Analog-to-Digital Converters (ADCs)

TEP:	101
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Jan-Hinrich Hauer, Philip Levis, Vlado Handziski, David Gay

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with [TEP1].

Abstract

This TEP proposes a hardware abstraction for analog-to-digital converters (ADCs) in TinyOS 2.x, which is aligned to the three-layer Hardware Abstraction Architecture (HAA) specified in [TEP2]. It describes some design principles and documents the set of hardware-independent interfaces to an ADC.

1. Introduction

Analog-to-digital converters (ADCs) are devices that convert analog input signals to discrete digital output signals, typically voltage to a binary number. The interested reader can refer to Appendix A for a brief overview of the ADC hardware on some current TinyOS platforms. In earlier versions of TinyOS, the distinction between a sensor and an ADC were blurred: this led components that had nothing to do with an ADC to still resemble one programatically, even though the semantics and forms of operation were completely different. To compensate for the difference non-ADC sensors introduced additional interfaces, such as `ADCError`, that were tightly bound to sensor acquisition but separate in wiring. The separation between the ADC and `ADCError` interface is bug prone and problematic, as is the equation of a sensor and an ADC. TinyOS 2.x separates the structure and interfaces of an ADC from those of sensor drivers (which may be on top of an ADC stack, but this fact is hidden from higher level components). This TEP presents how TinyOS 2.x structures ADC software. [TEP109] (Sensor Boards) shows how a platform can present actual named sensors.

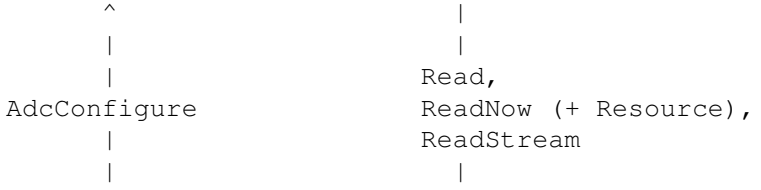
As can be seen in Appendix A the ADC hardware used on TinyOS platforms differ in many respects, which makes it difficult to find a chip independent representa-

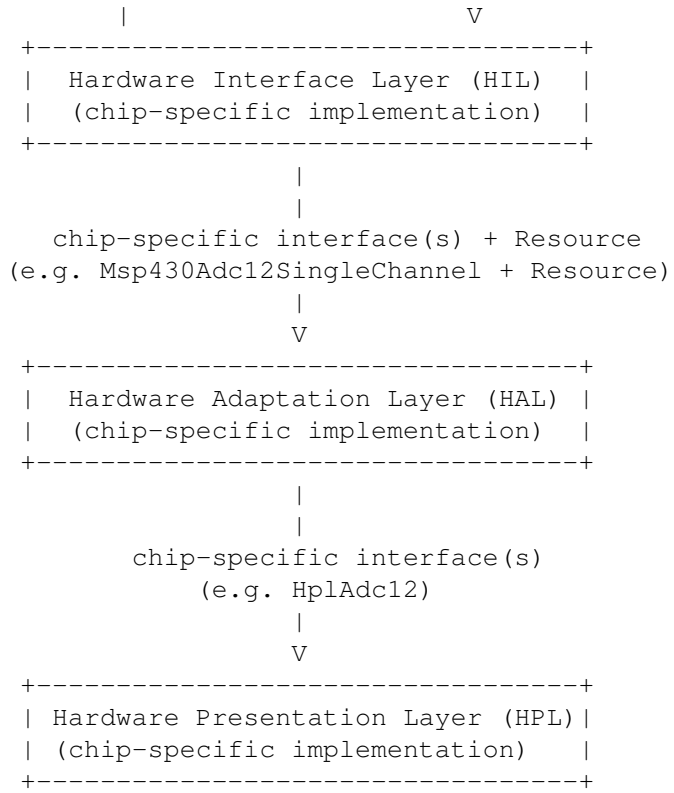
tion for an ADC. Even if there were such a representation, the configuration details of an ADC would still depend on the actual device producing the input signal (sensor). Neither a platform independent application nor the ADC hardware stack itself has access to this information, as it can only be determined on a platform or sensor-board level. For example, determining which ADC port a sensor is attached to and how conversion results need to be interpreted is a platform specific determination. Although the actual configuration details may be different the procedure of configuring an ADC can be unified on all ADCs with the help of **hardware independent interfaces**: in a similar way as the `Read` interface definition does not predefine the type or semantics of the exchanged data (see [TEP114]), a configuration interface definition can abstract from the data type and semantics of the involved configuration settings. For example, like a component can provide a `Read<uint8_t>` or `Read<uint16_t>` interface, it can also provide a `AdcConfigure<atm128_adc_config_t>` or `AdcConfigure<msp430adc12_channel_config_t>` interface depending on what ADC it represents. This TEP proposes the (typed) `AdcConfigure` interface as the standard interface for configuring an ADC in TinyOS 2.x.

In spite of their hardware differences, one aspect represents a common denominator of ADCs: they all produce conversion results. To facilitate sensor software development conversion results are returned by the ADC stack through the interfaces `Read`, `ReadStream` and `ReadNow` (see 2. Interfaces and [TEP114]). Conversion results are returned as uninterpreted values and translating them to engineering units can only be done with the configuration knowledge of the respective platform, for example, the reference voltage or the resistance of a reference resistor in ratiometric measurements. Translating uninterpreted values to engineering units may be performed by components located on top of the ADC stack and is out of the scope of this TEP.

The top layer of abstraction of an ADC - the Hardware Interface Layer (HIL) - thus provides the interfaces `Read`, `ReadNow` and `ReadStream` and uses the `AdcConfigure` interface for hardware configuration (why it **uses** and does not **provide** `AdcConfigure` is explained below). Since the type and semantics of the parameters passed through these interfaces is dependent on the actual ADC implementation, it is only a "weak" HIL (see [TEP2]).

Following the principles of the HAA [TEP2] the Hardware Adaptation Layer (HAL, which resides below the HIL) of an ADC should expose all the chip-specific capabilities of the chip. For example, the ADC12 on the MSP430 MCU supports a "Repeat-Sequence-of-Channels Mode" and therefore this function should be accessible on the HAL of the MSP430 ADC12 hardware abstraction. Other ADCs might not exhibit such functionality and might therefore - on the level of HAL - provide only an interface to perform single conversions. Since all ADCs have the same HIL representation it may be necessary to perform some degree of software emulation in the HIL implementation. For example, a `ReadStream` command can be emulated by multiple single conversion commands. Below the HAL resides the Hardware Presentation Layer (HPL), a stateless component that provides access to the hardware registers (see [TEP2]). The general structure (without virtualization) of the ADC stack is as follows





The rest of this TEP specifies:

- the set of standard TinyOS interfaces for collecting ADC conversion results and for configuring an ADC (2. Interfaces)
- guidelines on how an ADC's HAL should expose chip-specific interfaces (3. HAL guidelines)
- what components an ADC's HIL MUST implement (4. HIL requirements)
- guidelines on how the HIL should be implemented (5. HIL guidelines)
- a section pointing to current implementations (6. Implementation)

This TEP ends with appendices documenting, as an example, the ADC implementation for the TI MSP430 MCU.

2. Interfaces

This TEP proposes the `AdcConfigure` interface for ADC hardware configuration and the `Read`, `ReadStream` and `ReadNow` interfaces to acquire conversion results. The `Read` and `ReadStream` interfaces are documented in [TEP114] and the `ReadNow` interface is documented in this TEP. A `Read[Now|Stream]` interface is always provided in conjunction with a `AdcConfigure` interface.

Interface for configuring the ADC hardware

The `AdcConfigure` interface is defined as follows:

```
interface AdcConfigure< config_type >
{
  async command config_type getConfiguration();
}
```

This interface is used by the ADC stack to retrieve the hardware configuration of an ADC HIL client. `config_type` is a chip-specific data type (simple or structured) that contains all information necessary to configure the respective ADC hardware. For example, on the ADC12 of the MSP430 the `AdcConfigure` interface will be instantiated with the `const msp430adc12_channel_config_t*` data type. A client MUST always return the same configuration through a `AdcConfigure` interface and, if configuration data is passed as a pointer, the HIL component (see 4. HIL requirements) MUST NOT reference it after the return of the `getConfiguration()` command. If a client wants to use the ADC with different configurations it must provide multiple instances of the `AdcConfigure` interface.

Note that the `AdcConfigure` interface is **provided** by an ADC HIL client and it is **used** by the ADC HIL implementation. Therefore an ADC HIL client cannot initiate the configuration of the ADC hardware itself. Instead it is the ADC HIL implementation that can "pull" the client's ADC configuration just before it initates a conversion based on the respective client's configuration. The rationale is that the ADC HIL implementation does not have to store an ADC configuration per client - instead the ADC client can, for example, store its configuration in program memory.

Interfaces for acquiring conversion results

This TEP proposes to adopt the following two source-independent data collection interfaces from [TEP114] for the collection of ADC conversion results on the level of HIL:

```
interface Read< size_type >
interface ReadStream< size_type >
```

In addition it proposes the following data collection interface for low-latency reading of conversion results:

```
interface ReadNow< size_type >
```

Every data collection interface is associated with an `AdcConfigure` interface (how this association is realized is explained in Section 4. HIL requirements). As the resolution of conversion results is chip-specific, the `size_type` parameter reflects an upper bound for the chip-specific resolution of the conversion results - the actual resolution may be smaller (e.g. uint16_t for a 12-bit ADC).

Read

The `Read` interface can be used to sample an ADC channel once and return a single conversion result as an uninterpreted value. The `Read` interface is documented in [TEP114].

ReadStream

The `ReadStream` interface can be used to sample an ADC channel multiple times with a specified sampling period. The `ReadStream` interface is documented in [TEP114] .

ReadNow

The `ReadNow` interface is intended for split-phase low-latency reading of small values:

```
interface ReadNow<val_t>
{
  async command error_t read();
  async event void readDone( error_t result, val_t val );
}
```

This interface is similar to the `Read` interface, but works in asynchronous context. A successful call to `ReadNow.read()` means that the ADC hardware has started the sampling process and that `ReadNow.readDone()` will be signalled once it has finished (note that the asynchronous `ReadNow.readDone()` might be signalled even before the call to `ReadNow.read()` has returned). Due to its timing constraints the `ReadNow` interface is always provided in conjunction with an instance of the `Resource` interface and a client must reserve the ADC through the `Resource` interface before the client may call `ReadNow.read()`. Please refer to [TEP108] on how the `Resource` interface should be used by a client component.

3. HAL guidelines

As explained in 1. Introduction the HAL exposes the full capabilities of the ADC hardware. Therefore only chip- and platform-dependent clients can wire to the HAL. Although the HAL is chip-specific, both, in terms of implementation and representation, its design should follow the guidelines described in this section to facilitate the mapping to the HIL representation. Appendix B shows the signature of the HAL for the MSP430.

Resource reservation

As the ADC hardware is a shared resource that is usually multiplexed between several clients some form of access arbitration is necessary. The HAL should therefore provide a parameterized `Resource` interface, instantiate a standard arbiter component and connect the `Resource` interface to the arbiter as described in [TEP108]. To ensure fair and uniform arbitration on all platforms the standard round robin arbiter is recommended. Resource arbiters and the `Resource` interface are the topic of [TEP108].

Configuration and sampling

As the ADC hardware is a shared resource the HAL should support hardware configuration and sampling per client (although per-port configuration is possible, it is not recommended, because it forces all clients to use the same configuration for a given

port). Therefore the HAL should provide sampling interfaces parameterized by a client identifier. A HAL client can use its instance of the sampling interface to configure the ADC hardware, start the sampling process and acquire conversion results. It wires to a sampling interface using a unique client identifier (this may be hidden by a virtualization component). All commands and events in the sampling interface should be 'async' to reflect the potential timing requirements of clients on the level of HAL. A HAL may provide multiple different parameterized sampling interfaces, depending on the hardware capabilities. This allows to differentiate/group ADC functionality, for example single vs. repeated sampling, single channel vs. multiple channels or low-frequency vs. high-frequency sampling. Every sampling interface should allow the client to individually configure the ADC hardware, for example by including the configuration data as parameters in the sampling commands. However, if configuration data is passed as a pointer, the HAL component MUST NOT reference it after the return of the respective command. Appendix B shows the HAL interfaces for the MSP430.

HAL virtualization

In order to hide wiring complexities and/or export only a subset of all ADC functions generic ADC wrapper components may be provided on the level of HAL. Such components can also be used to ensure that a sampling interface is always provided with a `Resource` interface and both are instantiated with the same client ID if this is required by the HAL implementation.

4. HIL requirements

The following generic components MUST be provided on all platforms that have an ADC:

```
AdcReadClientC
AdcReadNowClientC
AdcReadStreamClientC
```

These components provide virtualized access to the HIL of an ADC. They are instantiated by an ADC client and provide/use the four interfaces described in Section 2. Interfaces. An ADC client may instantiate multiple such components. The following paragraphs describe their signatures. Note that this TEP does not address the issue of how to deal with multiple ADCs on the same platform (the question of how to deal with multiple devices of the same class is a general one in TinyOS 2.x). Appendix C shows the `AdcReadClientC` for the MSP430.

AdcReadClientC

```
generic configuration AdcReadClientC() {
  provides {
    interface Read< size_type >;
  }
  uses {
    interface AdcConfigure< config_type >;
  }
}
```

The AdcReadClientC component provides a Read interface for acquiring single conversion results. The associated ADC channel (port) and further configuration details are returned by the AdcConfigure.getConfiguration() command. It is the task of the client to wire this interface to a component that provides the client's ADC configuration. The HIL implementation will use the AdcConfigure interface to dynamically "pull" the client's ADC settings when it translates the Read.read() command to a chip-specific sampling command. Note that both, size_type and config_type, are only placeholders and will be instantiated by the respective HIL implementation (for an example, see the AdcReadClientC for the MSP430 in Appendix C).

AdcReadNowClientC

```
generic configuration AdcReadNowClientC() {
  provides {
    interface Resource;
    interface ReadNow< size_type >;
  }
  uses {
    interface AdcConfigure< config_type >;
  }
}
```

The AdcReadNowClientC component provides a ReadNow interface for acquiring single conversion results. In contrast to Read.read() when a call to ReadNow.read() succeeds, the ADC starts to sample the channel immediately (a successful Read.read() command may not have this implication, see [TEP114] and 2. Interfaces). A client MUST reserve the ADC through the Resource interface before the client may call ReadNow.read() and it MUST release the ADC through the Resource interface when it no longer needs to access it (for more details on how to use the Resource interface please refer to [TEP108]). The associated ADC channel (port) and further configuration details are returned by the AdcConfigure.getConfiguration() command. It is the task of the client to wire this interface to a component that provides the client's ADC configuration. The HIL implementation will use the AdcConfigure interface to dynamically "pull" the client's ADC settings when it translates the ReadNow.read() command to a chip-specific sampling command. Note that both, size_type and config_type, are only placeholders and will be instantiated by the respective HIL implementation (for an example how this is done for the AdcReadClientC see Appendix C).

AdcReadStreamClientC

```
generic configuration AdcReadStreamClientC() {
  provides {
    interface ReadStream< size_type >;
  }
  uses {
    interface AdcConfigure< config_type>;
  }
}
```

The AdcReadStreamClientC component provides a ReadStream interface for acquiring multiple conversion results at once. The ReadStream interface is explained in [TEP114] and 2. Interfaces. The AdcConfigure interface is used in the same way as described in the section on the AdcReadClientC. Note that both, size_type and config_type, are only placeholders and will be instantiated by the respective HIL implementation (for an example how this is done for the AdcReadClientC see Appendix C).

5. HIL guidelines

The HIL implementation of an ADC stack has two main tasks: it translates a Read, ReadNow or ReadStream request to a chip-specific HAL sampling command and it abstracts from the Resource interface (the latter only for the AdcReadClientC and AdcReadStreamClientC). The first task is solved with the help of the AdcConfigure interface which is used by the HIL implementation to retrieve a client's ADC configuration. The second task MAY be performed by the following library components: ArbitratedReadC, and ArbitratedReadStreamC (in tinyos-2.x/tos/system) - please refer to the Atmel Atmega 128 HAL implementation (in tinyos-2.x/tos/chips/atm128/adc) for an example. Note that since the ReadNow interface is always provided in conjunction with a Resource interface the HIL implementation does not have to perform the ADC resource reservation for an AdcReadNowClientC, but may simply forward an instance of the Resource interface from the HAL to the AdcReadNowClientC.

The typical sequence of events is as follows: when a client requests data through the Read or ReadStream interface the HIL will request access to the HAL using the Resource interface. After the HIL has been granted access, it will "pull" the client's ADC configuration using the AdcConfigure interface and translate the client's Read or ReadStream command to a chip-specific HAL command. Once the HIL is signalled the conversion result(s) from the HAL it releases the ADC through the Resource interface and signals the conversion result(s) to the client though the Read or ReadStream interface. When a client requests data through the ReadNow interface the HIL translates the client's command to the chip-specific HAL command without using the Resource interface (it may check ownership of the client through the ArbiterInfo interface - this check can also be done in the HAL implementation). Once the HIL is signalled the conversion result(s) it forwards it to the respective ReadNow client.

6. Implementation

TestAdc application

An ADC HIL test application can be found in tinyos-2.x/apps/tests/TestAdc. Note that this application instantiates generic DemoSensorC, DemoSensorStreamC and DemoSensorNowC components (see [TEP114]) and assumes that these components are actually wired to an ADC HIL. Please refer to tinyos-2.x/apps/tests/TestAdc/README.txt for more information.

HAA on the MSP430 and Atmega 128

The implementation of the ADC12 stack on the MSP430 can be found in tinyos-2.x/tos/chips/msp430/adc

- `HplAdc12P.nc` is the HPL implementation
- `Msp430Adc12P.nc` is the HAL implementation
- `AdcP.nc` is the HIL implementation
- `AdcReadClientC.nc`, `AdcReadNowClientC.nc` and `AdcReadStreamClientC.nc` provide virtualized access to the HIL
- the use of DMA or the reference voltage generator and the HAL virtualization components are explained in `README.txt`

The Atmel Atmega 128 ADC implementation can be found in `tinyos-2.x/tos/chips/atm128/adc`:

- `HplAtm128AdcC.nc` is the HPL implementation
- `Atm128AdcP.nc` is the HAL implementation
- `AdcP.nc`, `WireAdcP.nc` and the library components for arbitrating 'Read', 'ReadNow' and 'ReadStream', `ArbitratedReadC` and `ArbitratedReadStreamC` (in `tinyos-2.x/tos/system`), realize the HIL
- `AdcReadClientC.nc`, `AdcReadNowClientC.nc` and `AdcReadStreamClientC.nc` provide virtualized access to the HIL

Appendix A: Hardware differences between platforms

The following table compares the characteristics of two microcontrollers commonly used in TinyOS platforms:

	Atmel Atmega 128	**TI MSP430 ADC12**
Resolution	10-bit	12-bit
channels	• 8 multiplexed external channels • 16 differential voltage input combinations • 2 differential inputs with gain amplification	• 8 individually configurable external channels • internal channels (AVcc, temperature, reference voltages)
internal reference voltage	2.56V	1.5V or 2.5V

... continued on next page

	Atmel Atmega 128	TI MSP430 ADC12
conversion reference	• positive terminal: AVcc or 2.56V or AREF (external) • negative terminal: GND	individually selectable per channel: • AVcc and AVss • Vref+ and AVss • Veref+ and AVss • AVcc and (Vref- or Veref-) • AVref+ and (Vref- or Veref-) • Veref+ and (Vref- or Veref-)
conversion modes	• single channel conversion mode • free running mode (channels and reference voltages can be switched between samples)	• single conversion mode • repeat single conversion mode • sequence mode (sequence <= 16 channels) • repeat sequence mode
conversion clock source	clkADC with prescaler	ACLK, MCLK, SMCLK or ADC-oscillator (5MHz) with prescaler respectively
sample-hold-time	1.5 clock cycles (fixed)	selectable values from 4 to 1024 clock cycles

... continued on next page

	Atmel Atmega 128	TI MSP430 ADC12
conversion triggering	by software	by software or timers
conversion during sleep mode possible	yes	yes
interrupts	after each conversion	after single or sequence conversion

Appendix B: a HAL representation: MSP430 ADC12

This section shows the HAL signature for the ADC12 of the TI MSP430 MCU. It reflects the four MSP430 ADC12 conversion modes as it lets a client sample an ADC channel once ("Single-channel-single-conversion") or repeatedly ("Repeat-single-channel"), multiple times ("Sequence-of-channels") or multiple times repeatedly ("Repeat-sequence-of-channels"). In contrast to the single channel conversion modes the sequence conversion modes trigger a single interrupt after multiple samples and thus enable high-frequency sampling. The `DMAExtension` interface is used to reset the state machine when the DMA is responsible for data transfer (managed in an exterior component):

```
configuration Msp430Adc12P
{
  provides {
    interface Resource[uint8_t id];
    interface Msp430Adc12SingleChannel as SingleChannel[uint8_t id];
    interface AsyncStdControl as DMAExtension[uint8_t id];
  }
}

interface Msp430Adc12SingleChannel
{
  async command error_t configureSingle(const msp430adc12_channel_config_t
  async command error_t configureSingleRepeat(const msp430adc12_channel_cor
  async command error_t configureMultiple( const msp430adc12_channel_config
  async command error_t configureMultipleRepeat(const msp430adc12_channel_c
  async command error_t getData();
  async event error_t singleDataReady(uint16_t data);
  async event uint16_t* multipleDataReady(uint16_t buffer[], uint16_t numSa
}

typedef struct
{
  unsigned int inch: 4;            // input channel
  unsigned int sref: 3;            // reference voltage
  unsigned int ref2_5v: 1;         // reference voltage level
  unsigned int adc12ssel: 2;       // clock source sample-hold-time
  unsigned int adc12div: 3;        // clock divider sample-hold-time
  unsigned int sht: 4;             // sample-hold-time
  unsigned int sampcon_ssel: 2;    // clock source sampcon signal
  unsigned int sampcon_id: 2;      // clock divider sampcon signal
} msp430adc12_channel_config_t;
```

Appendix C: a HIL representation: MSP430 ADC12

The signature of the AdcReadClientC component for the MSP430 ADC12 is as follows:

```
generic configuration AdcReadClientC() {
  provides interface Read<uint16_t>;
  uses interface AdcConfigure<const msp430adc12_channel_config_t*>;
}
```

[TEP1] TEP 1: TEP Structure and Keywords.

[TEP2] TEP 2: Hardware Abstraction Architecture.

[TEP108] TEP 108: Resource Arbitration.

[TEP109] TEP 109: Sensor Boards.

[TEP114] TEP 114: SIDs: Source and Sink Independent Drivers.