NesC Overview (and Summary of Changes)

Eric Brewer, David Gay, Phil Levis, and Rob von Behren
June 17, 2002
Version 0.6

Overview

The NesC language has two goals: first, to be a pragmatic low-level language for programming the motes, and second to be the intermediate language for future higher-level languages. Here we will only deal with the first goal.

As a programming language, this first version of NesC has a few modest goals:
- Get rid of the awful macros for frame variables, commands and events.
- Eliminate wiring error through type checking.
- Simplify wiring by grouping logically related “wires” into interfaces.
- Increase modularity through the separation of interfaces and modules.
- Clean up the code base through better use of types and more compile time checking.
- Better performance by enabling whole program optimization.

Other positive side effects:
- Reduced the number of files per module.
- Reduced compilation time (and removed a lot of shaky perl code)
- Reduced code size by 25%
- Better looking code
- Easier to debug code

Files

In general, we have moved from three files per module to one plus some interfaces. The main file, the `.td` file, defines a *component*, which is either a *configuration* or a *module*. Both configurations and modules have a specification of the required and provided interfaces. A configuration also lists the components used and the internal wiring. A module provides code that implements the provided interfaces (and may use the required interfaces).

We keep these two separate in the belief that it makes it easier to understand the wiring and to have multiple implementations of a submodule (that use the same wiring). The slightly negative side effect of this is that components such as `Blink` that contain both wiring and code, become two components: `Blink`, which contains the definition and internal wiring, and `BlinkM`, which contains the code. (M stands for module.) Note this is still an improvement in file count, as we moved from three (`Blink.comp`, `Blink.c` and `Blink.desc`) to two (`Blink.td` and `BlinkM.td`).

Here is the mapping:

Old Style		NesC	
`.comp`	Defines the component, but not its implementation	`.td`	Define the whole module including the submodules, the interfaces used and provided, and the implementation.
`.c`	Defines the implementation		
`.desc`	Defines the wiring		
`.h`	Defines the shared types	`.th`	Defines the shared types (but no C preprocessor)
`--`		`.ti`	Defines an interface

Although the above implies that three old files map onto one `.td` file, the actual mapping is more complex than that:

Old Style	NesC	
`.desc (only)`	`.td`	Configuration with empty spec
`.desc and .comp`	`.td`	Configuration
`.c and .comp`	`.td`	Module
`.c, .desc and .comp`	`foo.td` `fooM.td`	Configuration and Submodule

Interfaces

We use interfaces to simplify the wiring together of modules, as well as to simplify interposition and modularity. They are similar to Java interfaces except there is no inheritance and they are bidirectional, although is in practice a primary direction, usually commands down and events back.

Interfaces have two goals. First, interfaces group together related commands and events. This simplifies wiring (one “wire” per interface rather than one per command/event), reduces wiring errors, and improves documentation. The second goal is modularity; interfaces make it much easier to mix and match modules, maintain multiple versions, and perform interposition.

The is some art to converting old modules to NesC, in that one must decide how to group the existing command and events into interfaces. The good news is that many of the key interfaces are already defined, such as those for communication and the clock. In general, you should subdivide a group into two interfaces if it would make sense to use only one of the two. Otherwise, bigger groups are simpler to understand and use.

Not every command/event has to be in an interface. There are a few cases where there are isolated commands that do not fit with any of the interfaces. These end up essentially as global commands (just as they were before). In general, it is better to partition the commands and events into interfaces.

Tasks are not part of an interface, as they are viewed as internal to a module. Obviously, it is easy to create commands that post tasks.

There are three ways to wire interfaces together:

`caller.foo -> callee.foo`	`caller` requires interface `foo`, which is provided by `callee`
`callee.foo <- caller.foo`	Same as above
`top.foo = sub.foo`	`top` provides/requires interface `foo`, which is mapped to the `foo` interface of submodule `sub.` For example, you use "=" when both `top` claims to provide the interface but it is really just passed through to sub. Similarly if both require the same interface. It is essentially renaming `sub.foo`.

As a shortcut, you can omit the interface name on but not both sides of the wiring, since the compiler can infer it (if there is exactly one such interface in that component).

Example Translation: Blink

Here is the wiring file for Blink (`Blink.td`), with some annotations.

```
configuration Blink {
}
```

Module does not provide or require anything, since it is a whole application.

```
implementation {
   uses Main, BlinkM, Clock, Leds;
```

List of the included modules

```
   Main.SimpleInit -> BlinkM.SimpleInit;
```

`Main` requires interface `SimpleInit`, which is provided by `BlinkM` (was 2 wires). See the appendix for the old wiring file.

```
   BlinkM.Clock -> Clock;
```

`BlinkM` requires the `Clock` (was 2 wires). Note that we omitted the name of the interface in the Clock module (rather than `Clock.Clock`).

```
   BlinkM.Leds -> Leds;
}
```

Connect to `Leds` (was 6 wires).

The code for `Blink` resides in `BlinkM.td`, listed below.

```
module BlinkM {
  provides {
    interface SimpleInit;
  }
  requires {
    interface Clock;
    interface Leds;
  }
}
```

The code for blink implements the `SimpleInit` interface and uses `Clock` and `Leds`.

```
implementation {
  bool state;
```

The implementation has one frame variable, `state`, which uses the new `bool` type.

```
  command result_t SimpleInit.init() {
    state = FALSE;
    return SUCCESS;
  }
```

Note the direct use of the frame variable. `result_t` is the always the result type for a command.

```
  command result_t SimpleInit.start() {
    return call Clock.setRate(128, 6);
  }
```

The call uses the interface name and the command name. The arg used to be “tick1ps”, which was a `#define` to be “128, 6” [hack].

```
  event result_t Clock.fire() {
    state = !state;

    if (state)
      call Leds.redOn();
    else
      call Leds.redOff();

    return SUCCESS;
  }
}
```

Define an event handler; events also have return type `result_t`. The `Clock` module signals the `fire` event.

Summary of Changes

Files

The old file types are not used at all; see above for the usage of the new files.

Types

We define several base types that should be used:

`result_t`	The return type of events and commands (8-bit unsigned int). It is possible to use a different return type.
`bool`	Basic boolean type; it is an 8-bit `enum {false, true}`
`uint8_t` `uint16_t`, etc.	Fully specified types to replace “char” and “int”, which are signed by default only partially defined.

No C Preprocessor

The current version of NesC does not support the C preprocessor (on .td, .th and .ti files). On the good side this prevents the use of those scary macros in the old version. On the bad side:

- Use enums instead of `#define` for constants. Shared constants can be put in .th files and included in multiple modules.
- Use the new `bool` type.

Frame Variables

Frame variables are easier to define and use. They are just defined at top of the implementation part of the .td file, and translate to a global variable with the name of the module prepended, e.g., in `BlinkM`, frame variable `state` becomes `BlinkM_state`. It is possible to have name conflicts because of this, although it is unlikely. The most common case of conflicts occurs when there is a local variable with the same name as the frame variable, since that used to work; to check for this compile with -Wshadow, which will detect the conflict. Note there is no need to use the `TOS_FRAME_BEGIN`/`TOS_FRAME_END` macros (in fact it is an error to do so).

To use the variable you can just reference it normally. (It can be shadowed by local variables.) There is no need to use the “`VAR(x)`” syntax.

Commands. Events and Tasks

It is easier to use define and use commands, events and tasks. Here is a comparison of the definition and use of commands, events and tasks:

Old Style	NesC
`char TOS_COMMAND(foo))(args) {`	`command result_t <`*`interface`*`>.foo(args) {`
`TOS_CALL_COMMAND(foo)(args)`	`call <`*`interface`*`>.foo(args)    [returns result_t]`
`void TOS_EVENT(foo)(args) {`	`event result_t <`*`interface`*`>.foo(args) {`
`TOS_SIGNAL_EVENT(foo)(args)`	`signal <`*`interface`*`>.foo(args)`
`TOS_TASK(foo) {`	`task void foo() {`

Old Style	NesC
`TOS_POST_TASK(foo);`	`post foo();`

For commands/events that are not part of an interface, the <interface> part is absent.

Miscellaneous

- The meaning of "`<type> foo()`" has changed for a procedure definition. It used to mean "old-style" C definition, which means that the number of arguments is not defined. It now means "`<type> foo(void)`" -- that is, the procedure is known to have no args. In addition, we do not support "`<type> foo(void)`", since it is now equivalent to "`<type> foo()`". [This latter restriction seems unnecessary.]
- The debugging macro has changed from "`dbg(x, (y))`" to "`dbg(x, y)`". It is now a function call, but we will inline it as needed. This is much cleaner, but does break the old usage.

Naming Convention

These are derived from Java.

- Files, components, interfaces, typedefs, struct/enum/union types: Format is caps for first letter of each word including the first, e.g. CntToLeds.
- Variables: Caps for the first letter of each word, except for the first word, which is lower case, e.g. sendBuffer.
- Constants: All caps with underscores to separate words, e.g. RED_BIT.

Appendix A: Previous Wiring Code for Blink (`Blink.desc`)

```
include modules{
  MAIN;
  BLINK;
  CLOCK;
  LEDS;
};

BLINK:BLINK_INIT MAIN:MAIN_SUB_INIT
BLINK:BLINK_START MAIN:MAIN_SUB_START

BLINK:BLINK_LEDy_on LEDS:YELLOW_LED_ON
BLINK:BLINK_LEDy_off LEDS:YELLOW_LED_OFF
BLINK:BLINK_LEDr_on LEDS:RED_LED_ON
BLINK:BLINK_LEDr_off LEDS:RED_LED_OFF
BLINK:BLINK_LEDg_on LEDS:GREEN_LED_ON
BLINK:BLINK_LEDg_off LEDS:GREEN_LED_OFF
BLINK:BLINK_SUB_INIT CLOCK:CLOCK_INIT
BLINK:BLINK_CLOCK_EVENT CLOCK:CLOCK_FIRE_EVENT
```