Coding Standard

TEP:	3
Group:	TinyOS 2.0 Working Group
Type:	Best Current Practice
Status:	Draft
TinyOS-Version:	2.x
Author:	Ion Yannopoulos, David Gay
Draft-Created:	31-Dec-2004
Draft-Version:	1.6
Draft-Modified:	2009-12-16
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Note

This document specifies a Best Current Practices 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].

Contents

1 Introduction

The purpose of a naming convention is twofold:

- To avoid collisions which prevent compilation or lead to errors. In TinyOS the most important place to avoid such collisions is in interface and component names.
- To enable readers of the code to identify which names are grouped together and which packages they are defined in.

Remember that code that is useful will end up being read far more often than it is written. If you deviate from the suggestions or requirements below, be consistent in how you do so. If you add any new conventions to your code, note it in a README.

2 General Conventions

2.1 General

- Avoid the use of acronyms and abbreviations that are not well known. Try not to abbreviate "just because".

- Acronyms should be capitalized (as in Java), i.e., Adc, not ADC. Exception: 2-letter acronyms should be all caps (e.g., AM for active messages, not Am)
- If you need to abbreviate a word, do so consistently. Try to be consistent with code outside your own.
- All code should be documented using *nesdoc* [nesdoc], *Doxygen* [Doxygen] or *Javadoc* [Javadoc]. Ideally each command, event and function has documentation. At a bare minimum the interface, component, class or file needs a paragraph of description.
- If you write code for a file, add an *@author* tag to the toplevel documentation block.

3 Packages

For the purposes of this document a package is a collection of related source and other files, in whatever languages are needed. A package is a logical grouping. It may or may not correspond to a physical grouping such as a single directory. In TinyOS a package is most often a directory with zero or more subdirectories.

nesC and C do not currently provide any package support, thus names of types and components in different packages might accidentally clash. To make this less likely, judiciously use prefixes on groups of related files (often, but not always, part of a single package). See the examples below.

In a package, we distinguish between public components (intended to be used and wired outside the package) and private components (only used and wired within the package). This distinction is not enforced by nesC.

3.1 Directory structure

- Each package should have it's own directory. It may have as many subdirectories as are necessary.
- The package's directory should match the package's prefix (if it uses one), but in lower-case.
- The default packages in a TinyOS distribution are:
  - ***tos/system/*. Core TinyOS components. This directory's** components are the ones necessary for TinyOS to actually run.
  - ***tos/interfaces/*. Core TinyOS interfaces, including** hardware-independent abstractions. Expected to be heavily used not just by *tos/system* but throughout all other code. *tos/interfaces* should only contain interfaces named in TEPs.
  - ***tos/platforms/*. Contains code specific to mote platforms, but** chip-independent.
  - ***tos/chips/*. Contains code specific to particular chips and to** chips on particular platforms.
  - ***tos/lib/*. Contains interfaces and components which extend the** usefulness of TinyOS but which are not viewed as essential to its operation. Libraries will likely contain subdirectories.

  - ***apps/, apps/demos, apps/tests, apps/tutorials.*** **Contain** applications with some division by purpose. Applications may contain subdirectories.
- It is not necessary that packages other than the core break up their components and their interfaces. The core should allow overrides of components fairly easily however.
- Each directory should have a README.txt describing its purpose.

4 Language Conventions

4.1 nesC convention

4.1.1 Names

- All nesC files must have a *.nc* extension. The nesC compiler requires that the filename match the interface or component name.
- Directory names should be lowercase.
- Interface and component names should be mixed case, starting upper case.
- All public components should be suffixed with 'C'.
- All private components should be suffixed with 'P'.
- Avoid interfaces ending in 'C' or 'P'.
- If an interface and component are related it is useful if they have the same name except for the suffix of the component.
- Commands, events, tasks and functions should be mixed case, starting lower case.
- Events which handle the second half of a split-phase operation begun in a command should have names that are related to the commands. Making the command past tense or appending *'Done'* are suggested.
- Constants should be all upper case, words separated by underscores. - Use of *#define* for integer constants is discouraged: use *enum*.
- Type arguments to generic components and interfaces should use the same case as C types: all lower-case separated by underscores, ending in '_t'.
- Module (global) variables should be mixed case, starting lower case.

4.1.2 Packages

- Each package may use a prefix for its component, interface and global C names. These prefixes may sometimes be common to multiple packages. Examples:
  - All hardware presentation layer names start with Hpl (this is an example of a shared prefix).
  - Chip-specific hardware abstraction layer components and interfaces start with the chip name, e.g., Atm128 for ATmega128.

  - The Maté virtual machine uses the Mate to prefix all its names.
  - Core TinyOS names (e.g., the timer components, the Init interface) do not use a prefix.
- Some packages may use multiple prefixes. For instance, the ATmega128 chip package uses an Hpl prefix for hardware presentation layer components and Atm128 for hardware abstraction layer components.

4.1.3 Preprocessor

- Don't use the nesC *includes* statement. It does not handle macro inclusion properly. Use *#include* instead.
- Macros declared in an *.nc* file must be *#define*'d after the *module* or *configuration* keyword to actually limit their scope to the module.
- Macros which are meant for use in multiple *.nc* files should be *#define*'d in a *#include*'d C header file.
- Use of macros should be minimized: *#define* should only be used where *enum* and *inline* do not suffice.
  - Arguments to *unique()* should be *#define* string constants rather than strings. This minimizes nasty bugs from typos the compiler can't catch.

4.2 C Convention

- All C files have a .h (header) or (rarely) a .c (source) extension.
  - Filenames associated with a component should have the same name as the component.
  - Filenames of a package should have a name with the package prefix (if any).
  - Filenames which are not associated with a component should be lowercase.
- C does not protect names in any way. If a package uses a prefix, it should also use it for all types, tags, functions, variables, constants and macros. This leads naturally to:
  - Minimize C code outside of nesC files. In particular: most uses of hardware specific macros in TinyOS 1.x should be replaced with nesC components in TinyOS 2.x.
- C type names (define with *typedef*) should be lower case, words separated by underscores and ending in *'_t'*.
- C tag names (for *struct*, *union*, or *enum*) should be lower case, words separated by underscores. Types with tag names should provide a typedef.
- C types which represent opaque pointers (for use in parameters) should be named similar to other types but should end in *'_ptr_t'*.
- Functions should be lower case, words separated by underscores.

- Function macros (*#define*) should be all upper case, words separated by underscores.
  - Using function macros is discouraged: use *inline* functions.
- Constants should be all upper case, words separated by underscores. - Use of *#define* for integer constants is discouraged: use *enum*.
- Global variables should be mixed case, starting lower case.

4.3 Java convention

- The standard Java coding convention [Java_Coding_Convention] should be followed.
- All core TinyOS code is in the package *net.tinyos*.

4.4 Other languages

- No established conventions.

5 TinyOS Conventions

TinyOS also follows a number of higher-level programming conventions, mostly designed to provide a consistent "look" to TinyOS interfaces and components, and to increase software reliability.

5.1 Error returns

TinyOS defines a standard error return type, `error_t`, similar to Unix's error returns, except that error codes are positive:

```
enum {
  SUCCESS         = 0,
  FAIL            = 1,
  ESIZE           = 2, // Parameter passed in was too big.
  ...
};
```

`SUCCESS` represents successful execution of an operation, and `FAIL` represents some undescribed failure. Operations can also return more descriptive failure results using one of the Exxx constants, see the `tos/types/TinyError.h` file for the current list of errors.

The `error_t` type has a combining function to support multiple wiring of commands or events retuning `error_t`, defined as follows:

```
error_t ecombine(error_t r1, error_t r2) { return r1 == r2 ? r1 : FAIL; }
```

This function returns `SUCCESS` if both error returns are `SUCCESS`, an error code if they both return the same error, and `FAIL` otherwise.

Commands that initiate a split-phase operation SHOULD return `error_t` if the operation may be refused (i.e., the split-phase event may not be signaled under some conditions). With such functions, the split-phase event will be signaled iff the split-phase command returns `SUCCESS`.

5.2 Passing pointers between components

Sharing data across components can easily lead to bugs such as data races, overwriting data, etc. To minimise the likelyhood of these occurrences, we discourage the use of pointers in TinyOS interfaces.

However, there are circumstances where pointers are necessary for efficiency or convenience, for instance when receiving messages, reading data from a flash chip, returning multiple results, etc. Thus we allow the use of pointers within interfaces as long as use of those pointers follows an "ownership" model: at any time, only one component may refer to the object referenced by the pointer. We distinguish two cases:

- Ownership transferred for the duration of a call: in the following command:

  ```
  command void getSomething(uint16_t *value1, uint32_t *value2);
  ```

  we are using pointers to return multiple results. The component implementing getSomething MAY read/write `*value1` or `*value2` during the call and MUST NOT access these pointers after getSomething returns.

- Permanent ownership transfer: in the following split-phase interface:

  ```
  interface Send {
    command void send(message_t *PASS msg);
    event void sendDone(message_t *PASS msg);
  }
  ```

  components calling send or signaling `sendDone` relinquish ownership of the message buffer. For example, take a program where component A uses the `Send` interface and B provides it. If A calls `send` with a pointer to `message_t` *x*, then ownership of *x* passes to B and A MUST NOT access *x* while B MAY access *x*. Later, when B signals the `sendDone` event with a pointer to *x* as parameter, ownership of *x* returns to A and A MAY access *x*, while B MUST NOT access *x*.

  If an interface with `PASS` parameters has a return type of `error_t`, then ownership is transferred iff the result is `SUCCESS`. For instance, in

  ```
  interface ESend {
    command error_t esend(message_t *PASS msg);
    event void esendDone(message_t *PASS msg, error_t sendResult);
  }
  ```

  ownership is transferred only if `esend` returns `SUCCESS`, while ownership is always transferred with `esendDone`. This convention matches the rule for signaling split-phase completion events discussed above.

  `PASS` is a do-nothing macro defined as follows:

  ```
  #define PASS
  ```

In the future, some tool may check that programs respect these ownership transfer rules.

5.3 Usage of wiring annotations

TinyOS checks constraints on a program's wiring graph specified by annotations on a component's interfaces. Wiring constraints are specified by placing `@atmostonce()`, `@atleastonce()` and `@exactlyonce()` attributes on the relevant interfaces. For instance, writing

```
module Fun {
  provides interface Init @atleastonce();
...
```

ensures that programs using module `Fun` must wire its `Init` interface at least once.

The `@atleastonce()` and `@exactlyonce()` annotations SHOULD be used sparingly, as they can easily prevent modularising subsystem implementations, which is undesirable. However, the `@atleastonce()` annotation SHOULD be used on initialisation interfaces (typically, the `Init` interface) in modules, to prevent the common bug of forgetting to wire initialisation code.

6 Citations

[TEP_1] TEP 1 <http://www.tinyos.net/working_groups/tinyos-2.0wg/teps/tep-1.html>

[TEP_2] TEP 2 <http://www.tinyos.net/working_groups/tinyos-2.0wg/teps/tep-2.html>

[Doxygen] Doxygen <http://www.doxygen.org>

[Java_Coding_Convention] Java Coding Convention <http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html>

[JavaDoc] JavaDoc <http://java.sun.com/j2se/javadoc>

[nesdoc] nesdoc <http://www.tinyos.net/tinyos-1.x/doc/nesc/nesdoc.html>