In Proceedings of the 2nd European Workshop on WirelessSensor Networks (EWSN 2005)

# Flexible Hardware Abstraction for Wireless Sensor Networks

Vlado Handziski*, Joseph Polastre†, Jan-Hinrich Hauer*, Cory Sharp†, Adam Wolisz* and David Culler†

*Technische Universität Berlin; Telecommunication Networks Group
Sekr. FT 5, Einsteinufer 25, 10587 Berlin, GERMANY
†University of California, Berkeley; Computer Science Department
Berkeley, CA 94720 US

***Abstract*— We present a flexible Hardware Abstraction Architecture (HAA) that balances conflicting requirements of Wireless Sensor Networks (WSNs) applications and the desire for increased portability and streamlined development of applications. Our 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. We demonstrate the practical value of our approach by presenting how it can be applied to the most important hardware modules that are found in a typical WSN platform. We support our claims using concrete examples from existing hardware abstractions in TinyOS and our implementation of the MSP430 platform that follows the architecture proposed in this paper.**

## I. Introduction

The introduction of hardware abstraction in modern operating systems has proved valuable for increasing portability and simplifying application development by hiding the hardware intricacies from the rest of the system. Although enabling portability, hardware abstractions come into conflict with the performance and energy-efficiency requirements of Wireless Sensor Network (WSN) applications.

WSNs, with their application specific nature and severely constrained resources, push customization, rather than more general and reusable hardware abstraction designs. For maximum performance, the unique capabilities of the hardware should be made available unhindered to the application, but this can impede porting and rapid application development. By directly accessing the hardware, the application couples the evolution of the system with the underlying hardware.

Our analysis of several embedded and general-purpose operating systems (*eCos* [1], *WindowsCE* [2], *NetBSD* [3] and *Linux*[4]) that have mature hardware abstraction architectures has shown us that existing designs are poorly suited to the flexible abstraction required by WSNs. Their interfaces are rigid and expose the capabilities of the hardware at a single level of abstraction, preventing an application from taking full advantage of the hardware's capabilities when needed.

Thus, we need a better Hardware Abstraction Architecture (HAA) that can strike a balance between the two conflicting goals that are present in the WSNs context. The component-based model [5] is one framework that can provide the required tools to resolve this tension. The separation between the exposed interfaces and the internal implementation promotes modularity and reuse [6]. At the same time it allows rich interaction between building blocks. The main challenge is to select an appropriate organization of abstraction functionality in form of components to support reusability while maintaining energy-efficiency through access to the full hardware capabilities when it is needed. Based on our experience in porting TinyOS [7], [8] to new platforms we believe that an effective organization is possible when the strengths of the component-based approach are combined with a flexible, three-tier organization of the hardware abstraction architecture.

The rest of the paper is structured as follows: In Section II we introduce the details of the proposed architecture focusing on the functionality of each of the three tiers separately and the flexibility that the architecture provides in total. The following Section III, illustrates the practical applicability of our approach to the major hardware modules of a typical WSN platform. After discussing the related work in Section IV, we outline

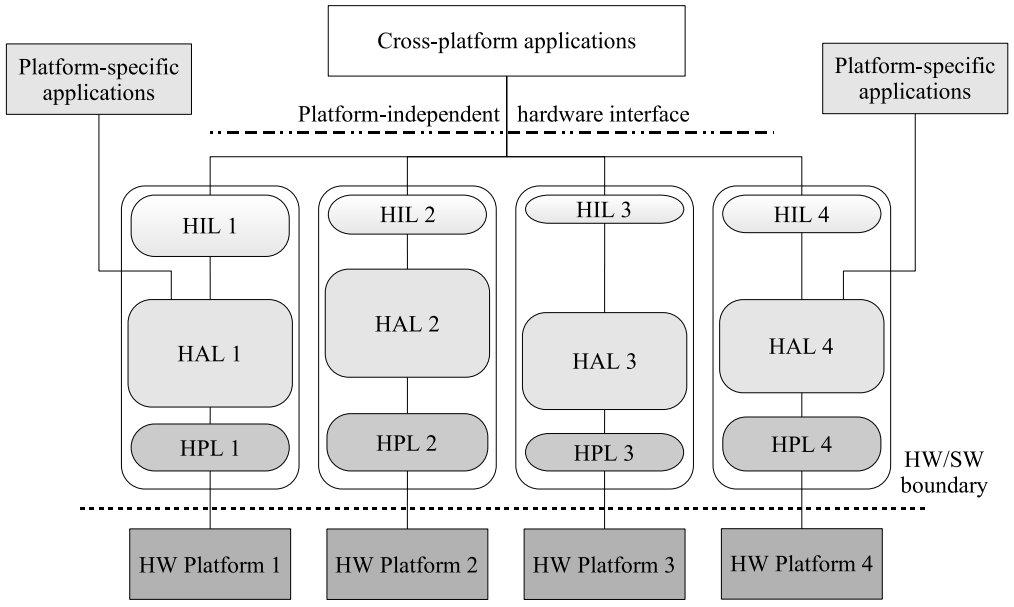



Fig. 1. The proposed hardware abstraction architecture

our planned future work (Section V) and conclude the paper in Section VI. For clarity, at the very end, we provide a list of the acronyms used in the text and their definitions.

## II. Architecture

In our 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.

### *A. 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 usable 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.

Our 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 WSN 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 (Section III-E), 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.

### *B. 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 WSNs, 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* as presented in Section III. According to our 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.

### *C. Hardware Interface Layer (HIL)*

The final tier in our 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 WSN 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 Hardware Adaptation Layers (HALs). 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 WSNs 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.

### *D. Selecting the level of abstraction*

The platform-dependence of the HAL in our architecture leads to the more general question about why we have opted for a three-layered design. In other words, why we did not expose the platform-independent hardware interface directly from the HAL components. The main reason behind our decision is the increased *flex-*

*ibility* 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.

Selecting the "right" level–whether an application should use the HIL or directly access the HAL–can sometimes cause one hardware asset to be accessed using two levels of abstraction from different parts of the application or the OS libraries. Let us take an application similar to the standard OscilloscopeRF application in TinyOS as an example. The application uses the Analog to Digital Converter (ADC) to sample several values from a temperature sensor and sends them in the form of a message over the radio. If the observed phenomenon does not have a large signal bandwidth and the time between subsequent conversions is long, for the sake of cross-platform compatibility, the programmer might decide to use the standard *ADCHILSingle* interface. This interface is exported by the HIL sensor wrapper (Figure 3) using the services of the platform-specific HAL component. When enough samples are collected in the message buffer, the application passes the message to the networking stack. The Media Access Control (MAC) protocol used for message exchange over the radio uses clear channel assessment to determine when it is safe to send the message. This usually requires taking several samples of the Receive Signal Strength Indicator (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 *MSP430ADC12Multiple* interface of the HAL component as it provides much finer control over the conversion process.

As a result of this chain of decisions, we end up with a concurrent use of the ADC hardware module using two different levels of abstraction. To support this type of "vertical" flexibility we include more complex arbitration and resource control functionality in the HAL components so that a safe shared access to the HPL exported resources can be guaranteed.

## III. Application to specific hardware modules

In this section we support our claims about the properties of the proposed architecture using real-world examples from the hardware abstraction functionality in TinyOS for different platforms. TinyOS does not have fixed rules or restrictions on how applications interface with the physical hardware. The original hardware abstraction was developed having specific capabilities of the Atmel family of microcontrollers in mind and with little consideration for the applicability to different hardware platforms. The organization of the existing hardware components generally follows a two-layered design that has some similarities with our HAA in the lowest HPL level, but uses different philosophy in separating the functionality at the higher abstractions.

The proposed HAA was applied for the first time during our implementation of the MSP430 platform [9] that abstracts the capabilities of the TI MSP430 microcontroller (Fig.2) in TinyOS 1.1.7. The implementation is currently being used by two hardware platforms (Telos and Eyes) and has quite successfully satisfied the requirements of a large range of applications.

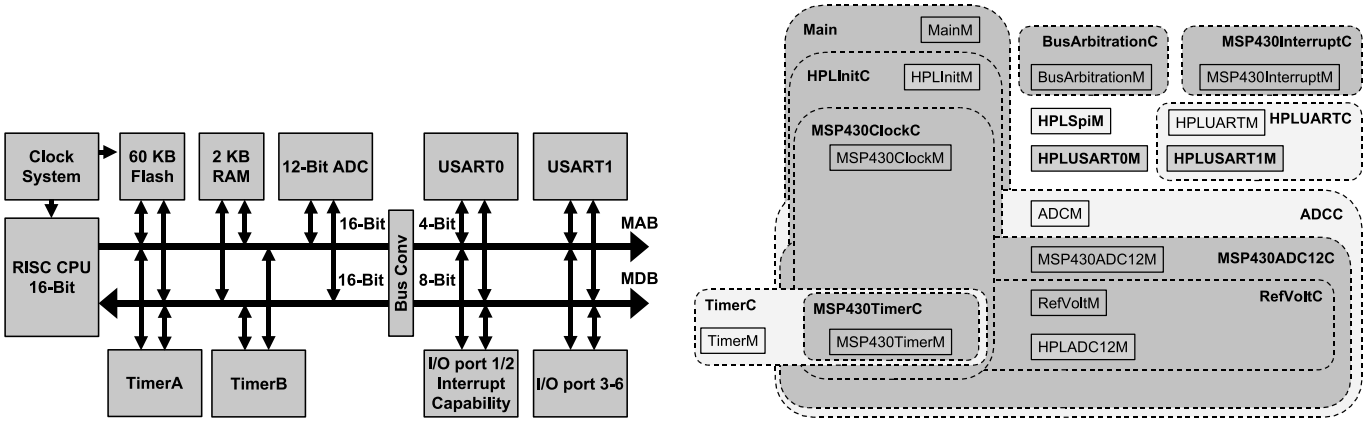
Based on the insight gained through this effort and our analysis of the hardware abstraction for the existing platforms in TinyOS (Mica, Mica2), we show how our HAA can be used for encapsulating the capabilities of the major hardware modules in a typical WSN platform [10]. For each module we discuss how the variability in the capabilities of the hardware might force different design decisions at separate layers and how this is finally reflected in the exposed interfaces.

### A. Processing unit

The microcontroller is the central part of a WSN platform. Microcontrollers have widely varying feature sets. A microcontroller consists of the MCU core processing unit, RAM, external interface pins, and a set of hardware modules. Examples of these modules include data bus support, analog to digital converters, and timers. Not only does the core vary, but the features provided by each module vary from vendor to vendor as discussed further later in this section.

MCU cores are available in 8-bit, 16-bit, and 32-bit architectures. For 8-bit and 16-bit cores, RAM sizes vary from 128 bytes to 16 kilobytes and program flash may be as little as 2 kilobytes or as big as 256 kilobytes. Usually 32-bit platforms have more RAM and flash storage at the expense of an order of magnitude higher energy consumption.

Abstracting the differences between the various MCU cores is the first step towards a more portable operating system. In TinyOS most of this variability is hidden from the OS simply by using a nesC/C based programming language with a common compiler suite (GCC). For ex-




(a) Functional block diagram of the TI MSP430F149 $\mu$C

(b) The components of the MSP430 platform abstraction in the original TinyOS 1.1.7 release

Fig. 2. Hardware abstraction of the TI MSP430 $\mu$C in TinyOS

ample, 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 MCU-specific tasks.

To unify things further, TinyOS provides mechanisms for declaring reentrant and non-reentrant interrupt service routines and critical code-sections. For the MCU's external pins, it provides macros that permit setting and clearing the pin, as well as changing its direction and function. For example, the TI MSP430's ADC pins may be used as either general I/O or as an analog input to the ADC hardware module. Macros are also provided for timed spin loops at microsecond resolution, independent of the microcontroller. These macros are defined in each platform's `hardware.h` descriptor file.

Finally, the HPL components deal with the different ways of accessing registers (memory-mapped or port-mapped I/O) using the definitions in the standard library header files. HPL implementations may also include code optimizations on a per platform basis for accessing hardware components–such as only fetching the needed 8-bits of a register on an 8-bit platform versus fetching the entire 16-bit register on a 16-bit platform.

Our three-layer architecture is not intended to abstract the features of the different MCU cores. For the currently supported MCUs, the combination of the compiler suite support with the thin abstraction in the *hardware.h* files 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.

### B. Power management

Power management is critical for long lived sensor network applications. Each microcontroller has a different power profile. A power profile consists of the low power modes provided by the microcontroller as well as the wakeup time (time to transition from sleep to active mode) and current consumption of each mode. Making the wakeup time semantically transparent to services running on the microcontroller is a difficult task. On Atmel based platforms where the wakeup time may be up to a few milliseconds, we use an HAL component that evaluates when the next timer event will occur, and only puts the Atmel into a sleep state if the next event occurs at a point in time longer than the wakeup time. On the TI MSP430, the wakeup time is "instantaneous"–under 6 $\mu$s–and therefore whenever the MCU is not processing, it is put into low power sleep mode.

On both the MSP430 and the Atmel, before entering a sleep mode, an HAL component checks if any hardware modules require that the MCU core is active. Additionally, all services including HPL and HAL components have an initialization, start, and stop function. When a service is no longer using a hardware module, it may call the stop function of the HPL or HAL component. Doing so disables the module for power savings, but also removes the MCU's dependence on that hardware module to enter sleep mode. For example, the ADC module may be clocked from a high speed oscillator. When a sample is not in progress, the ADC module may be shut down and it will no longer use the high speed oscillator. As a result, when the MCU is idle, it may

enter low power mode.

This rather efficient way of implementing the power management functionality is made possible by the fact that most of the hardware modules are on-chip, attached directly to the MCU system bus, and that there is no hardware memory protection hindering the access to their status registers. As TinyOS platforms add more external devices connected via the peripheral buses, this task will get increasingly complicated. Ultimately, keeping some state in the form of device enumeration or reference counting mechanisms might be needed for proper power management.

### C. Clocks and timers

A real-time crystal clock source can provide stable timing and interrupts for the microcontroller. These low-power clock sources enable the microcontroller to be woken from low-power sleep state by a timed alarm. This clock enables software timers with "jiffy" resolution – timing native to the oscillator, such as 30.5 microsecond resolution for a 32 kHz clock – and are generally also abstracted to one millisecond resolution. Because these clock sources remain powered, they do not need to stabilize after the MCU wakes from low-power mode, enabling a high-speed wake up into operational mode. And, because they provide a very stable clock source, they can be used to calibrate an internal fast-startup high speed digitally calibrated oscillator (DCO).

High speed clock sources are generated from an internal DCO (like the MSP430), from internal RC in the ADC unit, or a high frequency clock output from the radio (as in Eyes). Among other things, a high frequency clock can be used to drive a stable UART baud rate, SPI clock, or a high resolution ADC sampling timer.

The essence of the problem of creating a good software architecture is that different microcontrollers may provide different numbers of timers, different numbers of capture and compare registers, different counting modes, different scaling factors, and different resolution. For instance the MSP430 MCU on the Eyes and Telos platforms provides two distinct 16-bit timers, one with three capture/compare registers, the other with eight. Each timer can be given a distinct clock source and one of four scaling factors. Each compare register allows for a distinct interrupts based off the timer counter. The Atmel ATmega MCU on the MICA2 platform provides two 8-bit timers each with a 10-bit prescaler and one compare register, and two 16-bit timers each with a limited prescalers and three compare registers. These differences between platforms motivate the multiple-layer approach to maximally expose the hardware functionality at the lowest level while allowing for lowest common denominator reusable system services at the highest level.

The TimerM module in TinyOS provides timed periodic and one-shot millisecond resolution events, all multiplexed in essence from a single hardware timer compare register. The original TimerM in TinyOS was built for the Atmel ATmega MCU and attempted to provide a generic timer module based on a hardware specific abstraction of a clock. Because it was developed only for the Atmel platform, the lower clock component did not expose a platform independent design, presuming and exposing some of the more unique features of the Atmel timers. As a result, TimerM was not platform independent.

From our experience with the MSP430 on the Eyes and Telos platforms, we establish a hardware adaptation layer below TimerM that is more appropriate for both the Atmel and TI platforms. We design an *Alarm* interface and component that accesses the current time and permits setting alarms in the future relative to the past. Each interface describes the resolution of the Alarm that it provides, such as *TMilli*, *T32khz*, and *TMicro*.

The interface and configuration below show the Alarm interface as well as how it may be implemented as the HAL to a platform using resolution-specific interfaces. Each platform exports a number of interfaces for each resolution that correspond to the maximum number of alarms supported by that platform.

```
interface AlarmTMilli {
  async command uint32_t get();
  async command bool isSet();
  async command void cancel();
  async command void set(uint32_t t0, uint32_t dt);
  async event void fired();
}

configuration AlarmC {
  provides interface AlarmTMilli as AlarmTimerMilli;
  provides interface AlarmT32khz as AlarmTimer32khz;

  provides interface AlarmT32khz as Alarm32khz1;
  provides interface AlarmT32khz as Alarm32khz2;
  //...

  provides interface AlarmTMicro as AlarmMicro1;
  provides interface AlarmTMicro as AlarmMicro2;
  provides interface AlarmTMicro as AlarmMicro3;
  //...
}
```

Below these interfaces sits the hardware presentation layer. Without the presentation layer, hardware adaptation would be tied to very specific MCU registers and resources. Overflow flags, current hardware time, and system register settings are accessible from the HPL. The

HAL uses these primitives to build the Alarm interfaces accessible at the HAL.

At the HIL, we provide platform independent periodic and single instance timer events that provide a generic interface to applications. The interface used for the HIL timer is as follows:

```
interface Timer {
  command result_t setPeriodic(uint32_t dt);
  command result_t setOneShot(uint32_t dt);
  command result_t stop();
  command bool isSet();
  command bool isPeriodic();
  command bool isOneShot();
  command uint32_t getPeriod();
  event result_t fired();
}
```

All of the timers and alarm provided by our interface are 32-bit width. By choosing a 32 bits, we can support time synchronization, a local real time clock, and a variety of platforms with underlying timers of data width from 8-bits to 32-bits. For hardware timers that are less than 32-bits, they are emulated as a 32-bit timer by the HAL in software. The HIL then exposes this as a native 32-bit timer to applications and time synchronization services.

### *D. Analog-to-digital converters*

Just like with the Timers, the major challenge in defining a hardware abstraction for the Analog to Digital Converters (ADCs) is dealing with the variability in the hardware's capabilities. The ADC modules can differ in their resolution, the number of channels to be sampled and converted or the support for special conversion modes such as repeated or sequence conversion modes.

For example, the ADC12 module on the TI MSP430 MCU has 12-bit resolution and allows sampling and conversion of up to 8 external channels. Each channel can be assigned an individual reference voltage and one out of four clock sources can be selected to specify sampling and conversion timing. A sample-and-hold time, which represents the number of clock cycles in a sampling period, can be defined for each channel individually. The ADC12 on the MSP430 supports four different conversion modes the simplest being a single channel converted once. It also supports conversion of a sequence of channels where a sequence can be any combination of the available channels. There are 16 registers capable of storing channel number and reference voltage for the channels to be converted, i.e. the same channel can be sampled multiple times within a sequence conversion. Both of these modes are also available as repeat-modes, thus a single channel or a sequence of channels can be converted repeatedly. When multiple conversions are performed the MSP430 allows to define the time interval between subsequent conversions.

On the other side, the Atmel ATmega MCU on the Mica2 platform incorporates a 10-bit ADC. It supports 8 external channels and two different reference voltage levels. The clock source for sampling and converting is the MCU clock which can be prescaled. The sample-hold-time is a constant 13 cycles and is thus solely defined by MCU clock and prescaler. The internal ADC on the ATmega supports two conversion modes: Single channel conversion and free-running conversion which is a single channel being converted repeatedly.

If we compare these two ADC modules, it becomes obvious that they have some distinctive features that have to be gradually adapted before one can reach a general interface that can be used by the platform independent applications. The old *ADCM* module in TinyOS provides commands for triggering a single conversion and a repeated conversion for one channel. The *binding* of a component to the ADCM defines the port (channel) to be used for all subsequent conversions. Because the interface was geared towards the Atmel ATmega MCU it does not provide commands for controlling the sample-and-hold time or an individual reference voltage for different channels. Also there is, for example, no support for sequence conversion modes.

To overcome these limitations, we have developed a new ADC abstraction that follows our three-tier model. At the very bottom of the abstraction on each platform is the hardware presentation component that provides low-level access to all relevant registers and flags of the hardware ADC module, closely following the structure described in Section II-A.

The core of the abstraction is formed by the HAL component which will be explained for the MSP430 in the following. The HAL extends the existing concept of *binding* to include not only the channel for the conversion, but all of the per-channel settings supported by the hardware module as well. This component is also responsible for arbitration between multiple outstanding requests and for maintaining a state machine of the ADC's operation. By exposing all of the hardware-specific settings for each channel, the HAL interface provides unhindered access to the full capabilities of the MSP430's ADC12 module on a per-channel basis, resulting in very efficient sampling operations.

To promote clarity the four supported conversion modes of the MSP430 are separated into the interfaces *MSP430ADC12Single* and *MSP430ADC12Multiple* for

single and multiple conversions, respectively. Both interfaces provide two commands which allow the conversion(s) to be performed once or repeatedly. One characteristic of the MSP430 ADC12 is the ability to define the time interval between subsequent conversions. This is reflected by an additional parameter (*jiffies*) in the relevant commands of the HAL interfaces.

Our extended binding mechanism requires reconfiguration of the hardware for each sampling command. This process may take a non-negligible amount of time. Therefore a delay-sensitive application can *reserve* the ADC before calling one of the four conversion commands. A successful reservation will guarantee a minimum delay between the next invocation of the corresponding conversion command and the actual sampling of the channel. To cancel a reservation an additional *unreserve* command is provided.

The HAL for the MSP430 platform can be accessed by the following commands and events:

```
interface MSP430ADC12Single {
  command result_t bind(
          MSP430ADC12Settings_t settings);
  async command msp430ADCresult_t getData();
  async command msp430ADCresult_t getDataRepeat(
        uint16_t jiffies);
  async command result_t reserve();
  async command result_t reserveRepeat(
        uint16_t jiffies);
  async command result_t unreserve();
  async event result_t dataReady(uint16_t data);
}

interface MSP430ADC12Multiple {
  command result_t bind(
          MSP430ADC12Settings_t settings);
  async command msp430ADCresult_t getData(
        uint16_t *buf, uint16_t length,
        uint16_t jiffies);
  async command msp430ADCresult_t getDataRepeat(
        uint16_t *buf, uint16_t length,
        uint16_t jiffies);
  async command result_t reserve(uint16_t *buf,
        uint16_t length, uint16_t jiffies);
  async command result_t reserveRepeat(uint16_t *buf,
        uint16_t length, uint16_t jiffies);
  async command result_t unreserve();
  async event uint16_t* dataReady(uint16_t *buf,
        uint16_t length);
}

typedef struct {
  unsigned int refVolt2_5: 1;
  unsigned int clockSourceSHT: 2
  unsigned int clockSourceSAMPCON: 2;
  unsigned int clockDivSAMPCON: 2;
  unsigned int referenceVoltage: 3;
  unsigned int clockDivSHT: 3;
  unsigned int inputChannel: 4;
  unsigned int sampleHoldTime: 4;
} MSP430ADC12Settings_t;
```

The application-level OS service for the ADC is provided by HIL wrapper components that transform the platform-dependent settings for the HAL's *bind* command into a platform-independent representation of the individual sensor. The wrappers interact with the applications via the provided *ADCHILSingle* and *ADCHILMultiple* interface, and translate their requests to the underlying HAL primitives. An example configuration for a temperature sensor on the MSP430 platform is shown in Fig. 3.

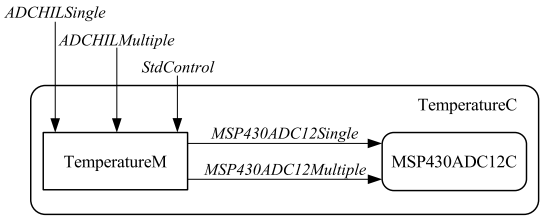



Fig. 3. The configuration for the temperature sensor HIL wrapper

The platform-independent HIL interfaces represent a compromise between the capabilities of the currently supported platforms. They resemble the interfaces provided by the HAL of MSP430, but do not support specifying a time interval between subsequent conversions. On the other hand they provide more extensive error information which is unnecessary for MSP430, but meaningful, for example, for external ADCs. Most importantly, the HIL interfaces do not include a *bind* command, because they cannot handle platform dependent settings. Instead, the commands in the HIL interfaces assume the wrapper to have bound to the HAL with appropriate settings. Despite the fact that the hardware of some ADCs might not support a multiple conversion mode (the ATmega's ADC does not) such a command is included in the HIL interface, because it can be easily emulated by software. The platform-independent HIL interfaces are defined in the following way:

```
interface ADCHILSingle {
  async command adcresult_t getData();
  async command adcresult_t getDataContinuous();
  async command adcresult_t reserve();
  async command adcresult_t reserveContinuous();
  async command adcresult_t unreserve();
  async event result_t dataReady(
        adcresult result,
        uint16_t data);
}

interface ADCHILMultiple {
  async command adcresult_t getData(
        uint16_t *buf, uint16_t length);
  async command adcresult_t getDataContinuous(
        uint16_t *buf, uint16_t length);
  async command adcresult_t reserve(
        uint16_t *buf, uint16_t length);
  async command adcresult_t reserveContinuous(
        uint16_t *buf, uint16_t length);
```

```
  async command adcresult_t unreserve();
  async event uint16_t* dataReady(adcresult result,
        uint16_t *buf, uint16_t length);
}
```

### *E. Data busses*

For any microcontroller to communicate with external digital hardware, it must do so through one of many standard data busses. These include SPI/USART, UART, $I^2C$, and 1-Wire busses. Only a subset of each bus is provided by various hardware–as a result some of these busses are implemented in software routines that use general purpose digital I/O pins. For those busses implemented with hardware, devices provide differing functionality. Our abstraction must export typical clock control including synchronous and asynchronous modes, clock source, clock prescale factors, and baud rate generation. To save power, submodules of the data bus module may be disabled to save energy or map to the external device's interface. For example, receive or transmit submodules of a UART may be disabled to meet a particular radio's control protocol or to save energy when no data is being transmitted to the device. Finally, when hardware supports double buffering, applications must be able to realize the performance gain from buffering.

The HPL functionality for the data busses includes two paths–one for data and a second for control. The control path allows the clock source, prescaler, and baud rate to be set. Interrupts may be enabled or disabled and various hardware flags may be read, set, or cleared, useful for polling or blocking implementations. Through the control path, the entire module may be started or stopped for power control. The data interface simply consists of sending and receiving a byte through the hardware's data registers, as well as interrupt based reporting of received data. Here is an example of the interfaces used in the MSP430 platform:

```
interface HPLUSARTControl {
  async command void enableUART();
  async command void disableUART();
  async command void enableUARTTx();
  async command void disableUARTTx();
  async command void enableUARTRx();
  async command void disableUARTRx();
  async command void enableSPI();
  async command void disableSPI();
  async command void setModeSPI();
  async command void setModeUART_TX();
  async command void setModeUART_RX();
  async command void setModeUART();
  async command void setClockSource(
        uint8_t source);
  async command void setClockRate(
        uint16_t baudrate, uint8_t mctl);
  async command result_t disableRxIntr();
  async command result_t disableTxIntr();
  async command result_t enableRxIntr();
  async command result_t enableTxIntr();
  async command result_t isTxIntrPending();
  async command result_t isRxIntrPending();
  async command result_t isTxEmpty();
  async command result_t tx(uint8_t data);
  async command uint8_t rx();
}

interface HPLUSARTFeedback {
  async event result_t txDone();
  async event result_t rxDone(uint8_t data);
}
```

Sometimes functionality for more than one bus protocol are supported through a single hardware module. In these cases, wrappers for each bus provide standard application interfaces for using the bus. Sharing the bus amongst different hardware devices or protocols may be done through a bus arbitration component.

Bus arbitration allows higher level services to attain exclusive use of the bus, complete its operations, and then release the bus to the next service:

```
interface BusArbitration {
  async command result_t getBus();
  async command result_t releaseBus();
  event result_t busFree();
}
```

### *F. External storage*

Because the on-chip memory on most of the processors used in the current WSN platforms is severely limited, the existence of a secondary memory storage is very important for supporting a range of applications like simple data logging, file systems, persistent storage of program images, etc.

Flash is typically used to store three classes of data: large objects, such as program images or bulk transfers, small objects, such as metadata or configuration information, and large sequential objects, such as logs. The challenge in creating an abstraction for external storage is interfacing with different flash technologies. Data flash, such as Atmel's AT45DB chips, has small write units–256-byte pages. Conversely, code flash, such as ST's M25P chips, has large write and erase units–64kB to 128kB sectors is the minimum erase unit.

Depending on the targeted applications, this secondary memory usually comes in the form of either EEPROM or flash chips that are interfaced with the microcontroller using some of the previously described data busses. Data buses may serve as the HPL for external storage devices.

The starting point of the HAL layer component can be the execution of an "atomic access" to the chip, i.e. sending a single command, read, write, or erase. Depending on the write unit and erase unit, the HAL

will provide the primitives for that storage technology. If single bytes may be written, but whole sectors must be erased, the HAL will provide the least common denominator by enabling byte writes (as arbitrary-length buffers) and sector erase commands.

These device-specific HAL interfaces are exposed to applications through HIL wrappers. At the HIL level, the flash is segmented into volumes that may be assigned to specific system services. The system services can use the HIL functionality for the three common classes of flash use–large writes, small writes, and large sequential writes.

The configuration, block read, log read, and volume interfaces are as follows:

```
interface FlashVolume {
  command result_t mount(uint8_t uid);
  command uint32_t physicalAddr(uint8_t volumeAddr);
  command uint8_t physicalAddrToVolume(
          uint32_t addr);
}

interface BlockRead {
  command result_t read(block_addr_t addr,
          uint8_t* buf, block_addr_t len);
  event   result_t readDone(result_t result);
  command result_t verify();
  event   result_t verifyDone(result_t result);
  command result_t computeCrc(block_addr_t addr,
          block_addr_t len);
  event result_t computeCrcDone(result_t result,
        uint16_t crc);
}

interface ConfigStorage {
  command result_t read(addr_t addr, void* dest,
          addr_t len);
  event result_t readDone(storage_result_t result);
  command result_t write(addr_t addr, void* source,
          addr_t len);
  event result_t writeDone(storage_result_t result);
  command result_t commit();
  event result_t commitDone(storage_result_t result);
}

interface LogRead {
  command result_t read(uint8_t* data,
          uint32_t numBytes);
  event result_t readDone(uint8_t* data,
        uint32_t numBytes, result_t success);
  command result_t seek(uint32_t cookie);
  event result_t seekDone(storage_result_t success);
}
```

The volume-based system has been proposed to allow hardware independent access to the flash. The arbitration between the potential multiple users of a particular flash volume is performed through the *AllocationReq* interface that reserves parts of the memory to each user at compilation time. Even more powerful is the *Matchbox* and *Elf* systems that could be built above our HIL abstraction providing a simple filing abstraction over the flash.

### *G. Radios*

While hardware module functionality such as ADC, Timer, and data busses has remained fairly consistent, the radio functionality changes more frequently. Changing the radio usually results in improvements in performance or power consumption. Sensor network platforms are frequently modified to reap the benefits of the newest generation transceiver.

For a radio, the HPL includes the basic functionality of that radio abstracted from the particular microcontroller. Unlike microcontrollers, radios have varying data interfaces. Some accept a single bit as input, while others use a data bus. Sensor network radios have included bit, byte, and packet level inputs for data transmission. The radio's basic data unit is exposed directly to the radio stack by the HPL.

For control of the radio, a combination of hardware pins (for transmit/receive mode switching or RSSI data, for example) and a data bus are frequently used. The radio may be controlled through registers and RAM values internal to the radio and set via a data bus. For register-based radios like the Chipcon CC1000, CC2420 or the Infineon TDA5250, registers and RAM may be set or read through the HPL implementation similar to the HPL implementation of a microcontroller hardware module. Pin values, interrupts, and data transmission are also exposed through the HPL. The radio's HPL is implemented on top of the abstractions provided for each microcontroller. Physical layer software logic may then be written once for each radio and run over various hardware. Our abstraction is used by the CC2420 radio stack and runs on the Telos, MicaZ, iMote2, and Chipcon CC2420EB platforms. The abstraction for the Infineon TDA5250 radio also follows the three-layer model and runs on the eyesIFX and eyesIFXv2 platforms.

## IV. Related work

One of the major roles of operating systems is to create a unified abstract computing environment for the application programmer that is independent from the details of the underlying hardware. In order to obtain OS portability, it is useful to separate the parts that directly interact with the hardware from the parts that have general applicability and can be reused over different platforms.

Traditionally, OS hardware transparency has been achieved with two related concepts: an *Abstraction*

*Layer*[1] that deals with the architectural differences of the processing units; and a *Device Driver Model* that deals with the way the system interacts with the other hardware devices.

Although widely used, the realization of these concepts varies significantly among the different operating systems and can lead to different trade-offs between efficient resource use and portability. In the following we present several concrete examples.

The *NetBSD* is claimed to be the most portable of the modern UNIX-like operating systems and currently runs on over 50 different hardware platforms. This remarkable portability in mainly due to the design of the machine-independent driver framework that is based on clean separation between the chipset drivers and the bus attach code. In addition, the access to the bus memory and register areas is implemented in a machine-independent way, allowing the same device driver source to be used on different system architectures and bus types [11]. In a very similar fashion, the new *Linux* device driver model [12] structures the hardware abstraction in terms of buses, classes, devices and drivers.

*eCos* is a component-based RTOS for embedded applications that also takes pride in its portability. Like TinyOS, it uses compile-time reconfiguration to trim the OS to the specific requirements of each application. The hardware abstraction functionality in eCos is organized in two main parts [13]: an "abstraction layer" that provides architecture-independent support for handling interrupts, virtual vectors and exceptions; and a group of "device drivers" that abstract the capabilities of the hardware modules. The drivers are implemented as monolithic components and are accessed by the rest of the system via the "I/O Sub-System" that defines a standard interface for communication with the exposed driver "handlers".

In contrast, the device drivers in the *WindowsCE* embedded operating system from Microsoft can be either monolithic or structured in two customized layers [14]. The upper layer is formed by the platform-independent "Model Device Driver (MDD)" that uses the services of the "Platform-Dependent Driver (PDD)" that forms the lower layer. Although this resembles our HIL/HAL stratification, the layering here is mostly done to separate the regions of responsibility between Microsoft (that provides the MDD specification and implementation) and the OEMs that provide the PDD. In particular, direct access to the lower PDD interface is not allowed and all communication with the hardware has to be performed via the standard "Device Driver Interface (DDI)".

In the above examples, the multiple levels of indirection and the forced use of generic abstractions unavoidably leads to suboptimal utilization of the hardware resources for the platform-specific applications. On general-purpose systems this may well be acceptable, but the resource constrained nature of WSNs requires a more balanced approach.

The problem of decreased efficiency due to unsuitable hardware abstractions has also been subject of interest in the general operating systems research field. The Exokernel architecture [15], for example, proposes a radical change in the organization of the abstractions. According to their framework, the kernel exports all hardware resources through a low-level interface to the user-space, where the application or a library OS can build the optimal abstraction. Because of the simple microcontroller core architecture and the non-existence of a costly system-space/user-space barrier, many of the motivating factors for the Exokernel ideas are not present in our domain. Nevertheless, our intention to expose the *full* capabilities of the hardware at the HAL level and to support *flexible* selection of the most appropriate abstraction level, shares the same spirit as their work.

## V. Future Work

Efficient implementation of the ideas presented in this paper requires support from the underlying programming language. As the interfaces and abstractions evolve, the programming language also evolves to better support these abstractions. TinyOS uses nesC for programming language support. We anticipate evolving our interfaces for the new version of nesC (version 1.2 and later) [16] that introduces *generic interfaces* and *generic components*.

By using generic interfaces and components, we can further simplify the interfaces used in our abstraction. Generic components may take a data type as an argument, similar to templates in C. To illustrate the usefulness of generic interfaces and template-like syntax they provide, we step through the changes to the Timer abstraction.

The Timer is made up of underlying alarms provided by the hardware and exported by the HAL. Alarms provide various resolutions–32kHz, millisecond, and microsecond. In section III-C, we supported different resolutions by using identical Alarm interfaces with different names. Instead, in nesC 1.2+, we support this notion

[1]Usually called the *Hardware Abstraction Layer (HAL)*. Not to be confused with the *Hardware Adaptation Layer (HAL)* in our architecture.

through structs, used with generic interfaces, that define the resolution. This changes our abstraction as follows:

```
typedef struct { } TMilli;
typedef struct { } T32khz;
typedef struct { } TMicro;

typedef struct { } TNano;
//as an example of a future extension

interface Alarm<resolution> {
  async command uint32_t get();
  async command bool isSet();
  async command void cancel();
  async command void set(uint32_t t0, uint32_t dt);
  async event void fired();
}

configuration AlarmC {
  provides interface Alarm<TMilli>
                     as AlarmTimerMilli;
  provides interface Alarm<T32khz>
                     as AlarmTimer32khz;

  provides interface Alarm<T32khz> as Alarm32khz1;
  provides interface Alarm<T32khz> as Alarm32khz2;
  //...

  provides interface Alarm<TMicro> as AlarmMicro1;
  provides interface Alarm<TMicro> as AlarmMicro2;
  provides interface Alarm<TMicro> as AlarmMicro3;
  //...
}
```

Using the idea of generic interfaces and components that are instantiated with a structure, we can write a generic component that converts from one time structure (resolution) to another. Generic components may also be used to implement hardware-independent alarms, stopwatches, and synchronization services. Implementations and examples of these ideas can be found in the *TinyOS Enhancement Proposals* available from the TinyOS web site [17].

Another open area is the method for passing data between nodes with different microcontroller architectures. With the emergence of standardized radios like IEEE 802.15.4 for wireless sensor networks, a family of devices have come into existence that are unable to directly communicate with each other due to incompatibilities in architectures. A traditional view is to use additional storage to marshal and unmarshal all data as it passes to and from the physical layer.

Instead, nesC 1.2 introduces network types [18], which are structures prefixed with a *network* keyword such that they are interpreted by nesC. Whenever an element of a structure is written or read, it is converted to the architecture's native form. This method does not rely on copying messages at the physical layer and instead uses stack space when accessing the struct within an application. We intend to use this method for cross-platform communication in our architecture.

Finally, the architecture presented in this paper serves as the framework adopted for TinyOS 2.0. TinyOS 2.0 is a rewrite of the TinyOS operating system components to support a broader range of platforms. Many of the interfaces discussed in this paper will be used, and iterated upon, during the development of TinyOS 2.0.

## VI. Conclusion

The analysis in Section III shows that the three-layer design can be successfully used for exposing to the applications the functionality of the main hardware modules in different WSN platforms. Our architecture provides a set of core services that eliminate duplicated code and provide a coherent view of the system across different architectures and platforms.

Our architecture supports the concurrent use of platform-independent and the platform-dependent interfaces in the same application. 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.

It is important to stress that the proposed design has been successfully tested in practice on the implementation of TI MSP430 microcontroller port in TinyOS. The resulting MSP430 platform abstraction is part of the main TinyOS distribution starting from the 1.1.7 minor release. More details about the components and their interfaces can be found in the implementation code [19].

Our architecture provides a powerful, yet efficient way to build platform independent services applications, while still permitting direct access to the hardware's features for high performance applications. We have provided a powerful set of abstractions that enable timing, alarms, communication, sampling, storage, and low power operation across different hardware platforms.

## Acknowledgments

This work has been partially supported by the EC under the contract IST-2001-34734 (EYES), the National Science Foundation, and the DARPA "NEST" contract F33615-01-C1895.

The authors wish to thank Kevin Klues for his work on the implementation of the MSP430 and eyesIFX platforms in TinyOS, the members of the TinyOS 2.0 Working Group for their valuable comments and revisions, and the anonymous reviewers for their helpful comments.

## REFERENCES


[1] The eCos operating system home page. [Online]. Available: http://sources.redhat.com/ecos

[2] The WindowsCE operating system home page. [Online]. Available: http://msdn.microsoft.com/embedded/windowsce

[3] The NetBSD project home page. [Online]. Available: http://www.netbsd.org

[4] The Linux kernel archives. [Online]. Available: http://www.kernel.org

[5] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins, "A survey of configurable, component-based operating systems for embedded applications," *IEEE Micro*, vol. 21, no. 3, pp. 54–68, 2001.

[6] J. A. Rowson and A. Sangiovanni-Vincentelli, "Interface-based design," in *Proceedings of the 34th annual conference on Design automation*, 1997, pp. 178–183.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*. ACM Press, 2000, pp. 93–104.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*. ACM Press, 2003, pp. 1–11.

[9] V. Handziski, J. Polastre, J.-H. Hauer, and C. Sharp, "Poster abstract: Flexible hardware abstraction of the ti msp430 microcontroller in tinyos," in *Proceedings of the second international conference on Embedded networked sensor systems (SenSys 2004)*, 2004.

[10] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy, "The platforms enabling wireless sensor networks," *Commun. ACM*, vol. 47, no. 6, pp. 41–46, 2004.

[11] J. R. Thorpe, "A machine-independent dma framework for netbsd," in *Proceedings of USENIX Conference (FREENIX track)*. USENIX Association, 1998.

[12] P. Mochel. (2003) The linux kernel device model. Proceedings of the Linux.Conf.Au conference (LCA 2003). [Online]. Available: http://conf.linux.org.au

[13] A. Massa, *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[14] J. Murray, *Inside Microsoft Windows CE*. Microsoft Press, 1998.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: an operating system architecture for application-level resource management," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*. ACM Press, 1995, pp. 251–266.

[16] D. Gay, P. Levis, D. Culler, and E. Brewer, *nesC 1.2 Language Reference Manual*, July 2004.

[17] The TinyOS community forum. [Online]. Available: http://www.tinyos.net

[18] K. K. Chang and D. Gay, "Language support for messaging in heterogeneous networks," in *Intel Research Technical Report*, Berkeley, CA, Nov. 2004.

[19] The MSP430 platform implementation code. [Online]. Available: http://cvs.sourceforge.net/viewcvs.py/tinyos/tinyos-1.x/tos/platform/msp430


## ACRONYMS

| | |
|---|---|
| ADC | Analog to Digital Converter |
| API | Application Programming Interface |
| GCC | GNU C Compiler |
| DCO | Digitally Controlled Oscillator |
| EEPROM | Electrically Erasable Programmable Read Only Memory |
| HAA | Hardware Abstraction Architecture |
| HAL | Hardware Adaptation Layer |
| HIL | Hardware Interface Layer |
| HPL | Hardware Presentation Layer |
| I/O | Input/Output |
| $I^2C$ | Inter-IC |
| ISR | Interrupt Service Routine |
| MAC | Media Access Control |
| MCU | Micro Controller Unit |
| OEM | Original Equipment Manufacturer |
| OS | Operating System |
| RISC | Reduced Instruction Set Computer |
| RSSI | Receive Signal Strength Indicator |
| RTOS | Real-Time Operating System |
| SPI | Serial Peripheral Interface |
| UART | Universal Asynchronous Receiver Transmitter |
| USART | Universal Synchronous Asynchronous Receiver Transmitter |
| WSN | Wireless Sensor Network |