Creating a New Platform for TinyOS 2.x

TEP:	131
Group:	TinyOS 8051 Working Group
Type:	Informational
Status:	Draft
TinyOS-Version:	2.x
Author:	Martin Leopold
Draft-Created:	6-Nov-2007
Draft-Version:	1
Draft-Modified:	6-Nov-2007
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Note

This memo is informational. It will hopefully be a basis for discussions and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

Abstract

The purpose of this TEP is to provide on overview of how to build a new TinyOS 2 platform. While the purpose of most TEPs is to describe TinyOS 2 entities, we will present concrete suggestions on how to implement a new TinyOS 2 platform. We will use examples and briefly cover the relevant TEPs to present a platform that adheres to the current TinyOS standards. We will not cover the TEPs in detail, but to the full text of each TEP for further information.

This TEP will go through the tool chain setup and the most basic components for a functional TinyOS platform. We consider only TinyOS version 2.x (from now on TinyOS).

Before venturing on this quest we will take a diversion and introduce general TinyOS 2 concepts and terminology (Section 1), readers familiar to TinyOS 2 can skip this section. This document will introduce the TinyOS 2 platform (Section 2) and describes the 3 elements that make up a platform: the tool chain (Section 3) the platform definitions (Section 4) and the chips definitions (Section 5).

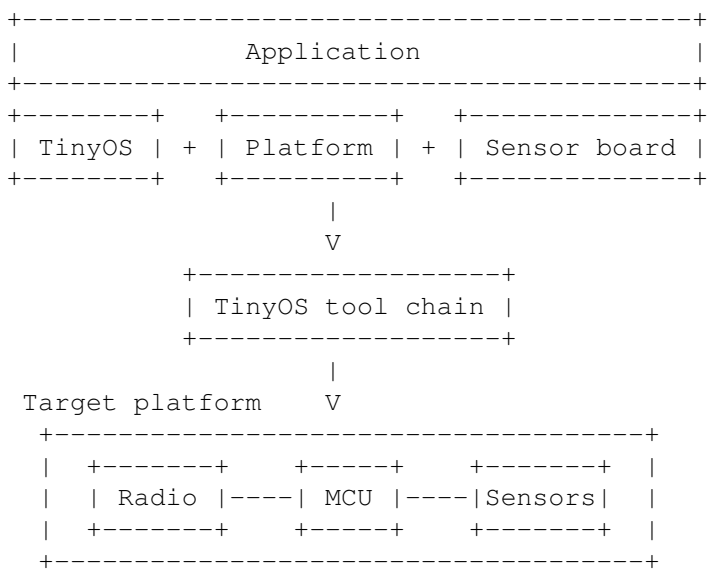
Table of Content

Contents

1. TinyOS Overview

Before describing the process of writing TinyOS platforms we will briefly sum up the TinyOS ecosystem and the terminology required in this TEP. To learn more visit the TinyOS website http://www.tinyos.net.

A systems overview is depicted below. In this TEP we will primarily concern our selves with the platform portion of figure and briefly cover the tool chain. This involves writing the necessary drivers and writing rules to pass the code to the TinyOS tool chain. We will not cover sensor boards in this TEP refer to, see [TEP109] for details.



1.1 TinyOS 2 architecture

TinyOS 2.x is built on a tree-layered hardware abstraction architecture (HAA)[TEP2]. This architecture separates the code for each platform into distinct layers:

1. the Hardware Independent Layer (HIL)
2. the Hardware Adaptation Layer (HAL)
3. the Hardware Presentation Layer (HPL)

A platform is built from bottom up, starting with the HPL level, building HAL and HIL layers on top. Platform independent applications are written using HIL level interfaces, allowing them to move easily from platform to platform. While applications can target a platform specific HAL layer for finer control of hardware specific features, this will could prohibit such an application from being easily portable. An overview of the TinyOS 2 architecture is given in [tos2.0view].

The requirements for the platform implementation is described in TinyOS Enhancement Proposals (TEP). Each TEP covers a particular area and specifies the recommendations within that area, some of which are relevant for platforms. While no specific label or designation is given to platforms adhering to the set of TEPs, [TEP1] states: “Developers desiring to add code (or TEPs) to TinyOS SHOULD follow all current

BCPs (Best Current Practice)". At the time of writing no TEP has been awarded this designation or been finalized and we will refer to the drafts as they are.

This document will not go through each of the requirements, but merely outline how to build a basic functional platform. For further information see "TinyOS 2.0 Overview" [tos2.0view] or the TEP list on the TinyOS website http://www.tinyos.net.

1.2 TinyOS Contrib

The core of TinyOS is maintained by a set of working groups that govern specific parts of the source code. New project can benefit from the *contrib* section of TinyOS. This is a separate section of the website and source repository maintained more loosely than the core of TinyOS. It is intended for sharing code at an early stage or code that may not gain the same popularity as the core.

New projects request a directory in this repository by following a simple procedure on the TinyOS contrib web page

In contrib is a skeleton project *skel* that provides the most basic framework for setting up a new platform, MCU, etc.

2. A TinyOS Platform

A TinyOS platform provides the code and tool chain definitions that enable an application writer to implement an application for a mote. A platform in TinyOS exposes some or all of the features of a particular physical mote device to TinyOS applications - it refers to an entire system, not a single chip. In order to write programs for a device using TinyOS a platform for that device must exist within TinyOS.

A physical platform is comprised of a set of chips. Similarly a TinyOS platform is the collection of the components representing these chips (corresponding to drivers) Common chips can be shared among platforms and implementing a new platform could simply mean wiring existing components in a new way. If the chips that make up the platform are not supported by TinyOS implementing the new platform consists if implementing components for those chips (much like implementing drivers).

2.1 A New Platform

Platforms are discovered at compile time by the TinyOS tool chain and a new platform placed in the search path will be discovered automatically. In addition sensor boards can be defined in a very similar manner, however we will not cover sensor boards, see [TEP109] for details. Defining a new platform boils down to 3 things:

1. definitions of the chips that make up the platform,
2. platform definitions (combining chips to a platform) and,
3. the tool chain or make definitions.

The code for a TinyOS platform is spread out in a few locations of the TinyOS tree depending based on those 3 categories. Below is an overview of the locations and some of the files we will be needing in the following (for further information see see "Coding Conventions" [TEP3] and the "README" files in each directory).

Through this TEP we will use the terms PlatformX and MCUX to denote the new generic platform and MCU being created:

```
tos
 +--chips                              1. Chip definitions
 |    +--chipX
 +--platforms
 |    +--platformX                     2. Platform definitions
 |         +--PlatformP/PlatformC
 |         +--PlatformLeds                 example component
 |         +--.platform
 |         +--hardware.h
 |         +--chips
 |              +--MCUX                    Platform specific features
 |              +--chipX
 +--sensorboards
 |    +--boardX
 |         +--.sensor
 +--support
      +--make                          3. Make definitions
           +--platformX.target             platformX make targets
           +--MCUX
                +--MCUX.rules              make rules for MCUX
                +--install.extra           additional target for MCUX
```

In the following we will briefly introduce each of the parts and describe them in more detail in sections 2 through 4.

2.2 The Chips

Each of the chips that provide software accessible functionality must have definitions present in the chips directory sensors, radios, micro controllers (MCU) alike. Each chip is assigned a separate directory under `tos/chips`. This directory contains chip specific interfaces (HPL and HAL interfaces) and their implementations as well as implementations of the hardware independent interface (HIL).

Some chips, MCUs in particular, contain distinct subsystems, such as uart, timer, A/D converter, SPI, and so forth. These subsystems are often put in a sub directory of their own within the chip-specific directory.

If some feature of a chip is available or used only on a particular platform, the platform directory can contain code that is specific to this combination of chip and platform, say pin assignments, interrupt assignments, etc. For example such additions would be placed in `tos/platforms/platformX/chips/chipX` for PlatformX.

2.3 The Platform Directory

The platform is the piece of the puzzle that ties the components corresponding to physical chips (drivers) together to form a platform. The platform ties together the code that exposes the features of the platform to TinyOS programs. In practise this is done by i) including code for each of the chips and ii) by providing any additional code that is specific to this particular platform.

A platform *PlatformX* would be placed in the directory `tos/platforms/platformX/` and code for subsystems reside in further sub directories. Also in this directory is the

`.platform` file that sets up include paths and more (see Section 3).

An empty platform with no code (null) is provided and serves as an example for other platforms.

2.4 The Tool-Chain (Make System)

The build system for TinyOS is written using GNU Make[1]. The build system controls the process from pre-processing a TinyOS application into a single C file and to pass this file to the appropriate compiler and other tools. The make system is documented in `support/make/README` and in TinyOS 2 Tutorial Lesson 10[TUT10].

The make system is located in the `support` directory. This directory contains a platform definition and Make rules to build an application for this platform (see Section 2).

2.5 The Minimal Platform

Before describing each of the subsystems, we will show a simple check list. The absolute minimal TinyOS platform would have to provide the following resources, given a *PlatformX* with *MCUX*:

- a platform directory `tos/platform/PlatformX/` with the following
  - a platform definition (*.platform* file)
  - a *hardware.h* header
- a *platformX.target* in `tos/support/make`
- a *MCUX.rules* in `tos/support/make/MCUX`
- a *MCUX* directory in `tos/chips/MCUX`, containing
  - a *McuSleepC* (must enable interrupts)
  - a *mcuxhardware.h* (defines *nesc_atomic_start*/*nesc_atomic_end*)

3. Tool Chain

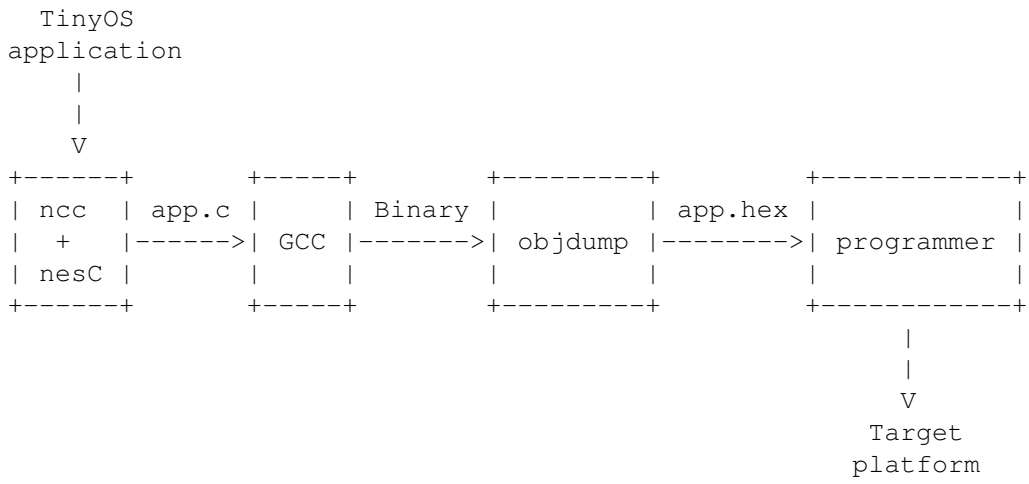
The major components in the tool chain of TinyOS are i) the compiler and ii) the build system that uses the compiler to produce an executable binary or hex file. The first is installed separately, while the second is part of the TinyOS source code. The compile process transforms a set of nesC files into a binary executable or hex file. Involved in this process is set of separate tools that are linked in a chain. We will briefly cover this chain in a moment, but a detailed description is beyond the scope of this TEP.

The make system is split in two: a general part and a platform specific part. Section 3.1 will introduce the general mechanism and Section 3.2 will cover how to introduce a new platform in the tool chain.

[1] http://www.gnu.org/software/make/

3.1 Compiling using nesC

The process of the build system is depicted below. This system feeds the source code through the tools to produce an executable or hex file for uploading to the platform. The nesC pre-compiler is split in two tools ncc an nescc. These two tools are used to assemble nesC source files into a single C file which is compiled using a regular C compiler. This requires that a C compiler is available for a given platform and that this compiler accepts the dialect produced by nesC.



The core TinyOS platforms are centered around GCC, this includes the telos family, mica family and intelmote2. The nesC compiler expects code resembling GCC C-dialect and also outputs code in GCC C-dialect. The current TinyOS platforms are supported by GCC, but for some processor architectures GCC is not available (e.g. Motorola HCS08, Intel MCS51).

Porting to platforms that are GCC supported can benefit from the existing tool flow, while porting to other platforms requires some effort. A straight forward solution adopted for these platforms is to post-process the C files produced by nesC to fit the needs of a specific compiler, see TEP121 for an example of such a solution.

3.2 The Make System

TinyOS controls the build process using make. The global make file searches certain locations to find make definitions for all available platforms. In order to make a new platform available to the TinyOS make system, a few files must be created in particular locations. These files will be read by the global make system and exposed to the users by make targets. Often the required rules are tied to a particular MCU that is shared among several platforms and TinyOS leverages this fact by creating a light-weight *.target* file pointing to the appropriate rules in a *.rules* file. The make system is documented in `support/make/README` and in TinyOS 2 Tutorial Lesson 10[TUT10].

The make system looks for *.target* files in `support/make` and the directories listed in the environment variable `TOSMAKE_PATH`. Each of the files found contain make targets for one TinyOS platform. The target files usually do not contain the rules to build the binary files, but include the appropriate rules from a *.rules* file, located in a sub directory for the appropriate MCU architecture (e.g. avr for the ATMega128 used by Mica). In this way many platforms share the build rules, but have different *.target*

files. In addition *.extra* targets can be used to define helper targets such as install or clean.

Setting up the make system, requires two steps (Section 3.2.1 gives an example):

1. Creating a *platformX.target* file that allows the make system to discover the new platform. This file must contain a make rule with the name of the platform. Further this target must depend on the targets given in the variable `BUILD_DEPS` - this variable contains the remainder of targets to be build during the build process.
2. Creating a *.rules* file in a sub diretory of `support/make`. Each file contain the actual target for producing the binaries, hex files, etc. for one platform. They are assembled in the `BUILD_DEPS` variable.

We will cover these two files next.

3.2.1 The .target file

As mentioned above TinyOS searches for targets in the `support/make` directory of the TinyOS source code and in the directories listed in the environment variable `TOSMAKE_PATH`. A *.target* file for the platform must exist in one of these locations. The *.target* usually only sets up variables related to this platform and provide a target named after the platform, this target depend on other rules to build the binaries. These rules are included by calling *TOSMake_include_platform*. As an example the `mica2.target` is listed below:

```
PLATFORM = mica2
SENSORBOARD ?= micasb
PROGRAMMER_PART ?= -dpart=ATmega128 --wr_fuse_e=ff
PFLAGS += -finline-limit=100000

AVR_FUSE_H ?= 0xd9

$(call TOSMake_include_platform,avr)

mica2: $(BUILD_DEPS)
        @:
```

Pay attention to the call to *TOSMake_include_platform,avr* this call includes `.rules` files in `support/make` or any sub directory of `TOSHMAKE_PATH` named *avr*.

3.2.2 The .rules file

The *.rules* file contain the make rules for building the target binary. If a MCU implementation already exists for a new platform, simply pointing to this *.rules* from the corresponding *.target* is sufficient. If not the *.rules* file must be built for a new MCU architecture.

`TOSMake_include_platform` expects a sub directory with a rule file of the form `avr/avr.rules`. The call also includes additional *.rules* and *.extra* files present in that sub directory. See `support/make/README` for details.

For example a simplified .rules file could look like this (from *avr/avr.rules*):

```
...

    BUILD_DEPS = srec tosimage

srec: exe FORCE
        $(OBJCOPY) --output-target=srec $(MAIN_EXE) $(MAIN_SREC)

tosimage: ihex build_tosimage FORCE
        @:

exe: builddir $(BUILD_EXTRA_DEPS) FORCE
        @echo "    compiling $(COMPONENT) to a $(PLATFORM) binary"
        $(NCC) -o $(MAIN_EXE) $(OPTFLAGS) $(PFLAGS) $(CFLAGS)
                  $(COMPONENT).nc $(LIBS) $(LDFLAGS)
...
```

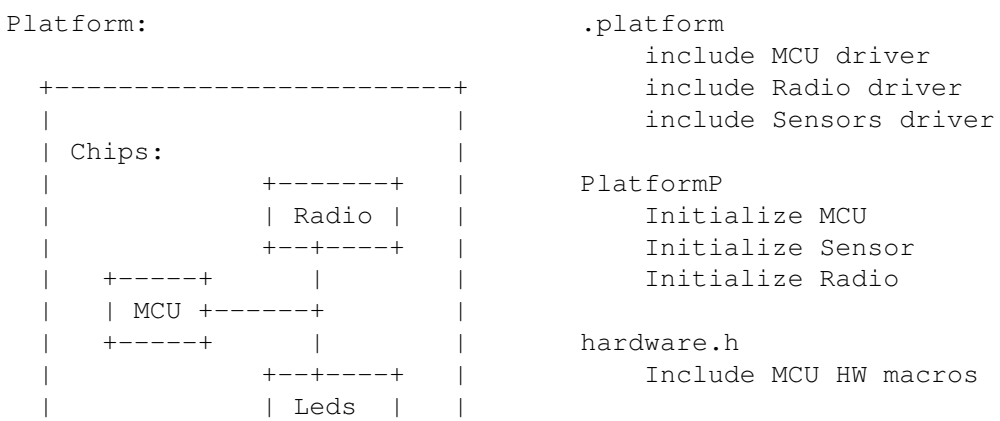
4. The Platform

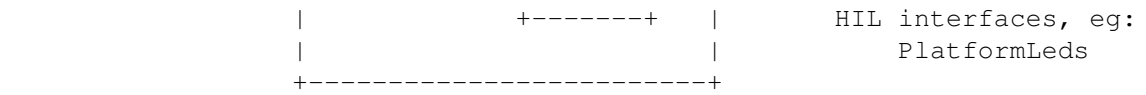
A TinyOS platform is a collection of components corresponding to a physical devices (a collection of drivers). A platform is constructed by defining which components make up the platform and by providing any additional platform specific components. The content of a platform is not covered by a TEP at the time of writing and the following is based on the available tutorials, *READMEs* and the current consensus.

The platform definitions for a *PlatformX* are located in `tos/platforms/platformX` and contain 3 major elements:

1. The *.platform* file containing include paths and other arguments for nesC
2. Platform boot procedure: PlatformP/PlatformC
3. Platform specific code, including a header for hardware specific funtions (hardware.h) and code that is specific to the combination of chip and platform (e.g. pin assignments, modifications, etc.).

In the following we will describe each of these in more detail, below is a depiction of a fictuous platform and the required definitions.





All the files we describe here must be found in a common directory named after the platform; we call this a platform directory. This directory must be found in the TinyOS tool chain search path for example `tos/platforms` (or found in `TOSHMAKE_PATH` see Section 3.2.1). For example a platform named PlatformX could be placed in the directory `tos/platforms/platformX`.

4.1 .platform file

All platform directories must carry a `.platform` file, this file defines what makes up the platform. This file carries instructions for the make system on where to locate the drivers for each of the components of the platform. The definitions are read in a two step process: the file is read by the ncc script that passes the appropriate arguments to the nesC pre-processor[nescman]. The .platform file is written as a Perl script interpreted by ncc.

The file is documented in form of the README file in the platforms directory (*tos/platforms*), and the source code of the ncc script found in the TinyOS distribution. Valuable information can also be found in [TEP106], [TEP109], TinyOS 2 Tutorial Lesson 10[TUT10].

In addition to setting up include paths for nesC other arguments can be passed on. In particular the components to be used for the scheduler are selected with the '-fnesc-scheduler' command line argument (see [TEP106]). The include paths are passed on using the '-I' command line argument covered in [TEP109].

As an example, an abbreviated *.platform* file for the mica2 platform looks like this:

```
push( @includes, qw(
  %T/platforms/mica
  %T/platforms/mica2/chips/cc1000
  %T/chips/cc1000
  %T/chips/atm128
  %T/chips/atm128/adc
  %T/chips/atm128/pins
  %T/chips/atm128/spi
  %T/chips/atm128/timer
  %T/lib/timer
  %T/lib/serial
  %T/lib/power
) );

@opts = qw(
  -gcc=avr-gcc
  -mmcu=atmega128
  -fnesc-target=avr
  -fnesc-no-debug
  -fnesc-scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,TaskBasic,TaskBa
);
```

4.2 PlatformP and PlatformC

The PlatformP and PlatformC components are responsible for booting this platform to a usable state. In general this usually means things like calibrating clock and initializing I/O pins. If this platform requires that some devices are initialized in a particular order this can be implemented in a platform dependent way here. The boot process covered in [TEP107].

Most hardware peripherals require an initialization procedure of some sort. For most peripheral devices this procedure is only required when the device is in use and can be left out if the device is unused. TinyOS accomplishes this by providing a few initialization call backs interfaces. When a given component is included it wires an initialization procedure to one of these interfaces. In this way the procedure will only be included when the component is included, this process is known as auto wiring[TOSPRG].

The boot sequence calls two initialization interfaces in the following order: `PlatformC.Init` and `MainC.SoftwareInit`. `PlatformC.Init` must take care of initializing the hardware to an operable state and initialization that has hidden dependencies must occur here. Other components are initialized as part of `SoftwareInit` and the orderign is determined at compile time.

Common tasks in `PlatformC.Init` include clock calibration and IO pins. However this component can be used to provide greater control of the boot process if required. Consider for example that some component which is initialized in SoftwareInit requires that some other component has been initialized previously - lets say that the radio must be initialized prior to the radio stack or that a component prints a message to the UART during boot. How can this order be ensured? One solution is to provide an additional initialization handle prior to SoftwareInit and wire such procedures to this handle. Below is an example from the Mica mote family using the `MoteInit` interface. Components that must be initialized early in the boot process is wired to this interface. If greater level of control is required this strategy can be trivially be expanded to further levels of interfaces for example MoteInit1 being initialized prior to MoteInit2, this strategy is chosen by the MSP430 implementation.

4.2.1 PlatformC

Below is depicted the PlatformC component from the Mica family of platforms.

```
#include "hardware.h"

configuration PlatformC {
  provides {
    interface Init;
    /**
     * Provides calibration information for other components.
     */
    interface Atm128Calibrate;
  }
  uses interface Init as SubInit;
} implementation {
  components PlatformP, MotePlatformC, MeasureClockC;

  Init = PlatformP;
```

```
  Atm128Calibrate = MeasureClockC;

  PlatformP.MeasureClock -> MeasureClockC;
  PlatformP.MoteInit -> MotePlatformC;
  MotePlatformC.SubInit = SubInit;
}
```

4.2.1 PlatformP

Below is depicted the PlatformP component from the Mica family of platforms.

```
module PlatformP
{
  provides interface Init;
  uses interface Init as MoteInit;
  uses interface Init as MeasureClock;

}
implementation
{
  void power_init() {
     ...
  }

  command error_t Init.init()
  {
    error_t ok;

    ok = call MeasureClock.init();
    ok = ecombine(ok, call MoteInit.init());
    return ok;
  }
}
```

4.2.3 Init example: PlatformLedsC

Below is an example from mica/PlatformLedsC.nc wiring to the platform specific *MoteInit* interface.

```
configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
...
  uses interface Init;
} implementation {
  components HplAtm128GeneralIOC as IO;
  components PlatformP;

  Init = PlatformP.MoteInit;
...
```

3.3 Platform Specific Code

In addition to PlatformP/PlatformC the platform directory contain additional code that is specific to this platform. First this could be code that ties platform independent resources to particular instances on this platform, second it could be code that is only relevant on this particular platform (e.g. the clock rate of this platform). For example the LEDs are a an example of the first: most platform have a few LEDs, and the particular pin on this platform is tied to the platform independent implementation using PlatformLedsC.

TinyOS requires that the header `hardware.h` is present while other component can be named freely.

4.3.1 Platform Headers

TinyOS relies on a few C-header files being present for each platform. These headers are then automatically included from the respective parts of the TinyOS system. TinyOS expects that the following files are present and that certain properties are defined in them.

hardware.h The `hardware.h` header file is included by `tos/system/MainC.nc` and usually in turn includes a MCU specific header file, with a few required macros (such as avr128hardware.h, see Section 5.1.1). The header is documented in TinyOS 2 Tutorial Lesson 10[TUT10]

In addition the hardware.h file can set flags that are not related to the hardware in general, but to this platform (e.g. clock rate). Below is a snippet from `mica2/hardware.h`

```
#ifndef MHZ
/* Clock rate is ~8MHz except if specified by user
   (this value must be a power of 2, see MicaTimer.h and MeasureClockC.nc)
#define MHZ 8
#endif

#include <atm128hardware.h>

enum {
  PLATFORM_BAUDRATE = 57600L
};
...
```

platform_message.h As part of the TinyOS 2 message buffer abstraction[TEP111], TinyOS includes the header *platform_message.h* from the internal TinyOS header *message.h*. This header is only strictly required for platforms wishing to use the message_t abstraction - this is not described further in this TEP, but is used widely throughout TinyOS (See [TEP111] for details). The is expected to define the structures: *message_header_t*, *message_footer_t*, and *message_metadata_t* which are used to fill out the generic *message_t* structure.

Below is an example from the `mica2` *platform_message.h*

```
typedef union message_header {
  cc1000_header_t cc1k;
  serial_header_t serial;
```

```
} message_header_t;

typedef union message_footer {
  cc1000_footer_t cc1k;
} message_footer_t;

typedef union message_metadata {
  cc1000_metadata_t cc1k;
} message_metadata_t;
```

4.3.2 Platform Specific Components

The code for platform dependent features also resides in the platform directory. If the code can be tied to a particular chip it can be placed in a separate directory below the `chips` directory. As an example the following section of `micaz/chips/cc2420/HplCC2420PinsC.nc` ties specific pins to general names on the MicaZ platform.

```
configuration HplCC2420PinsC {
  provides {
    interface GeneralIO as CCA;
    interface GeneralIO as CSN;
    interface GeneralIO as FIFO;
    interface GeneralIO as FIFOP;
    interface GeneralIO as RSTN;
    interface GeneralIO as SFD;
    interface GeneralIO as VREN;
  }
}

implementation {

  components HplAtm128GeneralIOC as IO;

  CCA    = IO.PortD6;
  CSN    = IO.PortB0;
  FIFO   = IO.PortB7;
  FIFOP  = IO.PortE6;
  RSTN   = IO.PortA6;
  SFD    = IO.PortD4;
  VREN   = IO.PortA5;

}
```

5. The chips

The functionality of each chip is provided by a set of one or more interfaces and one or more components, in traditional terms this makes up a driver. Each chip is assigned a sub directory in the the `tos/chips` directory. All code that define the functionality of a chip is located here regardless of the type of chip (MCU, radio, etc.). In addition

MCU's group the code related to separate sub systems into further sub directories (e.g. `tos/chips/atm128/timer`).

In this section we will go trough some of the peripherals commonly built into MCUs, but we will not go trough other chips such as sensors or radio.

5.1 MCU Internals

Apart from the drivers for each of the peripheral units, a few additional definitions are required for the internals of the MCU. This includes i) atomic begin/end and ii) low power mode. The first is defined in the header *hardware.h* and the latter component *MCUSleepC*.

5.1.1 mcuXardware.h

Each architecture defines a set of required and useful macros in a header filed named after the architecture (for example *atm128hardware.h* for ATMega128). This header is then in turn included from *hardware.h* in the platform directory (See Section 4.3.1).

A few of the macros are required by nesC code generation. nesC will output code using these macros and they must be defined in advance, other useful macros such as interrupt handlers on this particular platform can be defined here as well. The required macros are:

- *__nesc_enable_interrupt* / *__nesc_disable_interrupt*
- *__nesc_atomic_start* / *__nesc_atomic_end*

Below is a few examples from *atm128hardware.h*

```
/* We need slightly different defs than SIGNAL, INTERRUPT */
#define AVR_ATOMIC_HANDLER(signame) \
  void signame() __attribute__ ((signal)) @atomic_hwevent() @C()

#define AVR_NONATOMIC_HANDLER(signame) \
  void signame() __attribute__ ((interrupt)) @hwevent() @C()

...

inline void __nesc_enable_interrupt()  { sei(); }
inline void __nesc_disable_interrupt() { cli(); }

...

inline __nesc_atomic_t
__nesc_atomic_start(void) @spontaneous()  {
    __nesc_atomic_t result = SREG;
    __nesc_disable_interrupt();
    asm volatile("" : : : "memory");
    return result;
}

/* Restores interrupt mask to original state. */
```

```
inline void
__nesc_atomic_end(__nesc_atomic_t original_SREG) @spontaneous() {
  asm volatile("" : : : "memory");
  SREG = original_SREG;
}
```

5.1.2 MCUSleepC

TinyOS manages the power state of the MCU through a few interfaces. These interfaces allow components to signal how and when this platform should enter low power mode. Each new MCU in TinyOS must implement the *MCUSleepC* component that provide the *McuSleep* and *McuPowerState* interfaces.

The TinyOS scheduler calls *McuSleep.sleep()* when it runs out of tasks to start. The purpose of this function is to make the MCU enter the appropriate sleep mode. The call to *McuSleep.sleep()* is made from within an atomic section making it essential that sleep() enables interrupts before entering sleep mode! If the interrupts not enabled prior to entering sleep mode the MCU will not be able to power back up.

A dummy MCU sleep component that does not enter sleep mode, but merely switches interrupts on and off is shown below. This ensures that the platform will not lock up even without proper sleep support.

```
#include "hardware.h"

module McuSleepC {
  provides interface McuSleep;
  provides interface McuPowerState;
  uses  interface McuPowerOverride;
} implementation {
  async command void McuSleep.sleep() {
    __nesc_enable_interrupt();
    // Enter sleep here
    __nesc_disable_interrupt();
  }

  async command void McuPowerState.update() { }
}
```

5.2 GeneralIO

Virtually all micro controllers feature a set of input output (I/O) pins. The features of these pins is often configurable and often some pins only support a subset of features. The HIL level interface for TinyOS is described in [TEP117] and uses two interface to describe general purpose I/O pins common to many MCUs:

- **GeneralIO**: Digital input/output. The GeneralIO interface describes a digital pin with a state of either *clr* or *set*, the pin must be capable of both input and output. Some MCUs provide pins with different capabilities: more modes (e.g. an alternate peripheral mode), less modes (e.g. only input) or a third "tri-state" mode. Such chip specific additional features are not supported. Some chips group a set of pins into "ports" that can be read or written simultaneously, the HIL level interface does not support reading or setting an entire port.

- **GpioInterrupt**: Edge triggered interrupts. GpioInterrupt support a single pin providing an interrupt triggered on a rising or falling edge. Pins capable of triggering on an input level or only triggering on one edge is not supported.

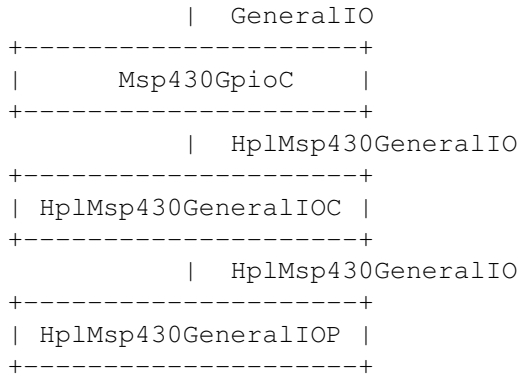
While these interfaces are aimed at the HIL level some platforms use the GeneralIO and GpioInterrupt interface to represent the HPL level (atm128 for example) and others platforms define their own interface to capture the entire functionality (msp430, pxa27x for example).

[TEP117] states that each platform must provide the general IO pins of that platform through the *GeneralIO* interface and should do this though the component *GeneralIOC*. It is however not clear how the entire set of pins should be provided - this could be in the form of a list of pins, group of pins, a generic component, etc.

The pin implementations are usually found in the *pins* sub directory for a particular MCU (e.g. *msp430/pins* for MSP430 pin implementation).

5.2.1 Example MSP430

The MSP430 implementation builds the platform independent pin implementation as a stack of components starting with a platform specific component for each pin.



At the bottom the component `HplMsp430GeneralIOP` provides a general implementation of one Msp430 pin using the `HplMsp430GeneralIO` interface. The generic component `HplMsp430GeneralIOP` is then instantiated for each pin by `HplMsp430GeneralIOC`.

```
interface HplMsp430GeneralIO {
  async command void set();
  async command void clr();
  async command void toggle();
  async command uint8_t getRaw();
  async command bool get();
  async command void makeInput();
  async command bool isInput();
  async command void makeOutput();
  async command bool isOutput();
  async command void selectModuleFunc();
  async command bool isModuleFunc();
  async command void selectIOFunc();
  async command bool isIOFunc();
```

```
}
```

This interface is implemented in the generic component HplMsp430GeneralIOP (abbreviated):

```
generic module HplMsp430GeneralIOP(uint8_t port_in_addr, ...)  {
   provides interface HplMsp430GeneralIO as IO;
 } implementation  {
   #define PORTx (*(volatile TYPE_PORT_OUT*)port_out_addr)

   async command void IO.set() { atomic PORTx |= (0x01 << pin); }
   async command void IO.clr() { atomic PORTx &= ~(0x01 << pin); }
   async command void IO.toggle() { atomic PORTx ^= (0x01 << pin); }

...
```

These are then instantiated from HplMsp430GeneralIOC (abbreviated):

```
configuration HplMsp430GeneralIOC {
  provides interface HplMsp430GeneralIO as Port10;
  provides interface HplMsp430GeneralIO as Port11;
...
} implementation {
  components
    new HplMsp430GeneralIOP(P1IN_, P1OUT_, P1DIR_, P1SEL_, 0) as P10,
    new HplMsp430GeneralIOP(P1IN_, P1OUT_, P1DIR_, P1SEL_, 1) as P11,
...
  Port10 = P10;
  Port11 = P11;
...
```

And finally these are transformed from the interface HplMsp430GeneralIO to the platform independent GeneralIO using the generic component Msp430GpioC (abbreviated):

```
generic module Msp430GpioC() {
  provides interface GeneralIO;
  uses interface HplMsp430GeneralIO as HplGeneralIO;
} implementation {
  async command void GeneralIO.set() { call HplGeneralIO.set(); }
  async command void GeneralIO.clr() { call HplGeneralIO.clr(); }
  async command void GeneralIO.toggle() { call HplGeneralIO.toggle(); }
...
```

5.3 LEDs

Having a few leds on a platform is quite common and TinyOS supports three leds in a platform independent manner. Three leds are provided through the *LedsC* component regardless of the amount of leds available on the platform. The LED implementation is not covered by a TEP at the time of writing, but the general consensus between

most platforms seems to be to provide access to 3 LEDs through the component *PlatformLedsC*. This component is then used by *LedC* and *LedP* (from `tos/system`) to provide a platform independent Led interface.

The PlatformLedsC implementation is found in the platform directory of each platform (e.g. *tos/platforms/mica2* for mica2). The consensus is as follows:

- Each platform provides the component PlatformLedsC
- PlatformLedsC provides Led0, Led1, Led2 using the GeneralIO interface. If the platform has less than 3 Leds these are wired to the component NoPinC
- PlatformLedsC *uses* the Init interface and usually it wired back to an initialization in PlatformP - this way when LedC is included in an application it will be initialized when PlatformP call this interface

5.3.1 Example Mica2dot

```
configuration PlatformLedsC
{
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation  {
  components HplAtm128GeneralIOC as IO;
  components new NoPinC();
  components PlatformP;

  Init = PlatformP.MoteInit;

  Led0 = IO.PortA2;  // Pin A2 = Red LED
  Led1 = NoPinC;
  Led2 = NoPinC;
}
```

5.3.2 LEDs implementation discussion

The Led interface described above is not specified in a TEP, though at the time of writing most platforms seem to follow this pattern. Not being covered by a TEP leads to a few uncertainties:

- *PlatformLedsC* exports leds using the *GeneralIO* interface. The intention is of course that a led is connected to a pin and this pin is used to turn the led on. *LedC* uses this assumption to calls the appropriate *makeOutput* and *set/clr* calls. However this might not be the case - a led could be connected differently making the semantics of, say *makeOutput* unclear. Furthermore it is assumed that the set call turns the led on, while the actual pin might have to be cleared (active low). And what is the semantics of *makeInput* on a LED? Abstracting these differences into on/off, enable/disable semantics would clear up the uncertainties.

- Initializing the Leds is important - the Leds can consume power even when turned off, if the corresponding pins are configured inappropriately. Including the LedsC component initializes the Leds as output when this component is used, if not they must be initialized elsewhere (e.g. PlatformP). It would seem elegant to group related code (e.g. one Leds component).
- The interface does not provide a uniform way of accessing any additional LEDs other than the 3 found on the Mica motes. While this is inherently platform specific, having a common consensus would be useful, say for example exporting an array of Led. In this way a program requiring say 4 leds could be compiled if 4 leds are available.

5.4 Timers

The timer subsystem in TinyOS 2 is described in [TEP102] the TEP specifies interfaces at HAL and HIL levels that a platform must adhere to. The timer does not cover the possible design choices at the HPL level. The timers defined in this TEP are described in terms of *width* of the underlying counters (e.g. 8, 16, 32 bit) and in the tick period or frequency denoted *precision* (e.g. 10 kHz, 1 ms, 1 us). The timer subsystem is provided through a set of hardware independent interfaces and a set of recommended configuration modules. As such some features of a particular device may not be available in a device independent manner.

The functionality commonly seen in MCU timer units is often split in 3 parts: control, timer/counter, capture/compare(trigger). Timer control in is inherently platform specific and is not covered by a TEP. The timer [TEP102] covers timer/counter while capture/compare is covered by [TEP117]. From the TEPs it is not clear how capture/compare and timer/counter relate to each other even though they are likely to refer to the same time base. The calibration of the system clock is often connected with the timer system and lives in the timer directory, but it is not covered in [TEP102].

The timer implementation for a given MCU is placed in the *timer* sub directory of the directory for a given MCU (e.g. *tos/chips/avr/timer*). Often clock initialization components are placed in the same directory, but these are not covered by the TEPs.

5.4.1 Timer Layers

TEP117 follows the 3 layer architecture and separates the platform dependent and independent abstractions. It poses not restrictions on the HPL layer, but provides a suggested HAL layer and requirements at the HIL level. It defines a set of interfaces and precisions that are used at the HAL and HIL levels.

The common features of a timer are separated into a set of interfaces, corresponding roughly to common features in MCU timer units. The interfaces covered by the [TEP102] are (capture from [TEP117]):

- **BusyWait<precision_tag,size_type>** Short synchronous delays
- **Counter<precision_tag, size_type>** Current time, and overflow events
- **Alarm<precision_tag,size_type>** Extend Counter with at a given time
- **Timer<precision_tag>** 32 bit current time, one shot and periodic events
- **LocalTime<precision_tag>** 32 bit current time without overflow

- **GpioCapture** 16 bit time value on an external event (precision unspecified)

The precisions defined in [TEP117] are defined as "binary" meaning 1024 parts of a second. Platforms must provide these precisions (if possible) regardless of the underlying system frequency (i.e. this could be a power of 10).

- **TMilli** 1024 Hz (period 0.98 ms)
- **TMicro** 1048576 Hz (period 0.95 us)
- **T32khz** 32.768 kHz (period 30.5 us)

HPL The features of the underlying hardware timers are generally exposed using one or more hardware specific interfaces. Many MCU provide a rich a set of different timer units varying in timer width, frequency and features.

Many MCUs provides features that are quite close to the features described by the interfaces above. The HPL layer exposes these features and the layers above utilize the available hardware and virtualize the features if necessary.

Consider for example generating events at a specific time. Often a timer unit has a single counter and a few compare registers that generate interrupts. The counter and compare registers often translate relative straightforward to Counter and Alarm interfaces.

HAL Each platform should expose a hardware timer of precision P and width W as Counter${P}${W}C, Alarm${P}${W}C(). For example an 8 bit, 32.768 kHz timer should be exposed as Counter32khz8C and Alarm32khz8C

Alarm/Counters come in pairs and refer to a particular hardware unit, while there is only one counter there is commonly multiple compare registers. Alarm is a generic component and each instantiation must provide an independent alarm allocating more Alarm components than available compare register should provide a compile time error.

HIL Each platform must expose the following components that provide specific width and precision that are guaranteed to exist on TEP compliant platforms.

- **HilTimerMilliC** A 32 bit Timer interface of TMilli precision
- **BusyWaitMicroC** A BusyWait interface of TMicro precision

5.4.2 Timer Implementation

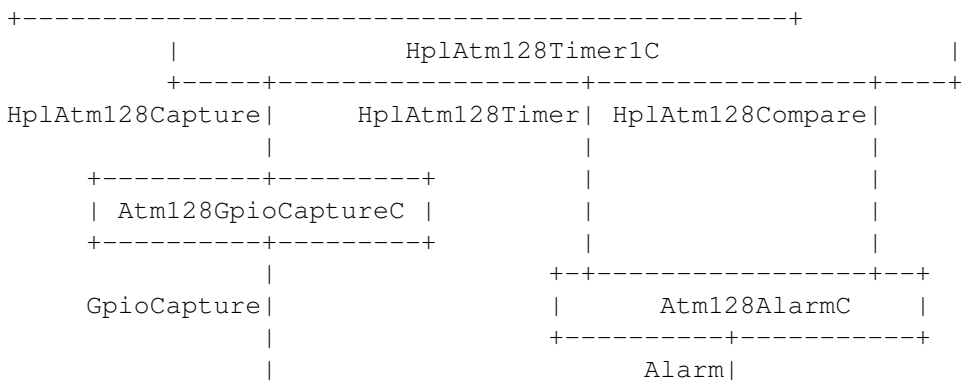
Implementing the timers for TinyOS in short consists of utilizing the available HW timer features to provide TinyOS style timers interfaces. In general the approach taken by existing platforms is to create a set of HPL level timer and control interfaces that export the available hardware features and adapt these to specific precisions and widths at the HAL level. However the specific HPL level design choices differ from platform to platform.

For example one could export all timers as a simple list, a component for each timer, a parameterised interface, etc. Similarly for the control features this could be provided with the timer interfaces or kept as a separate interface. In the following we will go through a few examples and point out their differences.

5.4.3 Example: ATMega128l

The ATMega128l implementation (atm128) exports the system timers one component each. Each component provide 4 interfaces - one for each of the functions of the unit: timer, capture, compare and control. The timer, capture, and compare interfaces are generic based on the width (either 8 or 16 bit) while two separate control interfaces are provided.

Depicted below is a diagram of how `HplAtm128Timer1C` could be stacked to provide platform independent interfaces (not shown are `HplAtm128Timer0Async`, `HplAtm128Timer2`, `HplAtm128Timer3`):



The hardware features is exported using 3 interfaces: `HplAtm128Timer`, `HplAtm128Compare` and `HplAtm128Capture`.

```
interface HplAtm128Timer<timer_size>
{
  async command timer_size get();
  async command void       set( timer_size t );
  async event void overflow();
  async command void reset();
  async command void start();
...

interface HplAtm128Compare<size_type>
{
  async command size_type get();
  async command void set(size_type t);
  async event void fired();           //<! Signalled on compare interrupt
  async command void reset();
...

interface HplAtm128Capture<size_type>
{
  async command size_type get();
  async command void      set(size_type t);
  async event void captured(size_type t);
  async command void reset();
```

In addition to the timer related interfaces two control interfaces are provided: 'HplAtm128TimerCtrl16' and 'HplAtm128TimerCtrl8' (below).

```
#include <Atm128Timer.h>

interface HplAtm128TimerCtrl8
{
  /// Timer control register: Direct access
  async command Atm128TimerControl_t getControl();
  async command void setControl( Atm128TimerControl_t control );

  /// Interrupt mask register: Direct access
  async command Atm128_TIMSK_t getInterruptMask();
  async command void setInterruptMask( Atm128_TIMSK_t mask);

  /// Interrupt flag register: Direct access
  async command Atm128_TIFR_t getInterruptFlag();
  async command void setInterruptFlag( Atm128_TIFR_t flags );
}
```

Each of the hardware timers are exported in one component for example 'HplAtm128Timer1P':

```
#include <Atm128Timer.h>

module HplAtm128Timer1P
{
  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 CompareA;
    interface HplAtm128Compare<uint16_t> as CompareB;
    interface HplAtm128Compare<uint16_t> as CompareC;
  }
  uses interface HplAtm128TimerCtrl8     as Timer0Ctrl;
}
implementation
{
  //=== Read the current timer value. ==================================
  async command uint16_t Timer.get() { return TCNT1; }

  //=== Set/clear the current timer value. =============================
  async command void Timer.set(uint16_t t) { TCNT1 = t; }
...
```

5.4.4 Example: MSP430

The MSP430 features two very similar (but not identical) timers. Both timers are provided through the same interfaces in a way similar to ATMega128l: 'Msp430Timer',

'Msp430Capture' 'Msp430Compare', 'Msp430TimerControl'. All timer interfaces (for both timers) are accessed through the component Msp430TimerC.

The `Msp430TimerControl` is show below. It is slightly different than the ATMega equivalent - notice that 'setControl' accepts a struct with configuration parameters instead of setting these with a command each.

```
#include "Msp430Timer.h"

interface Msp430TimerControl
{
  async command msp430_compare_control_t getControl();
  async command bool isInterruptPending();
  async command void clearPendingInterrupt();

  async command void setControl(msp430_compare_control_t control );
  async command void setControlAsCompare();
  async command void setControlAsCapture(uint8_t cm);

  async command void enableEvents();
  async command void disableEvents();
  async command bool areEventsEnabled();
}
```

Each timer is implemented through the generic component 'Msp430TimerP' that is instantiated for each timer in 'Msp430TimerC':

```
configuration Msp430TimerC
{
  provides interface Msp430Timer as TimerA;
  provides interface Msp430TimerControl as ControlA0;
  provides interface Msp430TimerControl as ControlA1;
  provides interface Msp430TimerControl as ControlA2;
  provides interface Msp430Compare as CompareA0;
  provides interface Msp430Compare as CompareA1;
  provides interface Msp430Compare as CompareA2;
  provides interface Msp430Capture as CaptureA0;
  provides interface Msp430Capture as CaptureA1;
  provides interface Msp430Capture as CaptureA2;
...
}
implementation
{
  components new Msp430TimerP( TAIV_, TAR_, TACTL_, TAIFG, TACLR, TAIE,
                TASSEL0, TASSEL1, FALSE ) as Msp430TimerA
           , new Msp430TimerP( TBIV_, TBR_, TBCTL_, TBIFG, TBCLR, TBIE,
                TBSSEL0, TBSSEL1, TRUE ) as Msp430TimerB
           , new Msp430TimerCapComP( TACCTL0_, TACCR0_ ) as Msp430TimerA0
           , new Msp430TimerCapComP( TACCTL1_, TACCR1_ ) as Msp430TimerA1
           , new Msp430TimerCapComP( TACCTL2_, TACCR2_ ) as Msp430TimerA2
...
```

5.4.5 Example: PXA27x

The PXA27x (XScale) platform provides a component for each of the timers classes on the system. Each component provides a few interfaces that exposes all features of that unit. Each interface combines timer and control interface features. For example 'HplPXA27xOSTimerC' provides 12 instances of the 'HplPXA27xOSTimer' interface and one instace 'HplPXA27xOSTimerWatchdog'. Similarly for 'HplPXA27xSleep' and 'HplPXA27xWatchdog'.

```
interface HplPXA27xOSTimer
{
  async command void setOSCR(uint32_t val);
  async command uint32_t getOSCR();
...


  async event void fired();
}
```

All the OS timers are exported through HplPXA27xOSTimerC

```
configuration HplPXA27xOSTimerC {

  provides {
    interface Init;
    interface HplPXA27xOSTimer as OST0;
    interface HplPXA27xOSTimer as OST0M1;
    interface HplPXA27xOSTimer as OST0M2;
    interface HplPXA27xOSTimer as OST0M3;
...
implementation {
  components HplPXA27xOSTimerM, HplPXA27xInterruptM;

  Init = HplPXA27xOSTimerM;

  OST0 = HplPXA27xOSTimerM.PXA27xOST[0];
  OST0M1 = HplPXA27xOSTimerM.PXA27xOST[1];
  OST0M2 = HplPXA27xOSTimerM.PXA27xOST[2];
  OST0M3 = HplPXA27xOSTimerM.PXA27xOST[3];
...
```

These interfaces are further abstracted through 'HalPXA27xOSTimerMapC' as a parameterised interface:

```
configuration HalPXA27xOSTimerMapC {
  provides {
    interface Init;
    interface HplPXA27xOSTimer as OSTChnl[uint8_t id];
  }
} implementation {
  components HplPXA27xOSTimerC;
```

```
  Init = HplPXA27xOSTimerC;

  OSTChnl[0] = HplPXA27xOSTimerC.OST4;
  OSTChnl[1] = HplPXA27xOSTimerC.OST5;
  OSTChnl[2] = HplPXA27xOSTimerC.OST6;
...
```

5.4.6 Timer Discussion

While the TEP is clear on many points there are few that are left up to the implementer. The relation between timers and capture events is unclear and covered in two TEPs. Capture refers to timers[TEP102] but the reverse is not clear[TEP102]. This has a few implications for the timer/capture relationship:

- Many devices feature multiple timers of the same width and precision. It is not clear how this is provided. In particular how this relates to a capture event - e.g. how does a capture event from timer 2 relate to a time value from timer 1. Presumably TinyOS has one system time based on a single timer which is used by all timer capture interfaces and visualized if necessary.
- The interfaces provide a an elegant way to expand a 16 bit timer to a 32 bit interface. However this is not the case for capture events.

5.5 I/O buses (UART, SPI, I2C)

Most modern microprocessors provide a selection of standard I/O bus units such as serial (UART or USART), I2C, SPI, etc. The drivers for the available buses are usually put in a sub directory for the MCU (e.g. *chip/atm128/spi* for SPI on ATMega128l).

There are a few TEP that cover different buses in TinyOS, and low level serial communication:

- TEP113 Serial Communication: Serial
- TEP117 Low-Level I/O: Serial, SPI, I2C

On some hardware platforms a single I/O unit is shared among a number of buses. The MSP430 for example implements SPI, I2C and USART in a single USART unit that can only operate in one mode at a time. In such cases the single hardware unit must be shared among the drivers.

5.5.1 Serial Communication

Serial communication in TinyOS 2.x is implemented as a serial communications stack with a UART or USART at the bottom. The bottom layer is covered in [TEP113] and [TEP117]. TEP117 states that each platform must provide access to a serial port through the 'SerialByteComm' interface from the component UartC. However UartC is not provided by any platforms at the time of writing. There seems to be a consensus to build the component 'PlatformSerialC' providing the 'UartStream' and 'StdControl' interfaces. Either component will be placed in the platform directory (e.g. *tos/platforms/mica2* for mica2). The serial stack is built upon 'PlatformSerialC' and this component will be required to take advantage of the serial stack.

[TEP117] defines 'UartStream' and 'UartByte' to access the a UART multiple bytes at a time and one byte at a time (synchronously), while [TEP113] defines 'UartByte' to send and receive one byte at a time (asynchronously).

5.5.2 I2C, SPI

SPI and I2C buses are provided through straightforward interfaces that match the functionality of the buses closely.

SpiByte, SpiPacket

```
interface SpiByte {
  async command uint8_t write( uint8_t tx );
}

interface SpiPacket {
  async command error_t send( uint8_t* txBuf, uint8_t* rxBuf, uint16_t len
  async event void sendDone( uint8_t* txBuf, uint8_t* rxBuf, uint16_t len,
                             error_t error );
}
```

I2CPacket

```
interface I2CPacket<addr_size> {
  async command error_t read(i2c_flags_t flags, uint16_t addr,
                             uint8_t length, uint8_t* data);
  async command error_t write(i2c_flags_t flags, uint16_t addr,
                              uint8_t length,  uint8_t* data);
  async event void readDone(error_t error, uint16_t addr,
                            uint8_t length, uint8_t* data);
  async event void writeDone(error_t error, uint16_t addr,
                             uint8_t length, uint8_t* data);
}
```

5.5.3 Example

..ATMega128 -> does not provide SerialByteComm

..MSP430 -> Uart1C provides SerialByteComm along with Init and StdControl -> sender det meste videre til Msp430Uart1C (som mangler Init!?!?) Msp430Uart0C -> SerialByteComm, Msp430UartControl, Msp430UartConfigure, Resurce -> Sender videre til Msp430Uart1P & Msp430Usart1C -> Sender videre til HplMsp430Usart1C->HplMsp430Usart1P

..PXA27 Parametriseret m. HplPXA27xUARTP (Init && HplPXA27xUART) -> init sætter en masse registre og enabler interrupt HalPXA27xSerialP: HplPXA27xUART, Init, noget DMA noget

6. Authors

Martin Leoold
University of Copenhagen, Dept. of Computer Science
Universitetsparken 1
DK-2100 København Ø
Denmark

Phone +45 3532 1464

email - leopold@diku.dk

7. Citations

[TEP1] TEP 1: TEP Structure and Keywords <http://www.tinyos.net/tinyos-2.x/doc/html/tep1.html>

[TEP2] TEP 2: Hardware Abstraction Architecture <http://www.tinyos.net/tinyos-2.x/doc/html/tep2.html>

[TEP3] TEP 3: Coding Standard <http://www.tinyos.net/tinyos-2.x/doc/html/tep3.html>

[TEP102] TEP 102: Timers <http://www.tinyos.net/tinyos-2.x/doc/html/tep102.html>

[TEP106] TEP 106: Schedulers and Tasks <http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html>

[TEP107] TEP 107: TinyOS 2.x Boot Sequence <http://www.tinyos.net/tinyos-2.x/doc/html/tep107.html>

[TEP109] TEP 109: Sensors and Sensor Boards <http://www.tinyos.net/tinyos-2.x/doc/html/tep109.html>

[TEP111] TEP 111: message_t <http://www.tinyos.net/tinyos-2.x/doc/html/tep111.html>

[TEP113] TEP 113: Serial Communication <http://www.tinyos.net/tinyos-2.x/doc/html/tep113.html>

[TEP117] TEP 117: Low-Level I/O <http://www.tinyos.net/tinyos-2.x/doc/html/tep117.html>

[TEP121] TEP 121: Towards TinyOS for 8051 <http://www.tinyos.net/tinyos-2.x/doc/html/tep121.html>

[TOSPRG] Philip Levis: TinyOS Programming <http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf>

[tos2.0view] Philip Levis. "TinyOS 2.0 Overview" *Feb 8 2006* <http://www.tinyos.net/tinyos-2.x/doc/html/overview.html>

[nescman] David Gay, Philip Levis, David Culler, Eric Brewer. "NesC 1.2 Language Reference Manual" *August 2005*

[TUT10]

TinyOS 2 Tutorial Lesson 10 <http://www.tinyos.net/tinyos-2.x/doc/html/tutorial/lesson10.html>

LocalWords: TinyOS TEP TEPs nesC