nesC Language Reference Manual

David Gay [add your names here as you edit]

August 2002

1 Introduction

nesC is a new programming language designed to embody the structuring concepts and execution model of TinyOS [?]. [Someone: insert the standard 2-sentence TinyOS/sensor network desc]. TinyOS has now been reimplemented in nesC.

The basic concepts behind nesC are:

- Program decompisition into *components*, which can be assembled ("wired") to form whole programs.
- Specification of component behaviour as a set of *interfaces* provided or used by the component. The provided interfaces are intended to represent the functionality that the component provides to its user, the used interfaces represent the functionality the component needs to perform its job.
- Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider ("commands") and a set to be implemented by the interface's user ("events"). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens).
- Components are statically linked to each other via their interfaces. This increases runtime efficiency and allows for better static analysis of program's.
- nesC is designed under the expectation that code will be generated by whole-program compilers. This should also allow for better code generation and analysis.

This document is a reference manual for nesC rather than a tutorial. The TinyOS tutorial [?] presents a gentler introduction to nesC.

2 Notation

The `typewriter` font is used for nesC code and for filenames.

The grammar of nesC is an extension the ANSI C grammar. We chose to base our presentation on the ANSI C grammar from Appendix A of Kernighan and Ritchie (K&R) [?, pp234–239]. We will not repeat productions from that grammar here. Words in *italics* are non-terminals and non-literal terminals, `typewriter` words and symbols are literal terminals. The subscript *opt* indicates optional terminals or non-terminals. In some cases, we change some ANSI C grammar rules. We indicate this as follows: *also* indicates additional productions for existing non-terminals, *replaced by* indicates replacement of an existing non-terminal. In the piecemeal presentation of the grammar, we sometimes use . . . to represent elided productions (not relevant to the construction being presented).

3 Concepts and Terminology

We use the following terminology in the rest of the reference manual:

- *component*: The basic unit of nesC programs. Components have a name and are of two kinds: *modules* and *configurations*. A component has a *specification* and an implementation.
- *specification*: A list of *interface instances*, *commands* and *events* that are *provided* or *used.*
- *specification element*: An *interface instance*, *command* and *event* in a specification.
- *module*: A component whose implementation is specified by C code.
- *configuration*: A component whose implementation is specified by *wiring* other components.
- *interface type*: An *interface type* has a name and specifies the interaction between two components, *the provider* and the *user.* This specification takes the form of a set of named functions called *commands* and *events.*
- *command*: A function that must be implemented by the provider of an interface.
- *event*: A function that must be implemented by the user of an interface.
- *interface instance*: An instance of a particular interface type in the specification of a component. An interface instance has a name, a *role* (provider or user), an interface type and, optionally, *interface parameters.* An interface instance without parameters is a *simple interface instance*, with parameters it is a *parameterised interface instance.*
- *interface parameter*: An interface parameter must be of integral type. There is (conceptually) a separate simple interface instance for each distinct list of parameter values of a parameterised interface instance.
- *interface*: When the context is unambiguous, we will say interface to refer to either an interface type or an interface instance.
- *task*: A TinyOS task.
- *scopes*: nesC has the standard C global, per-block and per-function scopes. Additionally there is are per-component-specification, per-component-implementation and per-interface-type scopes.
- *extents*: nesC has the standard C extents: indefinite, per-function, and per-block.
- *compile-time error*: An error that the nesC compiler must report at compile-time.

4 Interfaces

Interfaces in nesC are bidirectional: they specify the interaction of two components. An interface has two "sides": the provider and the user. The interface specifies a set of functions, called commands, to be implemented by the interface's provider and a set of functions, called events, to be implemented by the interface's user. This section explains how interfaces are specified, Section 5 explains how components specify the interfaces they provide and use, Section 6 explains how commands and events are called from C code and Section 7 explains how components are linked together via their interfaces.

Interfaces are specified by interface types, as follows:

nesC-file:
: *includes-list*$_{opt}$ *interface*
: ...

interface:
: `interface` *identifier* { *declaration-list* }

storage-class-specifier: also one of
: `command` `event`

This declares interface type *identifier*. The *declaration-list* must consist of function declarations with the `command` or `event` storage class (if not, a compile-time error occurs). An interface can optionally include C files via the *includes-list* (see Section 8).

A simple interface is:

```
interface SendMsg {
  command result_t send(uint16_t address, uint8_t length, TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

Provides of the `SendMsg` interface type must implement `send`, while users must implement `sendDone`.

5 Component Specification

A nesC component is either a module or a configuration:

nesC-file:
: *includes-list*$_{opt}$ *module*
: *includes-list*$_{opt}$ *configuration*
: ...

module:

`module` *identifier specification module-implementation*

configuration:
`configuration` *identifier specification configuration-implementation*

Component's names are specified by the *identifier.* A component can optionally include C files via the *includes-list* (see Section 8).

The *specification* lists the specification elements (interface instances, commands or events) used or provided by this component. Each specification element has a name, a compile-time error occurs if two specification elements have the same name:

specification:
{ *uses-provides-list* }

uses-provides-list:
uses-provides
uses-provides-list uses-provides

uses-provides:
`uses` *specification-element-list*
`provides` *specification-element-list*

specification-element-list:
specification-element
{ *specification-elements* }

specification-elements:
specification-element
specification-elements specification-element

There can be multiple `uses` and `provides` directives. Multiple used or provided specification elements can be grouped in a single directive by surrounding them with { and }. For instance, the following two specifications are identical:

```
module A1 {
  uses interface X;
  uses interface Y;
} ...
```

```
module A1 {
  uses {
    interface X;
    interface Y;
  }
} ...
```

An interface instance is specified as follows:

specification-element:
`interface` *renamed-identifier parameters*$_{opt}$

...

renamed-identifier:
    identifier
    identifier `as` *identifier*

parameters:
    `[` *parameter-type-list* `]`

The complete syntax for interface instance declaration is `interface X as Y`, explicitly specifying `Y` as the instance's name. The `interface X` syntax is a shorthand for `interface X as X`.

If the *parameters* are omitted, then `interface X as Y` declares a single interface to this component. If the *parameters* are present (e.g., `interface SendMsg[uint8_t id]`) then this interface instance declares multiple interfaces to this component, one for each distinct list of parameter values (so there are 256 `SendMsg` interfaces in the previous example). The types of the *parameters* must be integral types (`enum`s are not allowed). [why no enums?]

It is also possible to provide commands and events directly, rather than via interfaces (but this facility should only be used occasionally as interfaces provide specification reuse). A command or event is specified, as in interface types, as a standard C function declaration with `command` or `event` as its storage class specifier.

specification-element:
    declaration
    ...

storage-class-specifier: also one of
    `command` `event`

It is a compile-time error if the *declaration* is not a function declaration with the `command` or `event` storage class.

As with interface instances, commands and events can have integral parameters. These are specified before the function's argument list, e.g., `command void send[uint8_t id](int x)`:

direct-declarator: also
    direct-declarator parameters `(` *parameter-type-list* `)`

Note that parameters are only allowed inside component specifications, not inside interface types.

Here is a full specification example:

```
configuration GenericComm {
  provides {
    interface StdControl as Control;
    interface SendVarLenPacket as UARTSendRawBytes;
    // The interface are parameterised by the active message id
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];
  }
  uses {
    // signaled after every send completion for components which wish to
    // retry failed sends
    event result_t sendDone();
  }
} ...
```

In this example, `GenericComm`:

- Provides simple interface instance `Control` of type `StdControl`.
- Provides simple interface instance `UARTSendRawBytes` of type `SendVarLenPacket`.
- Provides parameterised instances of interface type `SendMsg` and `ReceiveMsg`; the parameterised instances are named `SendMsg` and `ReceiveMsg` respectively.
- Uses event `sendDone`.

6 Modules

Modules implement a component specification with C code:

module-implementation:
          `implementation` { *translation-unit* }

The name space inside a module has the following hierarchical structure:

- The outermost scope (the C global scope) contains the C declarations (see Section 8).
- The next scope (a per-component-specification scope) contains the specification element names from the module's specification.
- The next scope (a per-component-implementation scope) contains the top level declarations of the module's *translation-unit*. These declarations have indefinite extent and can be: any standard C declaration or definition, a TinyOS task declaration or definition, a commands or an event implementation.
- The remaining scopes are the usual C per-function and per-block scopes.

A command or event F of an interface instance X is named $X.F$ inside a module. It is a compile-time error if command or event F does not exist in X.

6.1 Implementing the Module's Specification

The *translation-unit* must implement the commands and events required by the module's specification:

- If a module provides an interface instance X of type Y, and interface type Y has a command C, then the module's implementation must specify a C implementation of a command called $X.C$.
- If a module uses an interface instance X of type Y, and interface type Y has an event E, then the module's implementation must specify a C implementation of an event called $X.E$.
- If a module provides a command (or event) F, then the module's implementation must specify a C implementation for a command (or event) called F.

Note that a module that provides or uses an interface can call any of the interface's commands and signal any of the interface's events.

These command and event declarations are made with the following C syntax extensions:

storage-class-specifier: also one of
    `command` `event`

declaration-specifiers: also
    `default` *declaration-specifiers*

direct-declarator: also
    identifier `.` *identifier*
    direct-declarator parameters `(` *parameter-type-list* `)`

The declaration of command or event $X.F$ of simple interface instance X has the syntax of a C function declaration of $X.F$ with storage class `command` or `event`. For example, in a module that provides interface `Send` of type `SendMsg`:

```
command result_t Send.send(uint16_t address, uint8_t length, TOS_MsgPtr msg) {
  ...
  return SUCCESS;
}
```

The declaration of command or event $X.F$ of parameterised interface instance X, with parameters P has the syntax of a C function declaration of $X.F$ with storage class `command` or `event` where the function's arguments are prefixed with the parameters P. For example, in a module that provides interface `Send[uint8_t id]` of type `SendMsg`:

```
command result_t Send.send[uint8_t id](uint16_t address, uint8_t length,
                                       TOS_MsgPtr msg) {
  ...
  return SUCCESS;
}
```

Compile-time errors are reported when:

- Any commands or events that must be implemented are missing.
- The type signature (and optional parameters) of a command or event does not match that given in the module's specification.

6.2 Calling Commands and Signaling Events

The following extensions to C syntax are used to call events and signal commands:

postfix-expression:
  postfix-expression `[` *argument-expression-list* `]`
  call-kind$_{opt}$ *primary* `(` *argument-expression-list*$_{opt}$ `)`
  . . .

call-kind: one of
  `call signal post`

A command C with no parameters is called with `call` C`(...)`, an event E is signaled with `signal` E`(...)`. For instance, in a module that uses interface `Send` of type `SendMsg`: `call Send.send(1, sizeof(Message), msg1)`.

A command C with parameters P is called with `call` C`[`P`](...)`, an event E is signaled with `signal` E`[`P`](...)`. For instance, in a module that uses interface `Send[uint8_t id]` of type `SendMsg`: `call Send.send[12](1, sizeof(Message), msg1)`.

Execution of commands and events is immediate, i.e., `call` and `signal` behave similarly to function calls. Section 7.4 explains what function(s) are actually executed and what result gets returned.

A module can specify a default implementation for a command that it uses or an event that it provides [note: need to define this terminology]. These default implementations are called when the command or event is not connected by any configuration component. Section 7.4 explains when default implementations are called. A default command or event is defined by prefixing a command or event definition with the `default` keyword:

declaration-specifiers: also
  `default` *declaration-specifiers*

For instance, in a in a module that uses interface `Send` of type `SendMsg`:

```
default command result_t Send.send(uint16_t address, uint8_t length,
                                   TOS_MsgPtr msg) {
  return SUCCESS;
}
/* call is allowed even if interface Send is not connected */
... call Send.send(1, sizeof(Message), &msg1) ...
```

6.3 Tasks

A TinyOS task is defined as a function of storage clas `task` returning `void` and with no arguments: `task void myTask() { ... }`.[1] Tasks are posted by prefixing a call to the task with `post`, e.g., `post myTask()`. A task can also have a forward declaration, e.g., `task void myTask();`.

storage-class-specifier: also one of
    `task`

call-kind: also one of
    `post`

7 Configurations

Configurations implement a component specification by connecting, or wiring, other components:

configuration-implementation:
    `implementation` { *component-list*$_{opt}$ *connection-list* }

The *component-list* lists the components that are used to build this configuration, the *connection-list* specifies how these components are wired together and to the configuration's specification.

The name space inside a module has the following hierarchical structure:

- The outermost scope (the C global scope) contains the C declarations (see Section 8).
- The next scope (a per-component-specification scope) contains the specification element names from the configuration's specification.
- The next scope (a per-component-implementation scope) contains the names of the components used in this configuration.

In the rest of this section, we call specification elements from the configuration's specification *external*, and specification elements from one of the configuration's components *internal*.

7.1 Included components

The *component-list* specifies the components used to build this configuration. These components can be optionally renamed within the configuration, either to avoid name conflicts with the configuration's specification elements, or to simplify changing the components a configuration uses (to avoid having to change the wiring):

[1] nesC functions with no arguments are declared with `()`, not `(void)`. See Section 8.

component-list:
    components
    component-list components

components:
    `components` *component-line* `;`

component-line:
    renamed-identifier
    component-line `,` *renamed-identifier*

renamed-identifier:
    identifier
    identifier `as` *identifier*

A compile-time error occurs if two components are given the same name using `as` (e.g., `components X, Y as X`).

There is only ever a single instance of a component: if a component X is used in two different configurations (or even twice within the same configuration) there is still only instance of X (and its variables) in the program.

7.2 Wiring

Wiring is used to connect specification elements (interfaces, commands, events) together. Taken together, all the wiring statements of a particular program determine which functions get called at each `call` and `signal` statement:

connection-list:
    connection
    connection-list connection

connection:
    endpoint `=` *endpoint*
    endpoint `->` *endpoint*
    endpoint `<-` *endpoint*

endpoint:
    identifier-path
    identifier-path `[` *argument-expression-list* `]`

identifier-path:
    identifier
    identifier-path `.` *identifier*

Wiring statements connect two *endpoints*. The *identifier-path* of an *endpoint* specifies a specification element. The *argument-expression-list* optionally specifies some parameter values. We say that an endpoint is parameterised if its specification element is parameterised and the endpoint has no parameter values. A compile-time error occurs if an endpoint has parameter values and any of the following is true:

- The parameter values are not all constant expressions.
- The endpoint's specification element is not parameterised.
- The parameter values are not in range for the specification element's parameter types.

A compile-time error occurs if the *identifier-path* of an *endpoint* is not of the three following forms:

- X, where X names an external specification element.
- $C.X$ where C is a component from the *component-list* and X is a specification element of C.
- C where C is a component name from the *component-list*. This form is used in implicit connections, discussed in Section 7.3. Note that this form cannot be used when parameter values are specified.

There are three wiring statements in nesC:

- *endpoint*$_1$ `=` *endpoint*$_2$ (equate wires): Any connection involving an external specification element. These effectively make two specification elements equivalent.

  Let S_1 be the specification element of *endpoint*$_1$ and S_2 that of *endpoint*$_2$. One of the following two conditions must hold or a compile-time error occurs:

  – S_1 is internal, S_2 is external (or vice-versa) and S_1 and S_2 are both provided or both used,

  – S_1 and S_2 are both external and one is provided and the other used.

- *endpoint*$_1$ `->` *endpoint*$_2$ (link wires): A connection involving two internal specification elements. Link wires always connect a used specification element specified by *endpoint*$_1$ to a provided one specified by *endpoint*$_2$. If these two conditions do not hold, a compile-time error occurs.

- *endpoint*$_1$ `<-` *endpoint*$_2$ is equivalent to *endpoint*$_2$ `->` *endpoint*$_1$.

In all three kinds of wiring, the two specification elements specified must be compatible, i.e., they must both be commands, or both be events, or both be interface instances. Also, if they are commands (or events), then they must both have the same function signature. If they are interface instances they must be of the same interface type. If these conditions do not hold, a compile-time error occurs.

If one endpoint is parameterised, the other must be too and must have the same parameter types; otherwise a compile-time error occurs.

It is legal to wire the same specification element multiple times. The discussion of wiring semantics in Section 7.4 explains nesC's behaviour in this case.

All external specification elements must be wired or a compile-time error occurs. However, internal specification elements may be left unconnected (these may be wired in another configuration, or they may be left unwired if the modules have the appropriate `default` event or command functions).

7.3 Implicit Connections

It is possible to write `C1 <- C2.I` or `C1.I <- C2` (and the same with `=`, or `->`). This syntax iterates through the specification elements of `C1` (resp. `C2`) to find a specification element `J` such that `C1.J <- C2.I` (resp. `C1.I <- C2.J`) forms a valid connection. If exactly one such `J` can be found, then the connection is made, otherwise a compile-time error occurs.

For instance, with:

```
module M1 {
  provides interface StdControl;
} ...
```

```
module M2 {
  uses interface StdControl as SC;
} ...
```

```
configuration C { }
implementation {
  components M1, M2;
  M2.SC -> M1;
}
```

The `M2.SC -> M1` line is equivalent to `M2.SC -> M1.StdControl`.

7.4 Wiring Semantics

[This section could use an informal presentation. I think the rather formal one below should stay because I doubt an informal one can be easily made unambiguous]

[issue: We should decide if wiring the same two interface instances together twice (with the same parameter values, etc) leads to one or two calls. If the answer is two (as the current implementation does), we should talk about multi-graphs below, and hence edge lists not sets.

Note: such multiple calls can also happen a bit more indirectly when configurations are involved, i.e., there are two distinct paths in the wiring graph between a pair of functions. One call or two?]

The semantics of wiring are defined in terms of the program's wiring graph, a directed graph which connects nodes representing command or event calls to nodes representing command or event definitions.

The program's wiring graph is built by assembling the *configuration graphs* of each configuration (by unioning their node and edge sets). A *configuration graph* is a directed graph representing the wiring specified in a single configuration.

We first introduce some notation. The configuration whose graph is being created will be denoted with C. Other components will be called C_1, C_2, D and M, M' if the component is a module. A specification element is called I, I_1, I_2 or J. A command or event is F, F_1 or F_2. Parameter lists are P, P_1 or P_2. Lists of parameter lists are denoted with L_P. The empty list is $()$, $::$ represents list concatenation.

Node names are of the form $D.F.P$ for command or event F in component D's specification with parameter values P, $D.I.F.P$ for command or event F in interface instance I of D's specification with parameter values P. We use X to stand for a node name. All endpoints in the wiring can be viewed as being of the form $D.I.P$ with $P = ()$ when no parameters are specified and $D = C$ if I is an external specification element. All wires can be viewed as being of the form $C_1.I_1.P_1$ `to` $C_2.I_2.P_2$ where I_1 is used if $C_1 \neq C$ and provided if $C_1 = C$, and I_2 is provided if $C_2 \neq C$ and used if $C_2 = C$.

The functions of endpoint $D.I.P$, $\mathcal{F}(D.I.P)$ are defined as:

- $\mathcal{F}(D.I.P) = \{D.I.P\}$ if I is a command or event.
- $\mathcal{F}(D.I.P) = \{D.I.F.P | F \text{is a command or event of} I\}$ if I is an interface instance.

We define a connection function for connecting two endpoints: $\mathcal{C}(C_1.I_1.P_1, C_2.I_2.P_2)$ returns a set of directed edges as follows:

- If I_1 is a command or event: $\mathcal{C}(C_1.I_1.P_1, C_2.I_2.P_2) = \{(C_1.I_1.P_1, C_2.I_2.P_2)\}$.
- If I_1 is an interface instance:

$$\begin{aligned}\mathcal{C}(C_1.I_1.P_1, C_2.I_2.P_2) &= \{C_1.I_1.F.P_1, C_2.I_2.F.P_2) | F \text{ is a command in} I_1\} \cup \\ &\quad \{C_2.I_2.F.P_2, C_1.I_1.F.P_1) | F \text{ is an event in} I_1\}\end{aligned}$$

Configuration graph (V, E) for configuration C is the smallest graph such that:

- For each wire $C_1.I_1.P_1$ `to` $C_2.I_2.P_2$:

$$\mathcal{F}(C_1, I_1, ()) \cup \mathcal{F}(C_2, I_2, ()) \cup \mathcal{F}(C_1, I_1, P_1) \cup \mathcal{F}(C_2, I_2, P_2) \in V$$

- For each wire $C_1.I_1.P_1$ `to` $C_2.I_2.P_2$:
  - $\mathcal{C}(C_1.I_1.P_1, C_2.I_2.P_2) \in E$
  - If $P_1 \neq ()$ then $\mathcal{C}(C_1.I_1.(), C_1.I_1.P_1) \in E$
  - If $P_2 \neq ()$ then $\mathcal{C}(C_2.I_2.P_2, C_2.I_2.()) \in E$

The semantics of a call to a command or event F of interface I in module M are now defined in terms of the whole program graph G.

First, if there is a cycle in G starting at $M.I.F.()$ a compile-time error occurs.

Otherwise, set S is the set of reachable command or event definitions $M'.I'.F'.()$ from $M.I.F.()$, along with the list of parameter values L_P on the path from $M.I.F.()$ to $M'.I'.F'.()$. Note that there will be two elements for $M'.I'.F'.()$ in S if there are paths with distinct parameter value lists. Also, the semantics of nesC guarantee that L_P has zero, one, or two elements (each element is a tuple of parameter values).

If $M.I.F$ is a non-parameterised call, then:

- If S is empty and M has no default defintion for $I.F$, a compile-time error occurs.
- If S is empty and M has a default defintion F' for $I.F$, then F' is called.

- If S is non-empty: the elements (X, L_P) of S are called in an arbitrary order. The parameterised elements receive the first element of L_P as parameter values.

If $M.I.F$ is a parameterised call with parameter values P (computed at runtime in M), then S is split into two sets:

- $S_1 = \{(X, ())|(X, ()) \in S\}$ (these are the parameterised definitions connected directly to $M.I.F$). Note that S_1 can be computed at compile-time.
- $S_2 = \{(X, (L_P)|(X, (P) :: L_P) \in S\}$ (these are the calls whose condition, as specified in configurations, matches P). Note that S_2 can only be computed at runtime.

The call then proceeds as follows:

- If S_1 is empty and there is no default definition for $I.F$, then a compile-time error occurs.
- If $S_1 \cup S_2$ is empty and M has a default definition F' for $I.F$, then F' is called with parameter values P.
- If $S_1 \cup S_2$ is non-empty: the elements of $S_1 \cup S_2$ are called in an arbitrary order. The elements of S_1 receive P as their parameter values. The parameterised elements (X, L_P) of S_2 receive the first element of L_P as parameter values.

The result of $M.I.F$ is the result of one of the functions called (note that the rules above guarantee that at least one function will always be called). [we could change this to "last function called"]

8 C

The C global scope is populated by including C `.h` files. All macro and C declarations and definitions in these files are placed in the global C scope and are available to all subsequently loaded components, interfaces and C files. Similarly, if a `#define` macros is undefined, then it will remain undefined for subsequently loaded files.

A C file called `tos.h` is automatically included before any components. Subsequent C files are loaded according to the `includes` directives found at the start of component and interface files:

nesC-file:
  includes-list$_{opt}$ *interface*
  includes-list$_{opt}$ *module*
  includes-list$_{opt}$ *configuration*

includes-list:
  includes
  includes-list includes

includes:
  `includes` *identifier-list* ;

An `includes X` directive tells nesC to load C file `X.h`. Once `X.h` has been loaded, all subsequent `includes X` directives are ignored.

As an example, interface type `Bar` might include C file `BarTypes.h` which defines types used in the interface:

```
Bar.nc:                                BarTypes.h:
includes BarTypes;                     typedef struct {
interface Bar {                          int x;
  command result_t bar(BarType arg1);    double y;
                                       } BarType;
```

8.1 Functions with no arguments, old-style C declarations

nesC functions with no arguments are declared with `()`, not `(void)`. The latter syntax reports a compile-time error.

Old-style C declarations (with `()`) and function definitions (parameters specified after the argument list) are not allowed in interfaces or components (and cause compile-time errors).

Note that neither of these changes apply to C files (so that existing `.h` files can be used unchanged).

8.2 nesC and the C preprocessor

supports the C preprocessor in all files (interfaces, components and C files). As an extension, C++-style // comments are also allowed.

The effect of `#define` and `#undef` within components is limited to the component in which they occur (this is somewhat like having a per-component scope for preprocessor macros).

While `#include` directives are allowed in components and interfaces, they are unlikely to be useful. For instance, a `#include <math.h>` directive will produce the following effects:

- In a configuration or interface type file: various compilation errors.
- In a module: this will declare a number of functions (e.g., `sin`) in the module's per-component-implementation scope. This is obviously not very useful.

9 Miscellaneous

9.1 Component and Interface Type Name Space

Components and interface types share a name space; this name space is global and distinct from the C global name space. Compile-time error if expected interface foo and found component foo.

9.2 Attributes

nesC uses gcc's [?] `__attribute__` syntax for declaring some properties of functions. These attributes can be placed either on function definitions or function declarations (after the parameter list).[2] The

[2] gcc doesn't allow attributes after the parameter list in function definitions.

attributes of a function f are the union of all attributes on all declarations and definitions of f. The attribute syntax in nesC is:

init-declarator-list: also
    init-declarator attributes
    init-declarator-list , *init-declarator attributes*

function-definition: also
    declaration-specifiers$_{opt}$ *declarator attributes declaration-list*$_{opt}$ *compound-statement*

attributes:
    attribute
    attributes attribute

attribute:
    `__attribute__` ((*attribute-list*))

attribute-list:
    single-attribute
    attribute-list , *single-attribute*

single-attribute:
    identifier
    identifier (*argument-expression-list*)

nesC supports two attributes:

- `C`: This attribute is used for a function f declared in modules (it is ignored for all other declarations). It specifies that f's should appear in the global C scope rather than in the module's per-component-implementation scope. This allows f to be called from C code.
- `spontaneous`: This attribute can be used on any function f (in modules or C code). It indicates that there are calls f that are not visible in the source code. Typically, functions that are called spontaneously are interrupt handlers, and the C `main` function. Section 9.4 discusses how the nesC compiler uses the `spontaneous` attribute during compilation.

Example of attribute use: in module `RealMain.td`:

```
module RealMain { ... }
implementation {
  int main(int argc, char **argv) __attribute__((C, spontaneous)) {
    ...
  }
}
```

This example declares that function `main` should actually appear in the C global scope (`C`), so that the linker can find it. It also declares that `main` can be called even though there are no function calls to `main` anywhere in the program (`spontaneous`).

9.3 Compile-time Constant Functions

nesC has a new kind of constant expression: *constant functions*. These are functions defined within the language which evaluate to a constant at compile-time.

nesC currently has one constant function, `unique`. More are planned for the near future.

`unsigned int unique(char *identifier)`
Returns: if the program contains n calls to `unique` with the same `identifier` string, each calls returns a different unsigned integer in the range $0..n-1$.

The intended use of `unique` is for passing a unique integer to parameterised interface instances, so that a component providing a parameterised interface can uniquely identify the various components connected to that interface.

9.4 Files and Compilation Model

The nesC compiler assumes that a component or interface x is found in a file called x`.nc`, and that an included C file y is found in a file called y`.h`. The current implementation of the nesC uses a search path to locate these files (details can be found in the `ncc` manual).

The nesC compiler's input is a component c to be compiled. The compiler locates c's file, loads and compiles this file and the files for all components, interfaces and C files used (directly or indirectly) by c. The compiler may assume that the only entry points to this component are the functions marked with the `spontaneous` attribute.

10 Grammar

Please refer to Appendix A of Kernighan and Ritchie (K&R) [?, pp234–239] while reading this grammar.

The following keywords are new for nesC: `as`, `call`, `command`, `components`, `configuration`, `event`, `implementation`, `interface`, `module`, `post`, `provides`, `signal`, `task`, `uses`, `includes`. These nesC keywords are not reserved in C files. The corresponding C symbols are accessible in nesC files by prefixing them with `__nesc_`*keyword* (e.g., `__nesc_keyword_as`).

nesC reserves all identifiers starting with `__nesc` for internal use. TinyOS reserves all identifiers starting with `TOS_` and `TOSH_`.

nesC files follow the *nesC-file* production; `.h` files included via the `includes` directive follow the *translation-unit* directive from K&R.

New rules:

nesC-file:
    includes-list$_{opt}$ *interface*

includes-list$_{opt}$ *module*
includes-list$_{opt}$ *configuration*

includes-list:
includes
includes-list includes

includes:
`includes` *identifier-list* ;

interface:
`interface` *identifier* { *declaration-list* }

module:
`module` *identifier specification module-implementation*

module-implementation:
`implementation` { *translation-unit* }

configuration:
`configuration` *identifier specification configuration-implementation*

configuration-implementation:
`implementation` { *component-list*$_{opt}$ *connection-list* }

component-list:
components
component-list components

components:
`components` *component-line* ;

component-line:
renamed-identifier
component-line , *renamed-identifier*

renamed-identifier:
identifier
identifier `as` *identifier*

connection-list:
connection
connection-list connection

connection:
endpoint `=` *endpoint*
endpoint `->` *endpoint*

endpoint `<-` *endpoint*

endpoint:
identifier-path
identifier-path `[` *argument-expression-list* `]`

identifier-path:
identifier
identifier-path `.` *identifier*

specification:
{ *uses-provides-list* }

uses-provides-list:
uses-provides
uses-provides-list uses-provides

uses-provides:
`uses` *specification-element-list*
`provides` *specification-element-list*

specification-element-list:
specification-element
{ *specification-elements* }

specification-elements:
specification-element
specification-elements specification-element

specification-element:
declaration
`interface` *renamed-identifier* $parameters_{opt}$

parameters:
`[` *parameter-type-list* `]`

Changed rules:

storage-class-specifier: also one of
`command` `event` `task`

declaration-specifiers: also
`default` *declaration-specifiers*

direct-declarator: also
identifier `.` *identifier*
direct-declarator parameters `(` *parameter-type-list* `)`

init-declarator-list: also

  init-declarator attributes
  init-declarator-list `,` *init-declarator attributes*

function-definition: also

  declaration-specifiers$_{opt}$ *declarator attributes declaration-list*$_{opt}$ *compound-statement*

attributes:

  attribute
  attributes attribute

attribute:

  `__attribute__` `(` `(` *attribute-list* `)` `)`

attribute-list:

  single-attribute
  attribute-list `,` *single-attribute*

single-attribute:

  identifier
  identifier `(` *argument-expression-list* `)`

postfix-expression: replaced by

  primary-expression
  postfix-expression `[` *argument-expression-list* `]`
  call-kind$_{opt}$ *primary* `(` *argument-expression-list*$_{opt}$ `)`
  postfix-expression `.` *identifier*
  postfix-expression `->` *identifier*
  postfix-expression `++`
  postfix-expression `--`

call-kind: one of

  `call signal post`