data retrieval functionality

Table 2: Infrastructure Group Examples

Every embedded product does not contain all of the functions exemplified in the table above, but at least a few of them can be found in every system. The similarities between products in a given application domain is of course large, but also completely different products contains many common infrastructure functions. The infrastructure functions in the address book application could just as well be used in an industrial controller (storing command sequences in a file system, logging diagnostic samples for trend analysis in a database, and having a graphical user interface in a small front LCD).

Highly optimized solutions impede re-use, since generality is traded for efficiency. As mentioned above, a web-server (on top of a TCP/IP stack) can store data in pre-computed IP-frames in order to minimize TCP/IP protocol processing. Such implementations will be very specific and cannot easily be moved to another platform operating under different conditions.

If a little more general implementation can be afforded the chances are much higher that an existing implementation can be modified for the new operating conditions without excessive re-design and programming.

2.2.2 Infrastructure Application Analysis

To further illustrate the importance and complexity of the infrastructure in modern embedded systems, let's take a look at an example application. Imagine a diagnostic unit that can be operated remotely (over a wireless communication link). It is basically a data logger with analyzing capabilities. The list below highlights the main product features:

- Up to 5 input channels can be monitored from the system that the unit is mounted on (e.g., a large motor). The sampled data records are stored in an internal file FLASH system. 1024 samples are buffered in a queue before transferred to the non-volatile file system storage.
- In order to be cost effective, the unit only communicates with a central at regular intervals. Alarms are signaled immediately (either as an e-mail or an SMS), but long-term trend data is uploaded once a week over a GSM/GPRS communication channel.
- The diagnostic unit can be remotely configured and controlled via a graphical web interface. A GSM/GPRS communication channel can be used for remote operation. An operator that works in close proximity to the unit can access the graphical web interface via a Bluetooth communication channel.
- The internal operation and signal processing (i.e., signal analyzing) is controlled through command scripts. Scripting facilitates future product upgrades.
- The system is resource-constrained (small amount of memory, low capacity processor) due to low production cost.
- Security is an important sales argument (all communication over the wireless channel is encrypted).

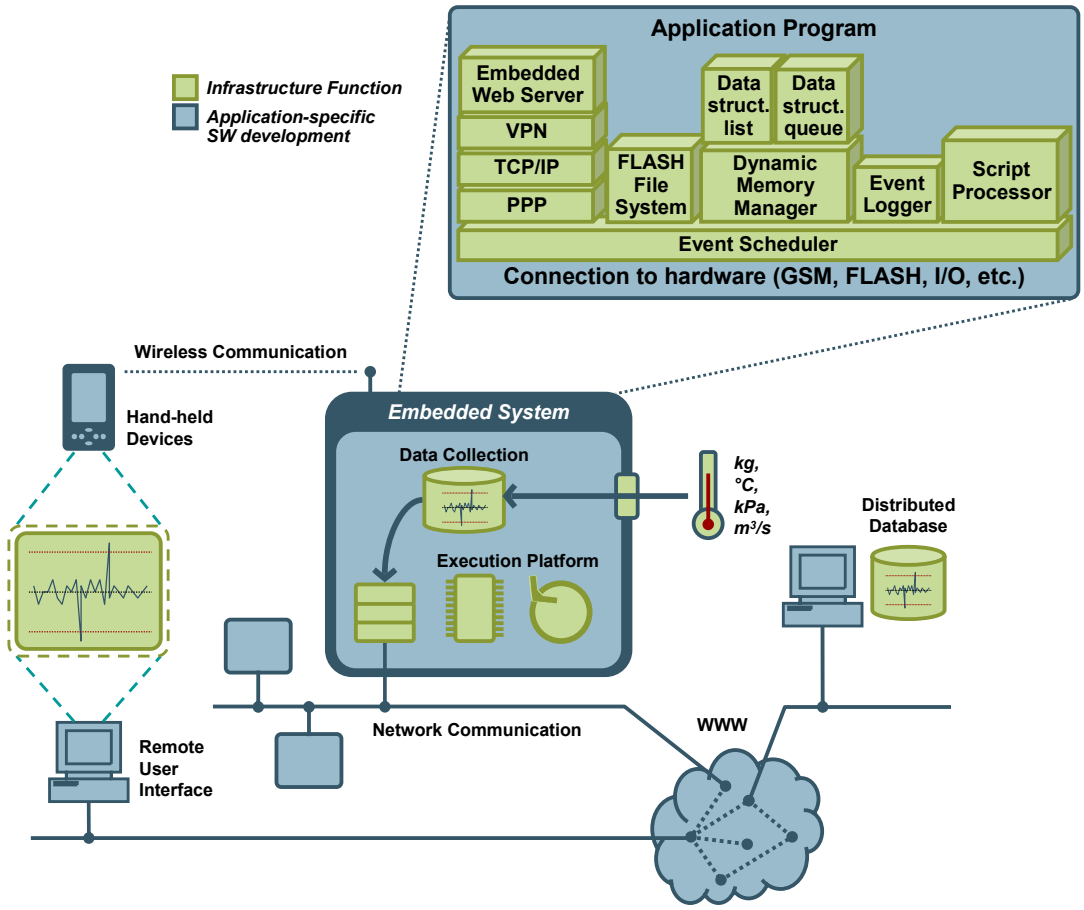


Figure 5: Application Example – Remote Diagnostic Unit (i.e., Wireless Data Logger)

Figure 5 above illustrates the data logger system and highlights the set of infrastructure functions required to implement the desired functionality. The different functions are described briefly.

- **Event Scheduler:** A full-blown pre-emptive real-time operating system is not necessary for this simple application. The system is event-driven and a simple event scheduler is only sufficient to coordinate all internal activity.
- **FLASH File System:** A FLASH file system is required to store (non-volatile) the data samples and command scripts. The file system can be very simple, it only needs to handle fixed size files and no hierarchy is necessary.
- **PPP:** The communication protocol PPP (Point-to-Point Protocol) is used to communicate over the wireless communication channel (GSM/GPRS). The protocol also contains important authentication functions, which are essential to protect the system from unauthorized access.
- **VPN:** A VPN (Virtual Private Network) function is needed to establish a secure communication channel. This functionality can possibly be handled by PPP (which is a huge protocol family that also contains encryption layers).
- **TCP/IP:** The standard communication protocol that allow seamless integration of the diagnostic unit with existing communication infrastructure.
- **Embedded Web Server:** An embedded web-server is used to provide a graphical user interface based on standard Internet technology.
- **Dynamic Memory Manager:** A dynamic memory manager is required to handle temporary data structures (for example lists and queues for data samples) and protocol processing (TCP/IP, PPP, and web server).
- **Data Structures:** A couple of simple data structures are needed to implement different functions (single-linked list and FIFO-queue).
- **Script Processor:** A general script processor is required to interpret and execute the stored command scripts (which control the main functionality of the data logger).
- **Event Logger:** An event logger is very useful in systems operating in the field. All important events are logged in a circular buffer, and can be examined if a problem / system crash is encountered during operation. The buffer will of course have a finite time horizon, but to know the last system events (before a system crash) greatly facilitates fault-tracking and debugging.

The remaining part of the program is the actual application - the product specific functionalities and all the glue code (for tying infrastructure functions together and interfacing to the hardware). At least 50% of the code in this application example can be referred to as infrastructure functions.

3 Re-using Infrastructure

So far, we have only argued that the feature explosion in many embedded systems can be attributed to infrastructure functionality. The software infrastructure for embedded systems has been defined, categorized, and exemplified. The infrastructure constitutes a significant portion of the software development time, and the commercial offer of infrastructure functions by no means covers the complete spectrum (see *Appendix A – Can I Buy Infrastructure?*). There is no doubt that if more infrastructure functions could be bought, the development time could be significantly reduced for many embedded projects. Unfortunately, there are a couple of obstacles in the way of achieving effective re-use of infrastructure functionality.

Even though typical infrastructure functionality may appear similar between applications (from a high-level view), the finer implementation details most likely differ greatly. It's not as simple as just creating a standard library of infrastructure functions that can be re-used directly. If this would be a feasible approach, it would have been done a long time ago. Also, most functions can be implemented in many different ways, depending on what trade-offs are made. The interfaces to the specific functions can also be implemented in many different ways (how to exchange data, blocking or call-back interface, how to signal errors, etc.).

3.1 Get Your Priorities Straight

A universal conclusion is that most infrastructure functions don't have *one* best implementation. Take sorting for example; there is no superior sorting algorithm that performs best in every situation. The influences are numerous: input data properties, principal storage data structure, best mean performance or guaranteed worst-case performance, primary memory usage, sequential/random access or input/output data, is stable sorting wanted or not, are record move or compare most costly, and one/multiple execution processors. The same principles apply for most infrastructure functions. Hence, when implementing infrastructure functions you need to make many trade-offs.

3.1.1 There Is More than One Way to Skin a Cat

There are a number of universal trade-offs that reoccur in many program design situations. *Table 3* below presents these general trade-offs and exemplifies with the sorting function.

One Side of It…	⇔	**Another Side of It…**
Speed QuickSort with improvements – quite large footprint and uses stack memory.	⇔	**Code Size / Memory Consumption** InsertionSort – very small and simple but has non-optimal performance in many cases.
General Implementation QuickSort does a decent job in most situations.	⇔	**Specific Impelmentation** MergeSort can be very efficient if the dataset does not fit in main memory.
Good Mean Performance QuickSort – amongst the fastest sorting algorithms but may experience $O(N^2)$ performance.	⇔	**Guaranteed Worst-case Performance** HeapSort – about half the speed of QuickSort but guaranteed O(N*logN) performance.

Table 3: General Trade-Offs

Below is a short discussion around each of these trade-off aspects:

- **Speed vs. Code Size / Memory Consumption**
  This trade-off has a larger scope than the usual compiler directive (optimize for speed or code size). Different algorithms can usually be selected, which perform the same task, but with very different behavior. Again, sorting algorithms illustrates this well; there exist very simple and compact algorithms that have non-optimal performance in many situations, and there exist more complex (larger code size) algorithms that perform optimal in some situations.
- **Specific vs. General Implementation**
  Is good performance under very specific conditions sufficient? Or is good performance required under more general conditions? Many embedded systems operate under well-known conditions, which sometimes allow the implementations to be very specific and optimized for those particular conditions. For example, the fragmentation functionality in a TCP/IP protocol stack can be removed in the IP protocol processing, if you know that other nodes in the network will not fragment protocol frames.
- **Good Mean Performance vs. Guaranteed Worst-case Performance**
  Since many embedded systems are hard real-time systems, guaranteed worst-case performance is often a must. However, guaranteed performance often has a trade-off with increased mean performance. Again, take the sorting example. Quick-sort is about two times as fast as heap-sort, but can under certain circumstances have $O(N^2)$ performance. Heap-sort, on the other hand, has guaranteed $O(N*logN)$ performance.

Another very important aspect in embedded systems is *memory usage*. It has in general many trade-offs; static versus dynamic allocation of required memory. If dynamic allocation is chosen, additional options present themselves; variable-sized, multiple fixed-sized, or single fixed-size (with chaining) allocation.

3.1.2 Driving Factors

Every solution has its pros and cons. The specific problem in mind must dictate what trade-offs to make. There are many factors that influence the specific trade-offs that are made:

- **Cost**
  Embedded systems are usually resource-constraint systems due to low target manufacturing costs. The amount of memory and computational power is often very limited. Low-power operation can also influence the memory and CPU resources.
- **Responsiveness**
  Embedded systems are typically real-time systems, with either hard or soft requirements. The common word is responsiveness, i.e., it is not acceptable that any part of the system hogs a resource for an extensive period of time.
- **System Architecture**
  A system can be inherently serial or concurrent, and can have extreme hard real-time or moderate responsiveness requirements. It can be event-driven or time-driven. These aspects dramatically influence the suitability of a certain infrastructure implementation.
- **Scope of Program**
  A system can be one specific implementation (highly optimized to achieve lowest possible production cost), or it can have a platform-like design, which allows future migration. Generality, as mentioned before, often has high penalties in terms of overhead and performance.

Unfortunately, the trade-offs described above are seldom fully described and explained by companies selling "off-the-shelf" software packages. Advertising material uses phrases like: "*developed for embedded applications*" or "*minimal footprint*". No design decision can be

based on this type of information. User's Guides seldom reveal the inner implementation, since such information is often regarded as trade secrets.

Sometimes this level of detailed information is not desired - certainly not by a time-pressured engineer. However, there are many times when this information would be very useful, e.g., when tracking down bugs or when integrating the software package with an existing code.

Speaking of the time-pressured engineer; this person is typically faced with three challenges when developing software for embedded systems, as illustrated in *Figure 6* below. Take out one of the three demands (quality, excess of functionality, and time-to-market), and the task becomes much simpler. However, all together embedded software development becomes really difficult to manage.

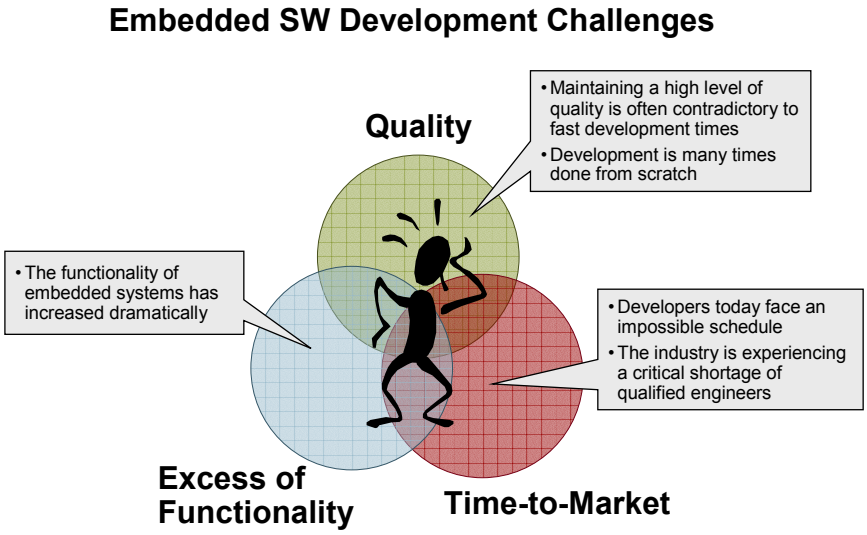


Figure 6: Embedded Software Development Challenges

4 The InfraBed™ Concept

This chapter presents Embedded Artists' solution for effective infrastructure software development for embedded systems, called ***InfraBed***. It is basically configurable software with an associated graphical configuration tool, which provides a simple point-and-click configuration mechanism. However, the program packages are much more configurable than `#define/#ifdef` preprocessor directives normally found in program packages. The graphical configuration tool also gives extensive guidance in the configuration process, in order to make sure the generated software implementation is adapted to the intended application and can be integrated seamlessly.

4.1 Principal Concept

The InfraBed technology is based on the concept of highly configurable software components called ESIC™ (Embedded System Infrastructure Components). Each ESIC provides one specific part of the software infrastructure of embedded systems. The principal workflow of InfraBed is illustrated in the *Figure 7* below. InfraBed is specifically designed to work with your other embedded software tools such as compilers, debuggers, and IDEs (Integrated Development Environment).

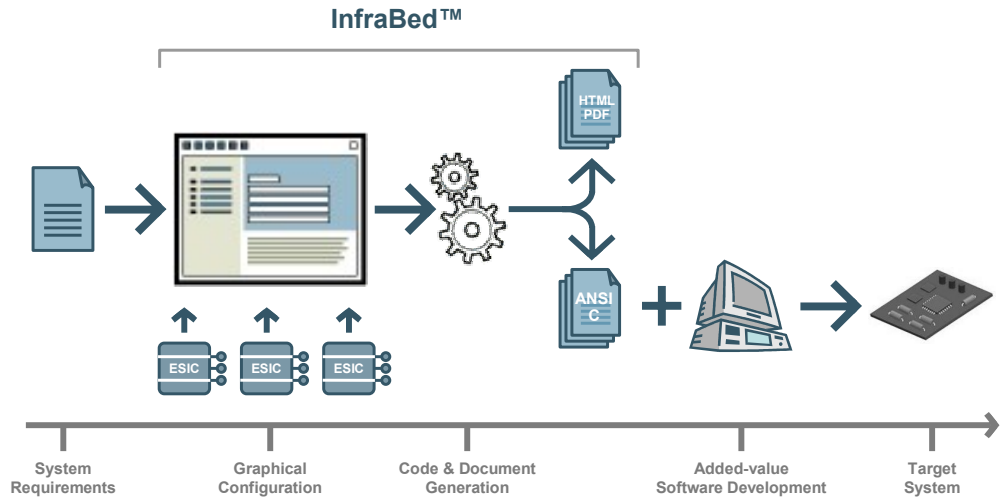


Figure 7: Principal Workflow

1. **Selection**: Based on the system requirements, the user selects one or more ESICs from a repository or database. InfraBed provides a comprehensive set of software components.
2. **Configuration:** The user then configures the components (ESICs) to fit the target applications requirements. The configuration is accomplished via a graphical configuration tool through a simple point-and-click mechanism. For each choice the user makes, direct feedback is provided regarding the consequences of that choice on the principal behavior, memory consumption, and performance. Each component also comes with an extensive reference documentation, which provides an in-depth description and analysis of the infrastructure functionality.
3. **Automatic Code Generation:** Once the ESIC has been configured, the source code along with all relevant documentation is automatically generated by InfraBed. In addition to the source code of the desired functions, InfraBed can also automatically generate test and debug programs that facilitate debugging and verification of the final application.
4. **Added-value Software Development:** Once the source code has been generated, the application development can commence, but with an enormous head-start. Access to the

generated software is accomplished through well-defined interfaces, specified by the user. This drastically minimizes the required integration work since an interface suitable for the particular system architecture can be defined.

4.2 Scope of InfraBed

InfraBed provides a comprehensive set of Embedded System Infrastructure Components (ESIC). Each ESIC provides one specific part of the software **infrastructure** of embedded systems. The infrastructure represents the software layer between the hardware and the application program, as illustrated in *Figure 8* below (copy of *Figure 3*). The closer you move towards the hardware or the application, the more specific the functionality becomes. However, the infrastructure contains functions that are quite generic between large groups of application domains, such as data structures, operating systems, communication protocols, security functions, and structured data storage.

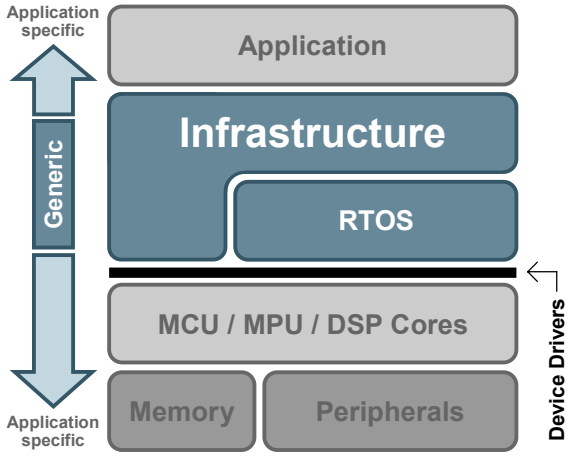


Figure 8: InfraBed - Targeting the Infrastructure of Embedded Systems

While the infrastructure functionality is generic, the requirements differ greatly! Each application has different requirements on the memory consumption, execution speed, interfaces, etc. For some applications worst-case performance is important, for others mean performance is more relevant. The trade-off between speed and low memory consumption is another example of a very diverse requirement. Depending on the requirements, completely different implementations might be required for the same principal functionality. Hence, one solution does not fit all, in the same manner as one sorting algorithm does not fulfill every applications needs.

InfraBed does not provide one single solution, but rather a selection of suitable implementations which can be customized by the developer to best suit the target applications needs. The essence of InfraBed is that no software supplier can make all the necessary trade-offs of every application, only the developer (who knows all application specific requirements) can make those decisions.

4.3 Target Hardware and Operating Systems

The software infrastructure generated by InfraBed is highly configurable, which means it can be adapted to fit your particular applications requirements on resources, performance, and interfaces. All of the software generated by InfraBed is 100% ANSI C compliant; no hardware or RTOS specific functions are used. The final connection, i.e., glue code, thus has to be made by the developer. To minimize this development effort, a configurable interface towards the hardware/RTOS layer is provided by each ESIC. Based on the target hardware/RTOS, a suitable interface can be selected and configured, e.g., a polling or event-driven interface towards a UART.

Figure 9 below illustrates the platform integration of InfraBed. One set of interfaces are selected and configured for *Platform A*. The platforms *B1* and *B2* on the other hand utilize another set of interfaces. Since *B1* and *B2* are quite similar, the differences between them are instead handled via the glue code provided by the developer. The configurable interfaces of an ESIC thus greatly simplify the integration towards a specific target platform, i.e., minimize the amount of glue code that has to be provided to make a suitable connection.

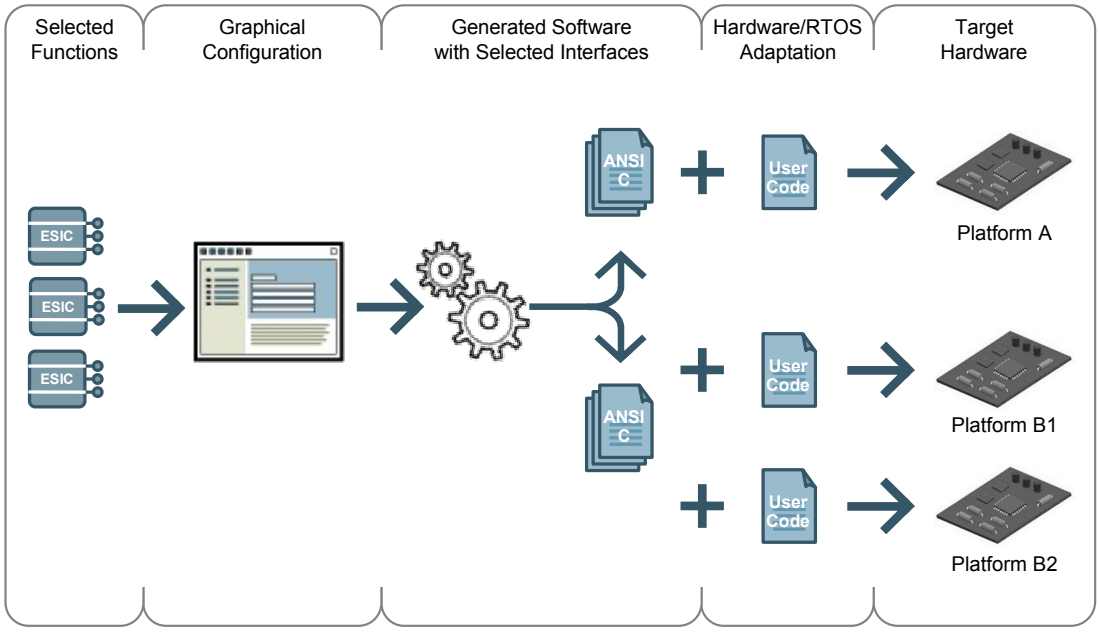


Figure 9: Platform Independent Software Development

All software generated by InfraBed can be used stand-alone or in conjunction with your choice of operating system. InfraBed also provides a highly configurable pre-emptive RTOS suitable for small resource-constraint systems.

4.4 ESIC™ – Embedded System Infrastructure Component

The core of InfraBed is the configurable software components called ESICs (Embedded System Infrastructure Component). ESICs are based on Embedded Artists' patent-pending component concept and provide packaged source code and expert knowledge of typical infrastructure function such as a communications protocol stack or an embedded database.

The user's of InfraBed can configure the components and generate source code optimized for their target applications needs. This configuration is accomplished graphically through a simple point-and-click mechanism. Direct feedback regarding the consequences on memory consumption and performance of the user selections is provided during the configuration process.

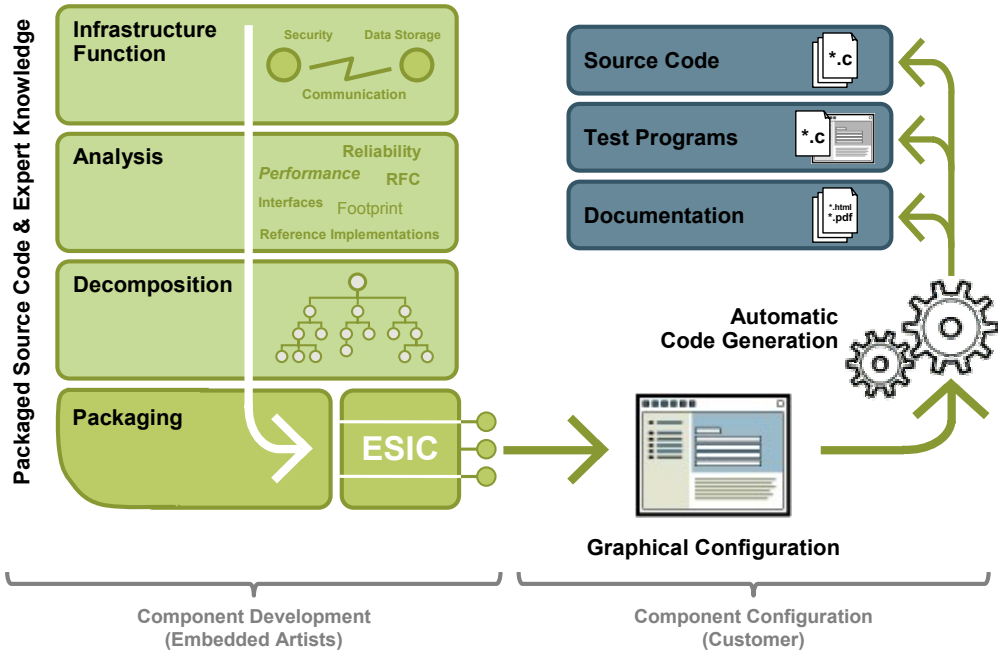


Figure 10: The InfraBed Concept

Figure 10 above illustrates the principal concept of InfraBed:

1. **Infrastructure Functionality.** The InfraBed concept takes on a functional approach to software development. The development of a new components starts with the identification of a generic infrastructure function. This could be e.g., a communication protocol, a security mechanism or structured data storage.
2. **Analysis.** Each infrastructure function undergoes an exhaustive requirements analysis, where the impact of every aspect is examined; footprint, performance, compliance to standards, reference designs, etc. Since one single solution does not fit all situations, each individual requirement might necessitate a completely different architecture and/or implementation approach.
3. **Decomposition.** Based on the analysis, the infrastructure function is broken down to a number of fine-grained building blocks and represented in a tree structure. Each node describes one sub-functionality and its implications on memory consumption, performance, and principal behavior. If alternative implementations are necessary, several nodes will exist for the same functionality, but with different implications on the behavior of the resulting code.
4. **Packaging.** All necessary source code, documentation, and information gathered during the analysis phase, are packaged into a re-usable, highly configurable, software component, called ESIC.

4.4.1 Levels of Configurability

Each ESIC allows the end-user to configure its functionality, performance, and interfaces. For this purpose, four types of configurations are provided by an ESIC, as illustrated in the *Figure 11* below:

- **Configuration of the functionality**
- **Configuration of the interfaces**
  - Hooks
  - High level interfaces
  - Low level interfaces

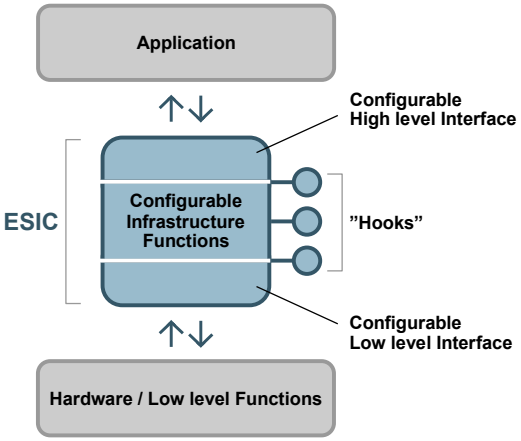


Figure 11: Levels of Configurability

Configurable Functionality and Performance

An ESIC is highly modularized and allows the user to select which features or optimizations that should be included. The final solution is thus extremely slimmed and adapted to your projects specific needs.

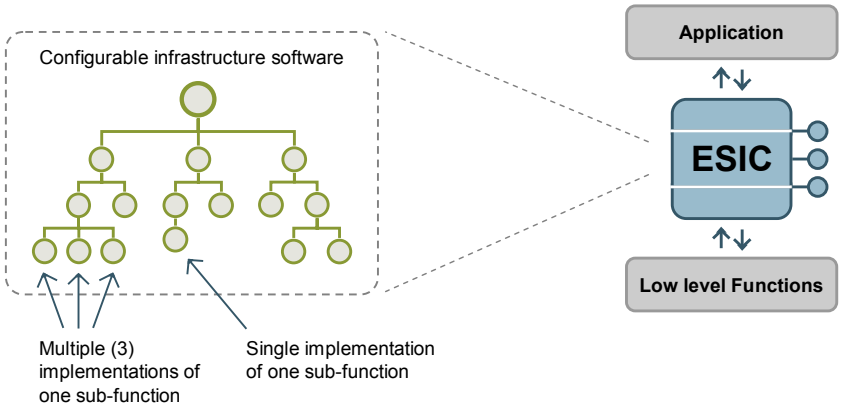


Figure 12: Configurable Infrastructure Functionality

Also, a specific function or algorithm often has many plausible implementations, with very different effects on code size, memory consumption, and performance. For some functions multiple implementations are provided by an ESIC to suit different application requirements. *Figure 12* above shows an ESIC with multiple and single implementations of specific sub-functions.

Configurable Interfaces

The interfaces of an ESIC are defined from a service perspective, i.e., the application code accesses an ESIC function through the *high level interface* (requests a service) and the ESIC returns the results (if any). If the ESIC in turn requires some other (low-level) service to carry out the request, this service is accessed through the *low level interface* of the ESIC. When carrying out the requested service, an ESIC supports calling optional user-defined functions called *hooks*. *Figure 13* below illustrates the interfaces of an ESIC.

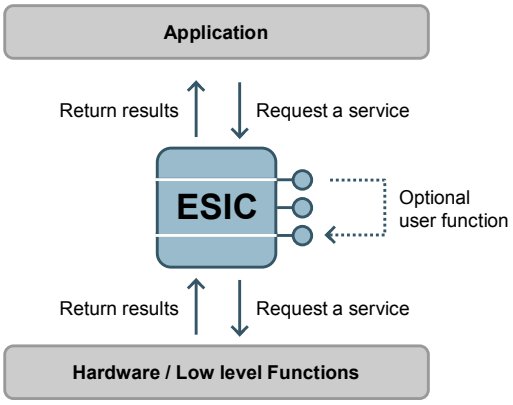


Figure 13: ESIC Interfaces

- **Hooks:** To enable user-specific processing hooks are provided which allow a user function to "hook" into the normal execution flow. Hooks are intended primarily for the inclusion of optional functionality, i.e., functions that are not critical for the operation of the ESIC such as gathering statistics or logging of internal events. Since hooks interrupt the normal execution flow of an ESIC, they should be kept as short as possible to ensure that the ESIC can perform its intended function within its given parameters.
- **High Level Interface:** The configurable interface towards the application enables the developer to choose how to interface towards the infrastructure function (callback or blocking interface, copying or non-copying interface, etc.). This greatly minimizes the integration work for the application programmer, since the application interface can be customized to match the system architecture of the target application.
- **Low Level Interface:** As the generated ESIC code is completely independent of the target hardware or operating system, no hardware specific or operating system commands are utilized. Instead, a highly configurable low level interface is provided. The low level interface of an ESIC enables the selection/configuration of interface aspects such as principal access mechanism, where and how interface data is stored, if and how data is passed between the layers, etc. Only the final access (service request) to the hardware or operating system function must be provided by the user, since this is very application and hardware specific.

4.5 Component Configuration

The configuration of an ESIC is accomplished via a graphical configuration tool through a simple point-and-click mechanism, as illustrated in *Figure 14* below. Each ESIC is represented by a selection tree, where the user can select which features to include, implementation strategy, interfaces, etc.

In addition to the selection tree, ESICs also provide on-line expert guidance during the configuration process. For each choice the user makes, direct feedback is provided regarding the consequences on the principal behavior, memory consumption, performance, etc. Since an ESIC can have a quite complex configuration structure, where a certain choice may be dependant on other choices, the configuration tool filters out invalid choices and automatically validates the consistency of the user configuration via a built in rule checking mechanism.

The configuration tool also provides on-line access to the comprehensive reference documentation that is packaged with each ESIC. The reference documentation provides an in-depth description of the infrastructure functionality, e.g., which principal approaches exist, what their pros and cons are, specific considerations for embedded systems, etc.

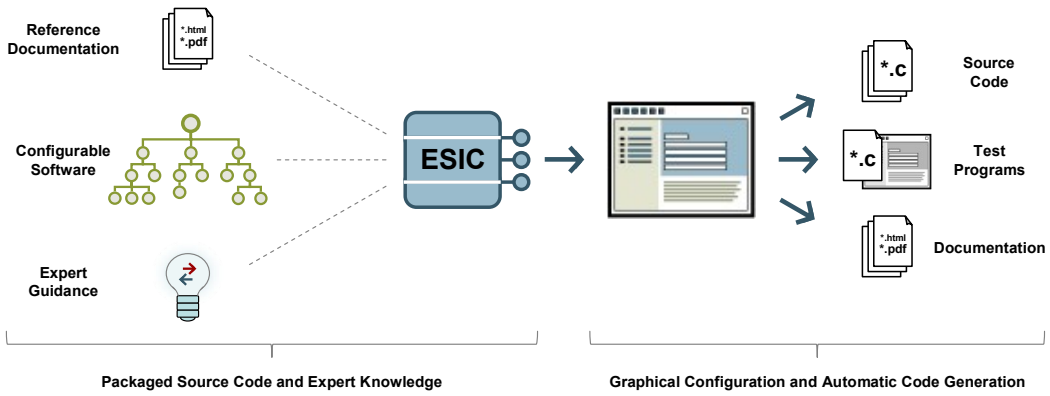


Figure 14: ESIC Configuration

Based on the user configuration, a code generator automatically generates all the source code required to implement the infrastructure functionality. The source code is highly optimized, based on the requirements the user provided during the configuration phase. Due to the extreme configurability of an ESIC it is virtually impossible to test all possible combinations of an ESIC. Therefore, each ESIC also contains all the necessary test programs required to verify and test the generated source code. This is accomplished through the verification of each individual user choice, i.e., each option in the configuration structure generates a test sequence. By running the extensive test program suite on the target platform, correct operation is verified, and a stable basis for application development is ensured.

In addition to the source code, all accompanying documentation is generated automatically. The generated documentation contains a description of the API, principal design of the source code, application examples, and a “to-do” list which describes the necessary steps that must be taken to integrate the function with the target hardware and application code. Similar to the source code, the documentation is highly adapted to include only the selected features and interfaces. I.e., no “general” documentation is generated that describes features or interfaces that are not used by the target application.

5 Around the Corner

If you take a peek around the corner, you will discover a very prevalent trend within the embedded industry, namely that of configurable software solutions. There are many examples of products that can be configured by compiler switches, which provide more or less detailed control. But this is only the beginning. Most packages offer include/exclude type of control. To be really useful (and support infrastructure software reuse) these switches must also control internal implementation strategies and interface operation.

The trend is also very apparent in other, but closely related, areas such as hardware design. Today, there are configurable processor cores that are configured basically the same way as a software package. The result is a netlist implementing the selected processor core (with all its trade-offs made during the configuration process).

Within the next few years, configurable infrastructure software will be offered as "off-the-shelf" code packages. **InfraBed** from Embedded Artists is one example of such a product. The real-time operating system market has reached the farthest in this area, with products like OSEK and eCos (actually an open-source product). Other infrastructure functions will most likely appear in the near future and offer valuable shortcuts to time-constrained embedded systems developers.

Commercial configurable packages will require massive development efforts (developing configurable software is typically an order more complex than developing a specific solution). Fortunately, these development costs can be spread over many users and will result in increased code quality (since many users help test the software instead of one or two).

Impatient developers will have to develop these configurable software packages themselves. Such a solution would most likely make use of `#define`/`#ifdef` –constructs, and would be extremely time-consuming. Most embedded product developers do not have the luxury of spending an order more time on a file system, while struggling with shorter time-to-market and increased functionality in their products. Meanwhile, there is still work that can be done.

It is well-spent time to identify and analyze your specific infrastructure needs. This knowledge is necessary in order to devise a sound outsourcing strategy, and in order to be able to evaluate a commercial offering. Also, as we have argued earlier, even though a commercial package exists, many developers choose to implement the functionality themselves. Switching to a commercial solution does not happen over night. It requires trust (in the quality of the software, in the sustainability of the supplier), openness (how the function is implemented, availability of complete and accurate documentation), flexibility (easy adaptation and extension of the software), and last but not least, professional and timely support. But, maybe more important than anything else, it requires a change of mind-set; what is core business, and can we handle loss of control in return of shorter time-to-market?

The future will be manageable for embedded systems developers; it's just a matter of knowing your core business. Identify Your Infrastructure!

6 Biography

Anders Rosvall (Anders.Rosvall@EmbeddedArtists.com) is a founding member and CTO of Embedded Artists AB, a company focused on providing configurable infrastructure software for embedded systems. Rosvall has a strong industrial background with various positions within the ABB group. His last position was at ABB Corporate Research where he was responsible for the technical area Embedded Systems and Communication. Rosvall has a long experience in embedded system design, both hardware and software. Rosvall is also lecturer at several of Sweden's leading universities, in classes covering C programming, data structures and algorithms, data communication, and digital signal processor programming. Rosvall holds an MSEE from Linköping Institute of Technology, Sweden.

Jan-Erik Frey (Jan-Erik.Frey@se.abb.com) is employed at ABB Corporate Research, working with business development for industrial wireless applications. Frey has a strong industrial background from the pulp and paper and automotive industries. He has previously been responsible for the technical area Diagnostics and Decision Support Systems. Frey holds a BSEE from Mid-Sweden University, Sweden and an MSIE from Georgia Institute of Technology, Atlanta, USA.

Appendix A – Can I Buy Infrastructure?

Many of the infrastructure functions mentioned in *section 2.2* and *Table 2* above can be bought today, but certainly not all of them, and not from a single vendor. The list below provides a general overview of the commercial offering.

- **Real-time Operating Systems:** Pre-emptive, co-operative, and event controlled
- **Internet-related Communication Protocols:** TCP/IP, PPP, SLIP, DHCP/BOOTP, (T)FTP, SMTP, POP3, SNMP, SSL, and web servers
- **Other Commonly Used Protocol Stacks:** Bluetooth, HDLC, USB, etc.
- **Security:** Encryption, Authentication, One-way Hashing, etc.
- **File Systems:** FLASH file systems, FAT-compatible, IDE/ATA/CF/MMC/SD-interface
- **Graphical Window Packages:** interface to graphical displays (LCD)

These groups by far do not cover the complete spectrum of infrastructure functions. There are a couple of reasons for this:

- **Big is Beautiful, Even in the Embedded World.** Many of the functions are too small, or too simple to be commercially viable. A certain "size" of the code is required in order to be able to sell it. "*If it's too small, I might as well build it myself*".
- **One Size does NOT Fit All.** There are often many different ways to implement a specific infrastructure function. If the options are too many, it's difficult to create one "piece of code" that fulfills enough customer requirements.

Even if a commercial product exists, developers quite often choose to develop their own solutions. There are many reasons for these (seemingly illogical) decisions:

- **Devil in Disguise.** It is often difficult to know exactly how the function is implemented. It's especially difficult to evaluate this at an early stage in a project since design specifications may not be completely frozen (the exact requirements are simply not known). Further, peculiarities in an implementation have a tendency to show up at a very late stage of a project. At that point it is very difficult to make a re-design or select another software package.
- **Bloated Code.** There is a commercial aspect (as already indicated above). Producers try to increase the value of their products by adding more and more functionality. A commercial package often includes much more than required by the specific project in mind. Sometimes, but not always, part of this excessive code/functionality can be removed by setting compiler switches. The trend towards more bloated code is clearly visible in the embedded industry.
- **Core Business.** The design group regards the infrastructure function in mind as core business, and therefore "must" develop it in-house. For example, surveys show that 20-25% of all TCP/IP protocol stacks have been developed in-house! The truth is that not many infrastructure functions (as exemplified in *Table 2* above) are core-business.
- **It's a Jungle Out There.** Although the industry for embedded design tools is not too large, it can still be difficult for a development group to have a complete overview of the market. The supplier of a specific infrastructure function may simply not be known.
- **Not-Invented-Here Syndrome.** "Not invented here"-syndrome and the fact that it might be fun to develop the functionality in-house, are certainly relevant aspects that have some weight in an "outsourcing" decision.

Appendix B – Can I do it Myself?

Is re-use of infrastructure functions a dream? Implementing a set of generic functions, where each of them has millions of possible implementations, certainly gives that impression. Luckily most embedded product manufacturers do not have to take on a forest of selection trees when implementing specific functions. As mentioned earlier, not even close to all the functions listed in *Table 2* are required by one single product, or even by a whole product family. Also, very few products are forced to implement all possible implementation permutations. Typically, one to three different implementations is more than enough.

So, is re-use in general a dream?

Not at all… but it is a lot of hard work the traditional way!

The key is to make traditional domain analysis – but with a twist…***only regard the infrastructure!***

The road to reuse of infrastructure is basically a three step process:

1. Identify Your Infrastructure
2. Make the Necessary Trade-Offs
3. Define Your Interfaces

Hence, make a careful selection of infrastructure functionality that is common within your product family. Based on a thorough requirements analysis, you then need to make the necessary trade-offs in the implementation of the selected functions. Finally you must define proper interfaces for your infrastructure.

6.1 Identify Your Infrastructure

The most attractive alternative is to find an "off-the-shelf" code package that implements the required functionality. If you are really fortunate, you will find a perfect match, or at least a package that requires minimal interface/wrapper design to fit into the existing system. However, a not uncommon scenario is that no suitable package can be found, and you will have to implement the infrastructure functionality yourself.

The selection of infrastructure functions can be a tricky task. What is infrastructure to you? As we mentioned earlier, Infrastructure is a relative concept, and defining the smallest common denominator within your product family is not always straightforward. There is no cook book solution, where, based on a few documented steps, you can pull your infrastructure out of the oven. When defining the infrastructure, it's important that as many as possible of your products are examined, otherwise you might end up with a suboptimal set of functions suitable for only some of your applications.

Don't over-do it. Sometimes going for one set of functions, which shall satisfy all of your products needs, is often a distant utopia (or simply not worth the extra work). Try to define a suitable hierarchy of functions. If possible share them among product families, if not, make separate blocks or keep them in the application layer. *Figure 15* below illustrates a hierarchy of infrastructure functions employed in two product families.

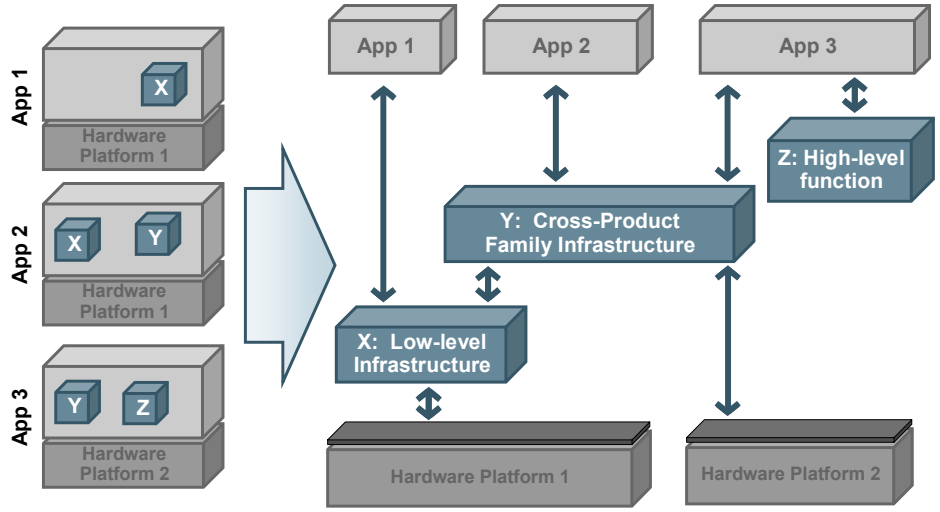


Figure 15: Hierarchy of Infrastructure Functions

Also, it's not merely a question of deciding whether a function belongs to the infrastructure or not. Many times is a more question of abstraction level. E.g., you might have several applications that make use of a database. However, the way you persist the data, might vary. A suitable infrastructure function could thus be a database, where the persistence level is kept open and implemented in each application. Therefore, having deep knowledge of one single product is insufficient. To be able to achieve effective re-use, it's essential to have a good idea of the "big picture".

6.2 Make the Necessary Trade-offs

Once you have divided the world into nice manageable boxes, the real work commences. Since many applications may make use of an infrastructure function, the function often has to accommodate many diverse requirements. The more basic the functionality, the more requirements are posed. Hence, simple data structures, which are used by a vast number of both infrastructure and application functions, can become quite complex. To implement every possible scenario is many times not feasible. You need to compromise.

A through requirements analysis is an absolute prerequisite to be able to make the necessary trade-offs. Very important is not to forget to encompass market requirements in your analysis. Many times we are so focused on the technical solution that we forget what we are selling. Factors such as target cost and future expansions can drastically alter the design of your system. Commonly flexibility is traded for efficiency, and this decision is not always up to the engineer. An extremely slim solution might be the preferred solution in some situations. Other times, the market strategy is to be flexible and able to accommodate future requirements more quickly.

If you have very diverse requirements on your infrastructure, implementing one standard function might not be feasible. There are a couple of ways to tackle such a situation. Try to break down the function in mind into smaller building blocks and see if there is a common denominator among your applications. A common mistake is that we take re-use to the extreme. Many times it's much more efficient to stop half way and make application-specific implementations for the remaining functionality. If this does not work, try going back to square one, and re-define your infrastructure. Maybe there is some lower level block that is common.

Another way to handle diverse requirements is to make multiple implementations. There might not be one universal solution. This is of course more costly, but it's anyway better to have two versions of a specific infrastructure function than a separate implementation for

each individual application. The additional work of providing multiple implementations can be alleviated through the use of `#defines` and `#ifdef` constructs. This is very suitable for specific, internal functions.

6.3 Define Your Interfaces

Just as important as the internal implementation, is the definition of the interfaces towards your infrastructure functions. In fact, the selection of a specific type of interface can alter the internal implementation of a function. In the same manner as described in the previous section, the definition of the interfaces has to be guided by your entire product family. Be careful not to make the interfaces too specific for one application. Again, many times it's better to stop half-way and make the final connection (i.e. glue code) for each application.

The work of defining a proper interface resembles that of making trade-offs in the implementation. You need to look at both how the application (or other high-level functions), as well as hardware/low-level functions access the infrastructure. Very diverse applications might require multiple interfaces. Apply the same reasoning as above. Supplying "hooks" or callbacks at strategic places in your infrastructure code can help accommodate a larger variety of applications. This enables you to clearly isolate application-specific processing from your infrastructure code. *Figure 16* below illustrate the interfaces.

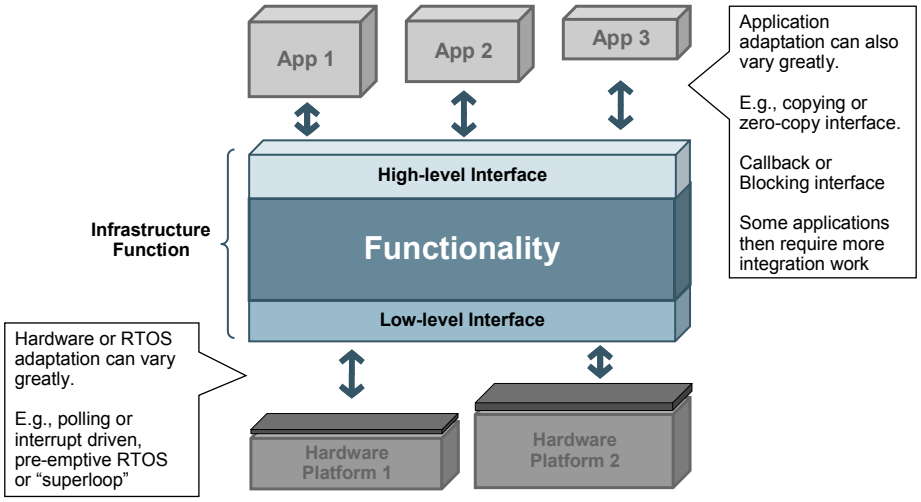


Figure 16: Infrastructure Interfaces