Hardware Abstraction Architecture

TEP:	2
Group:	Core Working Group
Type:	Best Current Practice
Status:	Final
TinyOS-Version:	2.x
Author:	Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, David Culler, David Gay

Abstract

This TEP documents a *Hardware Abstraction Architecture (HAA)* for TinyOS 2.0 that balances the conflicting requirements of code reusability and portability on the one hand and efficiency and performance optimization on the other. Its three-layer design gradually adapts the capabilities of the underlying hardware platforms to the selected platform-independent hardware interface between the operating system core and the application code. At the same time, it allows the applications to utilize a platform's full capabilities -- exported at the second layer, when the performance requirements outweigh the need for cross-platform compatibility.

1. Introduction

The introduction of hardware abstraction in operating systems has proved valuable for increasing portability and simplifying application development by hiding the hardware intricacies from the rest of the system. However, hardware abstractions come into conflict with the performance and energy-efficiency requirements of sensor network applications.

This drives the need for a well-defined architecture of hardware abstractions that can strike a balance between these conflicting goals. The main challenge is to select appropriate levels of abstraction and to organize them in form of TinyOS components to support reusability while maintaining energy-efficiency through access to the full hardware capabilities when it is needed.

This TEP proposes a three-tier *Hardware Abstraction Architecture (HAA)* for TinyOS 2.0 that combines the strengths of the component model with an effective organization in form of three different levels of abstraction. The top level of abstraction fosters portability by providing a platform-independent hardware interface, the middle layer promotes efficiency through rich hardware-specific interfaces and the lowest layer structures access to hardware registers and interrupts.

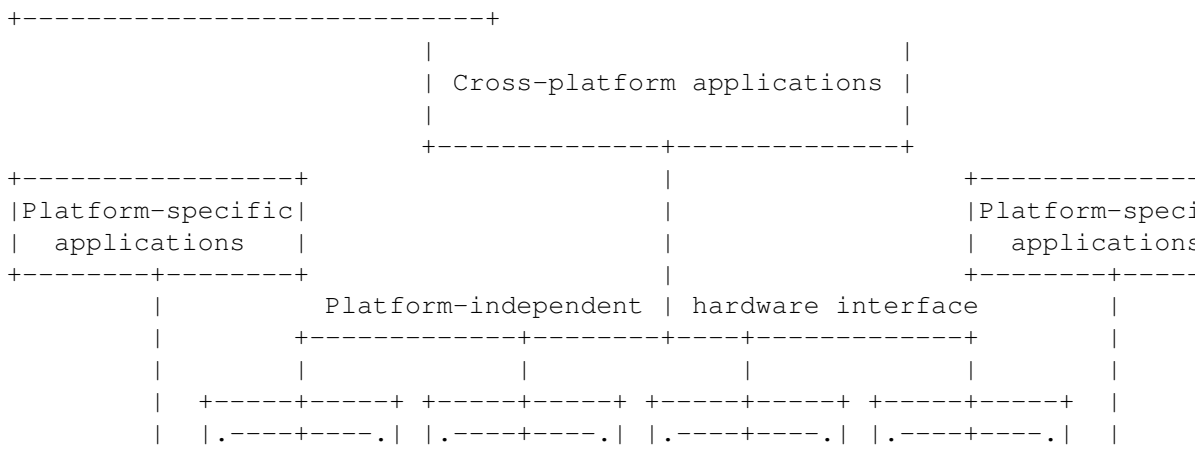
The rest of this TEP specifies:

- the details of the *HAA* and its three distinct layers (2. Architecture)
- guidelines on selecting the "right" level of abstraction (3. Combining different levels of abstraction)
- how hardware abstractions can be shared among different TinyOS platforms (4. Horizontal decomposition)
- the level of hardware abstraction for the processing units (5. CPU abstraction)
- how some hardware abstractions may realize different degrees of alignment with the *HAA* top layer (6. HIL alignment)

The *HAA* is the architectural basis for many TinyOS 2.0 documentary TEPs, e.g. [TEP101], [TEP102], [TEP103] and so forth. Those TEPs focus on the hardware abstraction for a particular hardware module, and [TEP112] and [TEP115] explain how power management is realized.

2. Architecture

In the proposed architecture (Fig.1), the hardware abstraction functionality is organized in three distinct layers of components. Each layer has clearly defined responsibilities and is dependent on interfaces provided by lower layers. The capabilities of the underlying hardware are gradually adapted to the established platform-independent interface between the operating system and the applications. As we move from the hardware towards this top interface, the components become less and less hardware dependent, giving the developer more freedom in the design and the implementation of reusable applications.



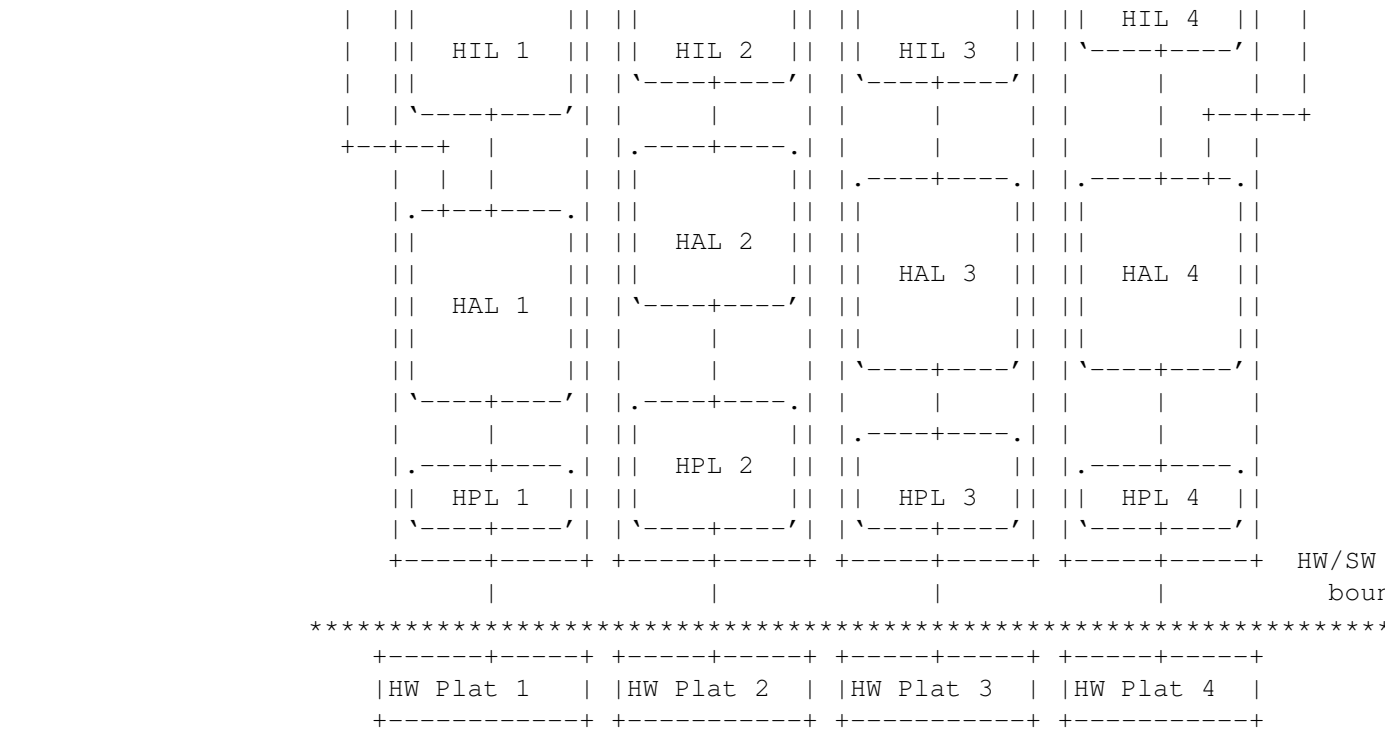


Fig.1: The proposed Hardware Abstraction Architecture

In contrast to the more traditional two step approach used in other embedded operating systems like [WindowsCE], the three-level design results in increased *flexibility* that arises from separating the platform-specific abstractions and the adaptation wrappers that upgrade or downgrade them to the current platform-independent interface. In this way, for maximum performance, the platform specific applications can circumvent the *HIL* components and directly tap to the *HAL* interfaces that provide access to the full capabilities of the hardware module.

The rest of the section discusses the specific roles of each component layer in more detail.

Hardware Presentation Layer (HPL)

The components belonging to the *HPL* are positioned directly over the HW/SW interface. As the name suggests, their major task is to "present" the capabilities of the hardware using the native concepts of the operating system. They access the hardware in the usual way, either by memory or by port mapped I/O. In the reverse direction, the hardware can request servicing by signaling an interrupt. Using these communication channels internally, the *HPL* hides the hardware intricacies and exports a more readable interface (simple function calls) for the rest of the system.

The *HPL* components SHOULD be stateless and expose an interface that is fully determined by the capabilities of the hardware module that is abstracted. This tight coupling with the hardware leaves little freedom in the design and the implementation of the components. Even though each *HPL* component will be as unique as the

underlying hardware, all of them will have a similar general structure. For optimal integration with the rest of the architecture, each *HPL* component SHOULD have:

- commands for initialization, starting, and stopping of the hardware module that are necessary for effective power management policy
- "get" and "set" commands for the register(s) that control the operation of the hardware
- separate commands with descriptive names for the most frequently used flag-setting/testing operations
- commands for enabling and disabling of the interrupts generated by the hardware module
- service routines for the interrupts that are generated by the hardware module

The interrupt service routines in the *HPL* components perform only the most time critical operations (like copying a single value, clearing some flags, etc.), and delegate the rest of the processing to the higher level components that possess extended knowledge about the state of the system.

The above *HPL* structure eases manipulation of the hardware. Instead of using cryptic macros and register names whose definitions are hidden deep in the header files of compiler libraries, the programmer can now access hardware through a familiar interface.

This *HPL* does not provide any substantial abstraction over the hardware beyond automating frequently used command sequences. Nonetheless, it hides the most hardware-dependent code and opens the way for developing higher-level abstraction components. These higher abstractions can be used with different *HPL* hardware-modules of the same class. For example, many of the microcontrollers used on the existing sensornet platforms have two USART modules for serial communication. They have the same functionality but are accessed using slightly different register names and generate different interrupt vectors. The *HPL* components can hide these small differences behind a consistent interface, making the higher-level abstractions resource independent. The programmer can then switch between the different USART modules by simple rewiring (*not* rewriting) the *HPL* components, without any changes to the implementation code.

Hardware Adaptation Layer (HAL)

The adaptation layer components represent the core of the architecture. They use the raw interfaces provided by the *HPL* components to build useful abstractions hiding the complexity naturally associated with the use of hardware resources. In contrast to the *HPL* components, they are allowed to maintain state that can be used for performing arbitration and resource control.

Due to the efficiency requirements of sensor networks, abstractions at the *HAL* level are tailored to the concrete device class and platform. Instead of hiding the individual features of the hardware class behind generic models, *HAL* interfaces expose specific features and provide the "best" possible abstraction that streamlines application development while maintaining effective use of resources.

For example, rather than using a single "file-like" abstraction for all devices, we propose domain specific models like *Alarm*, *ADC channel*, *EEPROM*. According to the model, *HAL* components SHOULD provide access to these abstractions via rich,

customized interfaces, and not via standard narrow ones that hide all the functionality behind few overloaded commands. This also enables more efficient compile-time detection of abstraction interface usage errors.

Hardware Interface Layer (HIL)

The final tier in the architecture is formed by the *HIL* components that take the platform-specific abstractions provided by the *HAL* and convert them to hardware-independent interfaces used by cross-platform applications. These interfaces provide a platform independent abstraction over the hardware that simplifies the development of the application software by hiding the hardware differences. To be successful, this API "contract" SHOULD reflect the *typical* hardware services that are required in a sensornet application.

The complexity of the *HIL* components mainly depends on how advanced the capabilities of the abstracted hardware are with respect to the platform-independent interface. When the capabilities of the hardware exceed the current API contract, the *HIL* "downgrades" the platform-specific abstractions provided by the *HAL* until they are leveled-off with the chosen standard interface. Consequently, when the underlying hardware is inferior, the *HIL* might have to resort to software simulation of the missing hardware capabilities. As newer and more capable platforms are introduced in the system, the pressure to break the current API contract will increase. When the performance requirements outweigh the benefits of the stable interface, a discrete jump will be made that realigns the API with the abstractions provided in the newer *HAL*. The evolution of the platform-independent interface will force a reimplementation of the affected *HIL* components. For newer platforms, the *HIL* will be much simpler because the API contract and their *HAL* abstractions are tightly related. On the other extreme, the cost of boosting up (in software) the capabilities of the old platforms will rise.

Since we expect *HIL* interfaces to evolve as new platforms are designed, we must determine when the overhead of software emulation of hardware features can no longer be sustained. At this point, we introduce *versioning* of *HIL* interfaces. By assigning a version number to each iteration of an *HIL* interface, we can design applications using a legacy interface to be compatible with previously deployed devices. This is important for sensor networks since they execute long-running applications and may be deployed for years. An *HIL* MAY also branch, providing multiple different *HIL* interfaces with increasing levels of functionality.

3. Combining different levels of abstraction

Providing two levels of abstraction to the application --the *HIL* and *HAL*-- means that a hardware asset may be accessed at two levels in parallel, e.g. from different parts of the application and the OS libraries.

The standard Oscilloscope application in TinyOS 2.0, for example, may use the ADC to sample several values from a sensor, construct a message out of them and send it over the radio. For the sake of cross-platform compatibility, the application uses the standard `Read` interface provided by the ADC *HIL* and forwarded by the `DemoSensorC` component wired to, for example, the temperature sensor wrapper. When enough samples are collected in the message buffer, the application passes the message to the networking stack. The MAC protocol might use clear channel assessment to determine when it is safe to send the message, which could involve taking

several ADC samples of an analog RSSI signal provided by the radio hardware. Since this is a very time critical operation in which the correlation between the consecutive samples has a significant influence, the programmer of the MAC might directly use the hardware specific interface of the *HAL* component as it provides much finer control over the conversion process. (Fig.2) depicts how the ADC hardware stack on the MSP430 MCU on the level of *HIL* and *HAL* in parallel.

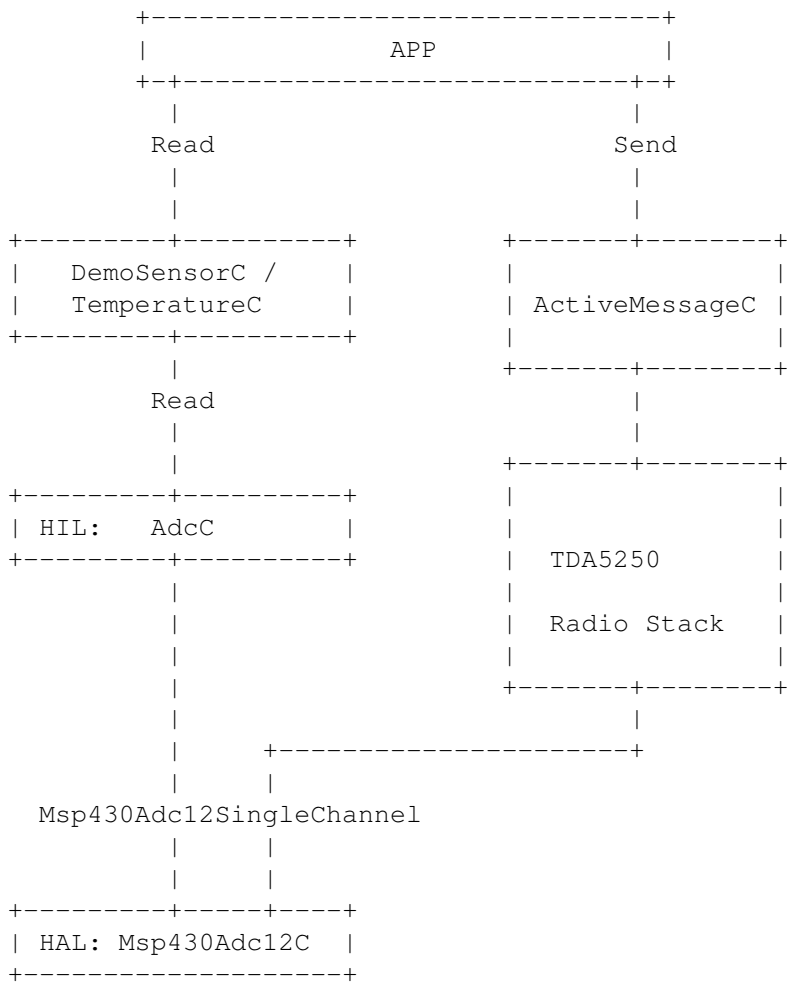


Fig.2: Accessing the MSP430 ADC hardware abstraction via *HIL* and *HAL* in parallel

To support this type of "vertical" flexibility the ADC *HAL* includes more complex arbitration and resource control functionality [TEP108] so that a safe shared access to the *HPL* exported resources can be guaranteed.

4. Horizontal decomposition

In addition to the *vertical* decomposition of the *HAA*, a *horizontal* decomposition can promote reuse of the hardware resource abstractions that are common on different platforms. To this aim, TinyOS 2.0 introduces the concept of *chips*, the self-contained abstraction of a given hardware chip: microcontroller, radio-chip, flash-chip, etc. Each chip decomposition follows the *HAA* model, providing *HIL* implementation(s) as the topmost component(s). Platforms are then built as compositions of different chip components with the help of "glue" components that perform the mapping (Fig.3)

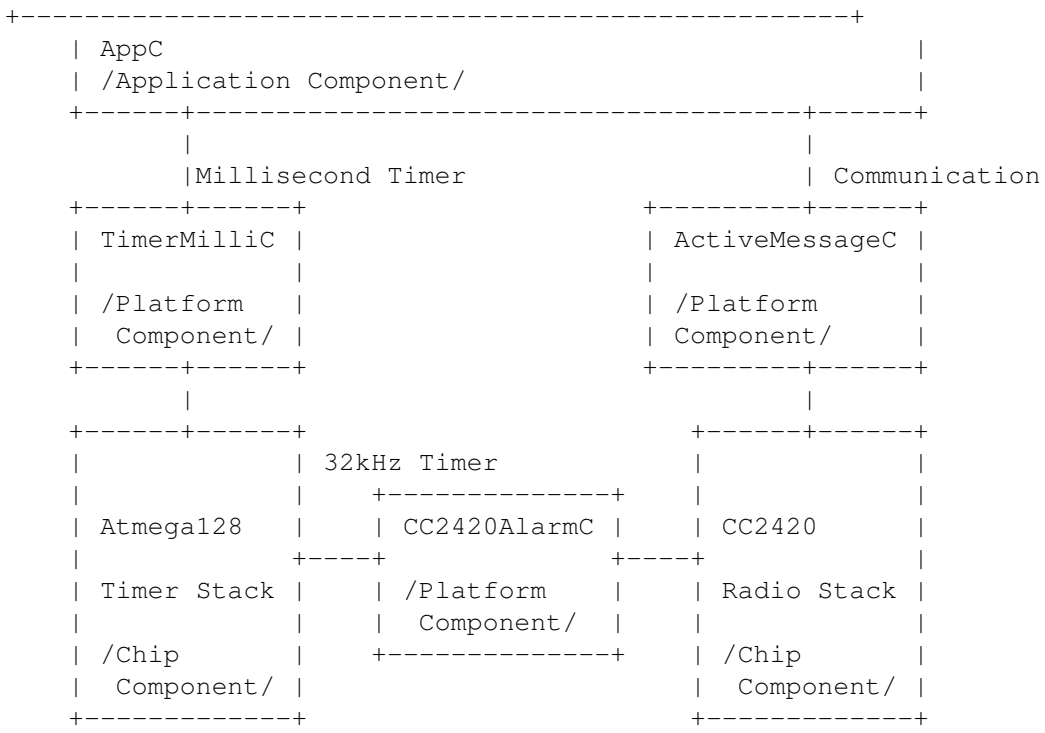


Fig.3: The CC2420 software depends on a physical and dedicated timer. The micaZ platform code maps this to a specific Atmega128 timer.

Some of the shared hardware modules are connected to the microcontroller using one of the standard bus interfaces: SPI, I2C, UART. To share hardware drivers across different platforms the issue of the abstraction of the interconnect has to be solved. Clearly, greatest portability and reuse would be achieved using a generic bus abstraction like in NetBSD [netBSD]. This model abstracts the different bus protocols under one generic bus access scheme. In this way, it separates the abstraction of the chip from the abstraction of the interconnect, potentially allowing the same chip abstraction to be used with different connection protocols on different platforms. However, this generalization comes at high costs in performance. This may be affordable for desktop operating systems, but is highly sub-optimal for the application specific sensor network

platforms.

TinyOS 2.0 takes a less generic approach, providing *HIL*-level, microcontroller-independent abstractions of the main bus protocols like I2C, SPI, UART and pin-I/O. This distinction enables protocol-specific optimizations, for example, the SPI abstraction does not have to deal with client addresses, where the I2C abstraction does. Furthermore, the programmer can choose to tap directly into the chip-specific *HAL*-level component, which could further improve the performance by allowing fine tuning using chip-specific configuration options.

The TinyOS 2.0 bus abstractions, combined with the ones for low-level pin-I/O and pin-interrupts (see [TEP117]), enable a given chip abstraction to be reused on any platform that supports the required bus protocol. The CC2420 radio, for example, can be used both on the Telos and on micaZ platforms, because the abstractions of the serial modules on the MSP430 and Atmega128 microcontrollers support the unified SPI bus abstraction, which is used by the same CC2420 radio stack implementation.

Sharing chips across platforms raises the issue of resource contention on the bus when multiple chips are connected to it. For example, on the micaZ the CC2420 is connected to a dedicated SPI bus, while on the Telos platform one SPI bus is shared between the CC2420 radio and the flash chip. To dissolve conflicts the resource reservation mechanism proposed in [TEP108] is applied: every chip abstraction that uses a bus protocol MUST use the `Resource` interface in order to gain access to the bus resource. In this way, the chip can be safely used both in dedicated scenarios, as well as in situations where multiple chips are connected to the same physical bus interconnect.

5. CPU abstraction

In TinyOS most of the variability between the processing units is hidden from the OS simply by using a nesC/C based programming language with a common compiler suite (GCC). For example, the standard library distributed with the compiler creates the necessary start-up code for initializing the global variables, the stack pointer and the interrupt vector table, shielding the OS from these tasks. To unify things further, TinyOS provides common constructs for declaring reentrant and non-reentrant interrupt service routines and critical code-sections.

The *HAA* is not currently used to abstract the features of the different CPUs. For the currently supported MCUs, the combination of the compiler suite support and the low-level I/O is sufficient. Nevertheless, if new cores with radically different architectures need to be supported by TinyOS in the future, this part of the hardware abstraction functionality will have to be explicitly addressed.

6. HIL alignment

While the *HAA* requires that the *HIL* provides full hardware independence (Strong/Real HILs), some abstractions might only partially meet this goal (Weak HILs). This section introduces several terms describing different degrees of alignment with the concept of a *HIL*. It also uses the following differentiation:

- *platform-defined X:* X is defined on all platforms, but the definition may be different
- *platform-specific X:* X is defined on just one platform

Strong/Real HILs

Strong/Real HILs mean that "code using these abstractions can reasonably be expected to behave the same on all implementations". This matches the original definition of the *HIL* level according to the *HAA*. Examples include the *HIL* for the Timer (TimerMilliC, [TEP102]), for LEDs (LedsC), active messages (ActiveMessageC, [TEP116], if not using any radio metadata at least), sensor wrappers (DemoSensorC, [TEP109]) or storage ([TEP103]). Strong *HILs* may use platform-defined types if they also provide operations to manipulate them (i.e., they are platform-defined abstract data types), for example, the TinyOS 2.x message buffer abstraction, `message_t` ([TEP111]).

Weak HILs

Weak HILs mean that one "can write portable code over these abstractions, but any use of them involves platform-specific behavior". Although such platform-specific behavior can --at least at a rudimentary syntactical level-- be performed by a platform-independent application, the semantics require knowledge of the particular platform. For example, the ADC abstraction requires platform-specific configuration and the returned data must be interpreted in light of this configuration. The ADC configuration is exposed on all platforms through the "AdcConfigure" interface that takes a platform-defined type (adc_config_t) as a parameter. However, the returned ADC data may be processed in a platform-independent way, for example, by calculating the max/min or mean of multiple ADC readings.

The benefit from weak *HILs* are that one can write portable utility code, e.g., a repeated sampling for an ADC on top of the data path. While code using these abstractions may not be fully portable, it will still be easier to port than code built on top of *HALs*, because weak *HILs* involve some guidelines on how to expose some functionality, which should help programmers and provide guidance to platform developers.

Hardware Independent Interfaces (HII)

Hardware Independent Interfaces (HII), is just an interface definition intended for use across multiple platforms.

Examples include the SID interfaces, the pin interfaces from [TEP117], the Alarm/Counter/etc interfaces from [TEP102].

Utility components

Utility components are pieces of clearly portable code (typically generic components), which aren't exposing a self-contained service. Examples include the components in tos/lib/timer and the ArbitratedRead* components. These provide and use HIIs.

6. Conclusion

The proposed hardware abstraction architecture provides a set of core services that eliminate duplicated code and provide a coherent view of the system across different platforms. It supports the concurrent use of platform-independent and the platform-dependent interfaces in the same application. In this way, applications can localize their platform dependence to only the places where performance matters, while using standard cross-platform hardware interfaces for the remainder of the application.

Author's Address

Vlado Handziski (handzisk at tkn.tu-berlin.de)[1]
Joseph Polastre (polastre at cs.berkeley.edu)[2]
Jan-Hinrich Hauer (hauer at tkn.tu-berlin.de)[1]
Cory Sharp (cssharp at eecs.berkeley.edu)[2]
Adam Wolisz (awo at ieee.org)[1]
David Culler (culler at eecs.berkeley.edu)[2]
David Gay (david.e.gay at intel.com)[3]

Citations

[1]Technische Universitaet Berlin Telecommunication Networks Group Sekr. FT 5, Einsteinufer 25 10587 Berlin, Germany

[2]University of California, Berkeley Computer Science Department Berkeley, CA 94720 USA

[3]Intel Research Berkeley 2150 Shattuck Ave, Suite 1300 CA 94704

[HAA2005] V. Handziski, J.Polastre, J.H.Hauer, C.Sharp, A.Wolisz and D.Culler, "Flexible Hardware Abstraction for Wireless Sensor Networks", in *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey, 2005.

[T2_TR] P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz, "T2: A Second Generation OS For Embedded Sensor Networks", *Technical Report TKN-05-007*, Telecommunication Networks Group, Technische Universitaet Berlin, November 2005.

[WindowsCE] "The WindowsCE operating system home page", *Online*, http://msdn.microsoft.com/embedded/windowsce

[NetBSD] "The NetBSD project home page", *Online*, http://www.netbsd.org

[TEP1] Philip Levis, "TEP structure and key words"

[TEP101] Jan-Hinrich Hauer, Philip Levis, Vlado Handziski, David Gay "Analog-to-Digital Converters (ADCs)"

[TEP102] Cory Sharp, Martin Turon, David Gay, "Timers"

[TEP103] David Gay, Jonathan Hui, "Permanent Data Storage (Flash)"

[TEP108] Kevin Klues, Philip Levis, David Gay, David Culler, Vlado Handziski, "Resource Arbitration"

[TEP109] David Gay, Philip Levis, Wei Hong, Joe Polastre, and Gilman Tolle "Sensors and Sensor Boards"

[TEP111] Philip Levis, "message_t"

[TEP112] Robert Szewczyk, Philip Levis, Martin Turon, Lama Nachman, Philip Buonadonna, Vlado Handziski, "Microcontroller Power Management"

[TEP115] Kevin Klues, Vlado Handziski, Jan-Hinrich Hauer, Philip Levis, "Power Management of Non-Virtualised Devices"

[TEP116] Philip Levis, "Packet Protocols"

[TEP117] Phil Buonadonna, Jonathan Hui, "Low-Level I/O"