Timers

TEP:	102
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Cory Sharp, Martin Turon, 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 TEP 1.

Abstract

This TEP proposes a Timer design that supports common timing requirements both in precision and width across common hardware configurations. This TEP focuses on aligning the Timer abstraction with the three-layer Hardware Abstraction Architecture (HAA).

1. Introduction

Most microcontrollers offer a rich timer system, with features like:

- several counters, possibly of different widths, with multiple clocking options
- one or more compare registers for each counter, which can trigger interrupts, changes to output pins and changes to the counter value
- capture of the time of input pin changes

The interested reader can refer to Appendix A for a brief overview of the timer hardware on some current TinyOS platforms.

TinyOS does not attempt to capture all this diversity in a platform-independent fashion. Instead, following the principles of the HAA[_tep2], each microcontroller should expose all this functionality via components and interfaces at the HPL and, where appropriate, HAL levels. However, two aspects of timers are sufficiently common and important that they should be made available in a well-defined way: measuring time, and triggering (possibly repeating) events at specific times. The rest of this TEP specifies:

- a set of platform-independent interfaces for counting time and triggering events (2. Interfaces)

- guidelines on how each microcontroller's HAL SHOULD expose its timer hardware in terms of the above interfaces (3. HAL guidelines)
- what components a microcontroller's timer HIL MUST implement (4. HIL requirements)
- a set of utility components whose use simplifies building the components specified by the HAL guidelines and HIL requirements (5. Utility components)

This TEP ends with appendices documenting, as an example, the mica2 timer subsystem implementation.

2. Interfaces

Before presenting the interfaces (2.2), we start with a general discussion of the issues of precision, width and accuracy in timer interfaces (2.1).

2.1 Precision, Width and Accuracy

Three fundamental properties of timers are *precision*, *width* and *accuracy*.

Examples of precision are millisecond, a cycle of a 32kHz clock, and microseconds. All precisions presented in this TEP are in "binary" units with respect to one second. That is, one second contains 1024 binary milliseconds, 32768 32kHz ticks, or 1048576 microseconds. This TEP emphasizes millisecond and 32kHz tick precisions while reasonably accommodating other precisions. The use of "binary" units is motivated by the common availability of hardware clocks driven by a 32768Hz crystal.

Examples of widths are 8-bit, 16-bit, 32-bit, and 64-bit. The width for timer interfaces and components SHOULD be 32-bits. This TEP emphasizes 32-bit widths while reasonably accommodating other widths - a particular platform may have good reasons not to expose a 32-bit interface.

Accuracy reflects how closely a component conforms to the precision it claims to provide. Accuracy is affected by issues such as clock drift (much higher for internal vs crystal oscillators) and hardware limitations. As an example of hardware limitations, a mica2 clocked at 7.37MHz cannot offer an exact binary microsecond timer -- the closest it can come is 7.37MHz/8. Rather than introduce a plethora of precisions, we believe it is often best to pick the existing precision closest to what can be provided, along with appropriate documentation. However, the accuracy MUST remain reasonable: for instance, it would be inappropriate to claim that a millisecond timer is a 32kHz timer.

This TEP parameterizes all interfaces by precision and some interfaces by width. This intentionally makes similar timer interfaces with different precision or width mutually incompatible. It also allows user code to clearly express and understand the precision and width for a given timer interface. Accuracy is not reflected in the interface type.

Precision is expressed as a dummy type -- TMilli, T32khz, and TMicro -- written in the standard Timer.h header like this:

```
typedef struct { int notUsed; } TMilli; // 1024 ticks per second
typedef struct { int notUsed; } T32khz; // 32768 ticks per second
typedef struct { int notUsed; } TMicro; // 1048576 ticks per second
```

Note that the precision names are expressed as either frequency or period, whichever is convenient.

2.2 Timer interfaces

This TEP proposes these timer interfaces:

```
interface Counter< precision_tag, size_type >
interface Alarm< precision_tag, size_type >
interface BusyWait< precision_tag, size_type >
interface LocalTime< precision_tag >
interface Timer< precision_tag >
```

The LocalTime and Timer interfaces are used primarily by user applications and use a fixed width of 32-bits. The Alarm, BusyWait, and Counter interfaces are used by the TinyOS timer system and advanced user components.

Counter

The Counter interface returns the current time and provides commands and an event for managing overflow conditions. These overflow commands and events are necessary for properly deriving larger width Counters from smaller widths.

```
interface Counter<precision_tag,size_type>
{
  async command size_type get();
  async command bool isOverflowPending();
  async command void clearOverflow();
  async event void overflow();
}
```

get() return the current time.

isOverflowPending() return TRUE if the overflow flag is set for this counter, i.e., if and only if an overflow event will occur after the outermost atomic block exits. Return FALSE otherwise. This command only returns the state of the overflow flag and causes no side effect.

clearOverflow() cancel the pending overflow event clearing the overflow flag.

overflow() signals that an overflow in the current time. That is, the current time has wrapped around from its maximum value to zero.

Alarm

Alarm components are extensions of Counters that signal an event when their compare register detects the alarm time has been hit. All commands and events of the Alarm interface are asynchronous (or in "interrupt context"). The Alarm interface provides a set of "basic" commands for common usage and provides a set of "extended" commands for advanced use.

```
interface Alarm<precision_tag,size_type>
{
  // basic interface
  async command void start( size_type dt );
  async command void stop();
  async event void fired();

  // extended interface
  async command bool isRunning();
  async command void startAt( size_type t0, size_type dt );
  async command size_type getNow();
  async command size_type getAlarm();
}
```

start(dt) cancel any previously running alarm and set to fire in dt time units from the time of invocation. The alarm will only fire once then stop.

stop() cancel any previously running alarm.

fired() signals that the alarm has expired.

isRunning() return TRUE if the alarm has been started and has not been cancelled or has not yet fired. FALSE is returned otherwise.

startAt(t0,dt)

cancel any previously running alarm and set to fire at time t1 = t0+dt. This form allows a delay to be anchored to some time t0 taken before the invocation of startAt. The timer subsystem uses this form internally, to be able to use of the full width of an alarm while also detecting when a short alarm elapses prematurely.

The time `t0` is always assumed to be in the past. A value of `t0` numerically greater than the current time (returned by `getNow()`) represents a time from before the last wraparound.

getNow() return the current time in the precision and width of the alarm.

getAlarm() return the time the currently running alarm will fire or the time that the previously running alarm was set to fire. getAlarm can be used with startAt to set an alarm from the previous alarm time, as in startAt(getAlarm(),dt). This pattern is used within the fired() event to construct periodic alarms.

BusyWait

The BusyWait interface allows for very short synchronous delays. BusyWait should be used sparingly and when an Alarm would not be reasonably efficient or accurate. The BusyWait interface replaces the TOSH_uwait macro from TinyOS 1.x.

BusyWait blocks for no less than the specified amount of time. No explicit upper bound is imposed on the enacted delay, though it is expected that the underlying implementation spins in a busy loop until the specified amount of time has elapsed.

```
interface BusyWait<precision_tag,size_type>
{
  async command void wait( size_type dt );
}
```

wait(dt) block until at least dt time units have elapsed

LocalTime

The LocalTime interface exposes a 32-bit counter without overflow utilities. This is primarily for application code that does not care about overflow conditions.

```
interface LocalTime<precision_tag>
{
  async command uint32_t get();
}
```

get() return the current time.

Timer

All commands and events of the Timer interface are synchronous (or in "task context"). The Timer interface provides a set of "basic" commands for common usage and provides a set of "extended" commands for advanced use. The Timer interface allows for periodic events.

```
interface Timer<precision_tag>
{
  // basic interface
  command void startPeriodic( uint32_t dt );
  command void startOneShot( uint32_t dt );
  command void stop();
  event void fired();

  // extended interface
  command bool isRunning();
  command bool isOneShot();
  command void startPeriodicAt( uint32_t t0, uint32_t dt );
  command void startOneShotAt( uint32_t t0, uint32_t dt );
  command uint32_t getNow();
  command uint32_t gett0();
  command uint32_t getdt();
}
```

startPeriodic(dt) cancel any previously running timer and set to fire in dt time units from the time of invocation. The timer will fire periodically every dt time units until stopped.

startOneShot(dt) cancel any previously running timer and set to fire in dt time units from the time of invocation. The timer will only fire once then stop.

stop() cancel any previously running timer.

fired() signals that the timer has expired (one-shot) or repeated (periodic).

isRunning() return TRUE if the timer has been started and has not been cancelled and has not fired for the case of one-shot timers. Once a periodic timer is started, isRunning will return TRUE until it is cancelled.

isOneShot() return TRUE if the timer is a one-shot timer. Return FALSE otherwise if the timer is a periodic timer.

startPeriodicAt(t0,dt) cancel any previously running timer and set to fire at time t1 = t0+dt. The timer will fire periodically every dt time units until stopped.

As with alarms, the time `t0` is always assumed to be in the past. A value of `t0` numerically greater than the current time (returned by `getNow()`) represents a time from before the last wraparound.

startOneShotAt(t0,dt) cancel any previously running timer and set to fire at time t1 = t0+dt. The timer will fire once then stop.

`t0` is as in `startPeriodicAt`.

getNow() return the current time in the precision and width of the timer.

gett0() return the time anchor for the previously started timer or the time of the previous event for periodic timers.

getdt() return the delay or period for the previously started timer.

3. HAL guidelines

Platforms SHOULD expose their relevant timing capabilities using standard Alarm and Counter interfaces. The design pattern presented here defines a component naming convention to allow platform independent access to particular Alarms and Counters if they exist and to cause compile errors if they do not.

A platform specific hardware timer with precision ${P} and width ${W} SHOULD be exposed as these two conventional Counter and Alarm components:

```
configuration Counter${P}${W}C
{
  provides interface Counter< T${P}, uint${W}_t >;
}

generic configuration Alarm${P}${W}C()
{
  provides interface Alarm< T${P}, uint${W}_t >;
}
```

Instantiating an Alarm${P}${W}C component provides a new and independent Alarm. If the platform presents a limited number of Alarm resources, then allocating more Alarms in an application than are available for the platform SHOULD produce a compile-time error. See Appendices B and C for an example of how to make allocatable Alarms that are each implemented on independent hardware timers.

For example, if a platform has an 8-bit 32kHz counter and three 8-bit 32kHz alarms, then the Counter and Alarm interfaces for ${P}=32khz and ${W}=16 are:

```
configuration Counter32khz8C
{
  provides interface Counter< T32khz, uint8_t >;
}

generic configuration Alarm32khz8C()
{
  provides interface Alarm< T32khz, uint8_t >;
}
```

This pattern MAY be used to define components for the platform that are mutually incompatible in a single application. Incompatible components SHOULD produce compile-time errors when compiled together.

4. HIL requirements

The following component MUST be provided on all platforms

```
HilTimerMilliC
BusyWaitMicroC
```

Both of these components use "binary" units, i.e., 1/1024s for HilTimerMilliC and 1/1048576s for BusyWaitMicroC. Components using other precisions (e.g., regular, non-binary milliseconds) MAY also be provided.

HilTimerMilliC

```
configuration HilTimerMilliC
{
  provides interface Init;
  provides interface Timer<TMilli> as TimerMilli[ uint8_t num ];
  provides interface LocalTime<TMilli>;
}
```

A new timer is allocated using unique(UQ_TIMER_MILLI) to obtain a new unique timer number. This timer number is used to index the TimerMilli parameterised interface. UQ_TIMER_MILLI is defined in Timer.h. HilTimerMilliC is used by the LocalTimeMilliC component and the TimerMilliC generic component, both found in `tos/system/`

BusyWaitMicroC

```
configuration BusyWaitMicroC
{
  provides interface BusyWait<TMicro,uint16_t>;
}
```

BusyWaitMicroC allows applications to busy-wait for a number of microseconds. Its use should be restricted to situations where the delay is small and setting a timer or alarm would be impractical, inefficient or insufficiently precise.

5. Utility components

A number of platform independent generic components are provided to help implementers and advanced users of the TinyOS timer system:

- AlarmToTimerC
- BusyWaitCounterC
- CounterToLocalTimeC
- TransformAlarmC
- TransformCounterC
- VirtualizeTimerC

Appendices B and C show how these can be used to help implement the timer HAL and HIL.

AlarmToTimerC

AlarmToTimerC converts a 32-bit Alarm to a Timer.

```
generic component AlarmToTimerC( typedef precision_tag )
{
  provides interface Timer<precision_tag>;
  uses interface Alarm<precision_tag,uint32_t>;
}
```

BusyWaitCounterC

BusyWaitCounterC uses a Counter to block until a specified amount of time elapses.

```
generic component BusyWaitC( typedef precision_tag,
  typedef size_type @integer() )
{
  provides interface BusyWait<precision_tag,size_type>;
  uses interface Counter<precision_tag,size_type>;
}
```

CounterToLocalTimeC

CounterToLocalTimeC converts from a 32-bit Counter to LocalTime.

```
generic component CounterToLocalTimeC( precision_tag )
{
  provides interface LocalTime<precision_tag>;
  uses interface Counter<precision_tag,uint32_t>;
}
```

TransformAlarmC

TransformAlarmC decreases precision and/or widens an Alarm. An already widened Counter component is used to help.

```
generic component TransformAlarmC(
  typedef to_precision_tag,
  typedef to_size_type @integer(),
  typedef from_precision_tag,
  typedef from_size_type @integer(),
  uint8_t bit_shift_right )
{
  provides interface Alarm<to_precision_tag,to_size_type> as Alarm;
  uses interface Counter<to_precision_tag,to_size_type> as Counter;
  uses interface Alarm<from_precision_tag,from_size_type> as AlarmFrom;
}
```

to_precision_tag and to_size_type describe the final precision and final width for the provided Alarm. from_precision_tag and from_size_type describe the precision and width for the source AlarmFrom. bit_shift_right describes the bit-shift necessary to convert from the used precision to the provided precision.

For instance to convert from an Alarm<T32khz,uint16_t> to an Alarm<TMilli,uint32_t>, the following TransformAlarmC would be created:

```
new TransformAlarmC( TMilli, uint32_t, T32khz, uint16_t, 5 )
```

It is the exclusive responsibility of the developer using TransformAlarmC to ensure that all five of its arguments are self consistent. No compile errors are generated if the parameters passed to TransformAlarmC are inconsistent.

TransformCounterC

TransformCounterC decreases precision and/or widens a Counter.

```
generic component TransformCounterC(
  typedef to_precision_tag,
  typedef to_size_type @integer(),
  typedef from_precision_tag,
  typedef from_size_type @integer(),
  uint8_t bit_shift_right,
  typedef upper_count_type @integer() )
{
  provides interface Counter<to_precision_tag,to_size_type> as Counter;
  uses interface Counter<from_precision_tag,from_size_type> as CounterFrom;
}
```

to_precision_tag and to_size_type describe the final precision and final width for the provided Counter. from_precision_tag and from_size_type describe the precision and width for the source CounterFrom. bit_shift_right describes the bit-shift necessary to convert from the used precision to the provided precision. upper_count_type describes the numeric type used to store the additional counter bits. upper_count_type MUST be a type with width greater than or equal to the additional bits in to_size_type plus bit_shift_right.

For instance to convert from a Counter<T32khz,uint16_t> to a Counter<TMilli,uint32_t>, the following TransformCounterC would be created:

```
new TransformCounterC( TMilli, uint32_t, T32khz, uint16_t, 5, uint32_t )
```

VirtualizeTimerC

VirtualizeTimerC uses a single Timer to create up to 255 virtual timers.

```
generic component VirtualizeTimerC( typedef precision_tag, int max_timers )
{
  provides interface Init;
  provides interface Timer<precision_tag> as Timer[ uint8_t num ];
  uses interface Timer<precision_tag> as TimerFrom;
}
```

6. Implementation

The definition of the HIL interfaces are found in `tinyos-2.x/tos/lib/timer`:

- `Alarm.nc`
- `BusyWait.nc`
- `Counter.nc`
- `LocalTime.nc`
- `Timer.h` defines precision tags and strings for unique()
- `Timer.nc`

The implementation of the utility components are also found in `tinyos-2.x/tos/lib/timer`:

- `AlarmToTimerC.nc`
- `BusyWaitCounterC.nc`
- `CounterToLocalTimeC.nc`
- `TransformAlarmC.nc`
- `TransformCounterC.nc`
- `VirtualizeAlarmC.nc`
- `VirtualizeTimerC.nc`

The implementation of timers for the MSP430 is in `tinyos-2.x/tos/chips/msp430/timer`:

- `Alarm32khz16C.nc` is generic and provides a new `Alarm<T32khz,uint16_t>`
- `Alarm32khz32C.nc` is generic and provides a new `Alarm<T32khz,uint32_t>`
- `AlarmMilli16C.nc` is generic and provides a new `Alarm<TMilli,uint16_t>`
- `AlarmMilli32C.nc` is generic and provides a new `Alarm<TMilli,uint32_t>`
- `BusyWait32khzC.nc` provides `BusyWait<T32khz,uint16_t>`
- `BusyWaitMicroC.nc` provides `BusyWait<TMicro,uint16_t>`

- Counter32khz16C.nc provides Counter<T32khz,uint16_t>
- Counter32khz32C.nc provides Counter<T32khz,uint32_t>
- CounterMilli16C.nc provides Counter<TMilli,uint16_t>
- CounterMilli32C.nc provides Counter<TMilli,uint32_t>
- GpioCaptureC.nc
- HilTimerMilliC.nc provides LocalTime<TMilli> and Timer<TMilli> as TimerMilli[uint8_t num]
- Msp430AlarmC.nc is generic and converts an MSP430 timer to a 16-bit Alarm
- Msp430Capture.nc HPL interface definition for MSP430 timer captures
- Msp430ClockC.nc exposes MSP430 hardware clock initialization
- Msp430ClockInit.nc HPL interface definition for hardware clock initialization
- Msp430ClockP.nc implements MSP430 hardware clock initialization and calibration and startup
- Msp430Compare.nc HPL interface definition for MSP430 timer compares
- Msp430Counter32khzC.nc provides Counter<T32khz,uint16_t> based on MSP430 TimerB
- Msp430CounterC.nc is generic and converts an Msp430Timer to a Counter
- Msp430CounterMicroC.nc provides Counter<TMicro,uint16_t> based on MSP430 TimerA
- Msp430Timer.h defines additional MSP430 timer bitmasks and structs
- Msp430Timer.nc HPL interface definition
- Msp430Timer32khzC.nc is generic and allocates a new 32khz hardware timer
- Msp430Timer32khzMapC.nc exposes the MSP430 hardware timers as a parameterized interface allocatable using Msp430Timer32khzC
- Msp430TimerC.nc exposes the MSP430 hardware timers
- Msp430TimerCapComP.nc is generic and implements the HPL for MSP430 capture/compare special function registers
- Msp430TimerCommonP.nc maps the MSP430 timer interrupts to Msp430TimerEvents
- Msp430TimerControl.nc HPL interface definition
- Msp430TimerEvent.nc HPL interface definition
- Msp430TimerP.nc is generic and implements the HPL for MSP430 timer special function registers

Implementation of timers for the ATmega128 and PXA27x may be found in tinyos-2.x/tos/chips/atm128 and tinyos-2.x/tos/chips/pxa27x/timer respectively.

7. Author's Address

Cory Sharp
Moteiv Corporation
55 Hawthorne St, Suite 550
San Francisco, CA 94105

phone - +1 415 692 0963
email - cory@moteiv.com

Martin Turon
P.O. Box 8525
Berkeley, CA 94707

phone - +1 408 965 3355
email - mturon@xbow.com

David Gay
2150 Shattuck Ave, Suite 1300
Intel Research
Berkeley, CA 94704

phone - +1 510 495 3055
email - david.e.gay@intel.com

Appendix A: Timer hardware on various microcontrollers

a. Atmega128

   i. Two 8-bit timers, each allowing
      - 7 prescaler values (division by different powers of 2)
      - Timer 0 can use an external 32768Hz crystal
      - One compare register, with many compare actions (change output pin, clear counter, generate interrupt, etc)

   ii. Two 16-bit timers, each with
      - 5 prescaler values
      - External and software clocking options
      - Three compare registers (again with many actions)
      - Input capture

b. MSP430

i. Two 16-bit timers with
   - One with three compare registers
   - One with eight compare registers
   - Each from distinct clock source
   - Each with limited prescalers

c. Intel PXA27x

i. One fixed rate (3.25MHz) 32-bit timer with
   - 4 compare registers
   - Watchdog functionality

ii. 8 variable rate 32-bit timers with
   - 1 associated compare register each
   - Individually selectable rates: 1/32768s, 1ms, 1s, 1us
   - Individually selectable sources: (32.768 external osc, 13 Mhz internal clock)

iii. Periodic & one-shot capability

iv. Two external sync events

Appendix B: a microcontroller: Atmega 128 timer sub-system

The Atmega128 exposes its four timers through a common set of interfaces:

- HplTimer<width> - get/set current time, overflow event, control, init
- HplCompare<width> - get/set compare time, fired event, control
- HplCapture<width> - get/set capture time, captured event, control, config

Parameterising these interfaces by width allows reusing the same interfaces for the 8 and 16-bit timers. This simplifies building reusable higher level components which are independent of timer width.

```
interface HplAtm128Timer<timer_size>
{
  /// Timer value register: Direct access
  async command timer_size get();
  async command void       set( timer_size t );

  /// Interrupt signals
  async event void overflow();        //<! Signalled on overflow interrupt

  /// Interrupt flag utilites: Bit level set/clr
  async command void reset(); //<! Clear the overflow interrupt flag
  async command void start(); //<! Enable the overflow interrupt
```

```
  async command void stop();  //<! Turn off overflow interrupts
  async command bool test();  //<! Did overflow interrupt occur?
  async command bool isOn();  //<! Is overflow interrupt on?

  /// Clock initialization interface
  async command void    off();                    //<! Turn off the clock
  async command void    setScale( uint8_t scale); //<! Turn on the clock
  async command uint8_t getScale();               //<! Get prescaler setti
}

interface HplAtm128Compare<size_type>
{
  /// Compare value register: Direct access
  async command size_type get();
  async command void      set(size_type t);

  /// Interrupt signals
  async event void fired();           //<! Signalled on compare interrupt

  /// Interrupt flag utilites: Bit level set/clr
  async command void reset();         //<! Clear the compare interrupt flag
  async command void start();         //<! Enable the compare interrupt
  async command void stop();          //<! Turn off comparee interrupts
  async command bool test();          //<! Did compare interrupt occur?
  async command bool isOn();          //<! Is compare interrupt on?
}

interface HplAtm128Capture<size_type>
{
  /// Capture value register: Direct access
  async command size_type get();
  async command void      set(size_type t);

  /// Interrupt signals
  async event void captured(size_type t);  //<! Signalled on capture int

  /// Interrupt flag utilites: Bit level set/clr
  async command void reset();          //<! Clear the capture interrupt fla
  async command void start();          //<! Enable the capture interrupt
  async command void stop();           //<! Turn off capture interrupts
  async command bool test();           //<! Did capture interrupt occur?
  async command bool isOn();           //<! Is capture interrupt on?

  async command void setEdge(bool up); //<! True = detect rising edge
}
```

These interfaces are provided by four components, corresponding to each hardware timer: HplAtm128Timer0AsyncC, and HplAtm128Timer0C through HplAtm128Timer3C. Timers 1 and 3 have three compare registers, so offer a parameterised HplAtm128Compare interface:

```
configuration HplAtm128Timer1C
{
  provides {
    // 16-bit Timers
    interface HplAtm128Timer<uint16_t>   as Timer;
    interface HplAtm128TimerCtrl16       as TimerCtrl;
    interface HplAtm128Capture<uint16_t> as Capture;
    interface HplAtm128Compare<uint16_t> as Compare[uint8_t id];
  }
}
...
```

where the `id` corresponds to the compare register number. The parameterised interface is only connected for `id` equal to 0, 1 or 2. Attempts to use another value cause a compile-time error. This is achieved as follows (code from the implementation of `HplAtm128Timer1C`)

```
Compare[0] = HplAtm128Timer1P.CompareA;
Compare[1] = HplAtm128Timer1P.CompareB;
Compare[2] = HplAtm128Timer1P.CompareC;
```

The Atmega128 chip components do not define a HAL, as the timer configuration choices (frequencies, use of input capture or compare output, etc) are platform-specific. Instead, it provides a few generic components for converting the HPL interfaces into platform-independent interfaces. These generic components include appropriate configuration parameters (e.g., prescaler values):

```
generic module Atm128AlarmC(typedef frequency_tag,
                            typedef timer_size @integer(),
                            uint8_t prescaler,
                            int mindt)
{
  provides interface Init;
  provides interface Alarm<frequency_tag, timer_size> as Alarm;
  uses interface HplTimer<timer_size>;
  uses interface HplCompare<timer_size>;
} ...

generic module Atm128CounterC(typedef frequency_tag,
                              typedef timer_size @integer())
{
  provides interface Counter<frequency_tag,timer_size> as Counter;
  uses interface HplTimer<timer_size> as Timer;
} ...
```

As a result of issues arising from using timer 0 in asynchronous mode, the HAL also offers the following component:

```
generic configuration Atm128AlarmAsyncC(typedef precision, int divider) {
  provides {
```

```
    interface Init @atleastonce();
    interface Alarm<precision, uint32_t>;
    interface Counter<precision, uint32_t>;
  }
}
...
```

which builds a 32-bit alarm and timer over timer 0. divider is used to initialise the timer0 scaling factor.

Appendix C: a mote: Mica family timer subsystem

Members of the mica family (mica2, mica2dot, micaz) use the Atmega128 microprocessor and have external crystals at 4 or 7.37MHz. Additionally, they can be run from an internal oscillator at 1, 2, 4, or 8 MHz. The internal oscillator is less precise, but allows for much faster startup from power-down and power-save modes (6 clocks vs 16000 clocks). Finally, power consumption is lower at the lower frequencies.

The mica family members support operation at all these frequencies via a `MHZ` preprocessor symbol, which can be defined to 1, 2, 4, or 8. If undefined, it defaults to a platform-dependent value (4 for mica2dot, 8 for mica2 and micaz).

The mica family configures its four timers in part based on the value of this MHZ symbol:

- Timer 0: uses Atm128AlarmAsyncC to divide the external 32768Hz crystal by 32, creating a 32-bit alarm and counter. This alarm and counter is used to build HilTimerMilliC, using the AlarmToTimerC, VirtualizeTimerC and CounterToLocalTimeC utility components.

  Timing accuracy is as good as the external crystal.

- Timer 1: the 16-bit hardware timer 1 is set to run at 1MHz if possible. However, the set of dividers for timer 1 is limited to 1, 8, 64, 256 and 1024. So, when clocked at 2 or 4MHz, a divider of 1 is selected and timer 1 runs at 2 or 4MHz. To reflect this fact, the HAL components exposing timer 1 are named `CounterOne16C` and `AlarmOne16C` (rather than the `CounterMicro16C` `AlarmMicro16C` as suggested in Section 3).

  32-bit microsecond Counters and Alarms, named `CounterMicro32C` and `AlarmMicro32C`, are created from `CounterOne16C` and `AlarmOne16C` using the TransformAlarmC and TransformCounterC utility components.

  Three compare registers are available on timer1, so up to three instances of `AlarmOne16C` and/or `AlarmMicro32C` can be created. The timing accuracy depends on how the mote is clocked:

  - internal clock: depends on how well the clock is calibrated
  - external 7.37MHz crystal: times will be off by ~8.6%
  - external 4MHz crystal: times will be as accurate as the crystal

- Timer 2: this timer is not currently exposed by the HAL.

- Timer 3: the 16-bit hardware timer 3 is set to run at a rate close to 32768Hz, if possible. As with timer 1, the limited set of dividers makes this impossible at some clock frequencies, so the 16-bit timer 3 HAL components are named `Counter3216C` and `Alarm3216C`. As with timer 1, the rate of timer 3 is adjusted in software to build 32-bit counter and 32-bit alarms, giving components `Counter32khz32C` and `Alarm32khz32C`. As with timer 1, three compare registers, hence up to three instances of `Alarm32khz32C` and/or `Alarm3216C` are available.

  At 1, 2, 4 and 8MHz, `Counter32khz32C` and `Alarm32khz32C` run at 31.25kHz (plus clock rate inaccuracy). At 7.37MHz, they run at ~28.8kHz.

The automatic allocation of compare registers to alarms (and corresponding compile-time error when too many compare registers are used) is achieved as follows. The implementations of `AlarmOne16C` and `AlarmThree16C` use the `Atm128AlarmC` generic component and wire it, using `unique`, to one of the compare registers offered by `HplAtm128Timer1C` and `HplAtm128Timer3C`:

```
generic configuration AlarmOne16C()
{
  provides interface Alarm<TOne, uint16_t>;
}
implementation
{
  components HplAtm128Timer1C, InitOneP,
    new Atm128AlarmC(TOne, uint16_t, 3) as NAlarm;

  Alarm = NAlarm;
  NAlarm.HplAtm128Timer -> HplAtm128Timer1C.Timer;
  NAlarm.HplAtm128Compare -> HplAtm128Timer1C.Compare[unique(UQ_TIMER1_COMP
}
```

On the fourth creation of an `AlarmOne16C`, `unique(UQ_TIMER1_COMPARE)` will return 3, causing a compile-time error as discussed in Appendix B (`HplAtm128Timer1C`'s `Compare` interface is only defined for values from 0 to 2).

When an Atmega128 is in any power-saving mode, hardware timers 1, 2 and 3 stop counting. The default Atmega128 power management *will* enter these power-saving modes even when timers 1 and 3 are enabled, so time as measured by timers 1 and 3 does *not* represent real time. However, if any alarms built on timers 1 or 3 are active, the Atmega128 power management will not enter power-saving modes.

The mica family HIL components are built as follows:

- HilTimerMilliC: built as described above from hardware timer 0.
- BusyWaitMicroC: implemented using a simple software busy-wait loop which waits for `MHZ` cycles per requested microsecond. Accuracy is the same as Timer 1.

Finally, the mica family motes measure their clock rate at boot time, based on the external 32768Hz crystal. The results of this clock rate measurement are made available via the `cyclesPerJiffy` command of the `Atm128Calibrate` interface of the `MeasureClockC` component. This command reports the number of cycles per 1/32768s. Please see this interface definition for other useful commands for more accurate timing.