

nesC Language Reference Manual

David Gay, David Culler, Philip Levis

September 2002

1 Introduction

nesC is a new programming language designed to embody the structuring concepts and execution model of TinyOS [?]. TinyOS is an event-driven operating system designed for sensor network nodes that have very limited resources (e.g., 8K bytes of program memory, 512 bytes of RAM). TinyOS has been reimplemented in nesC.

The basic concepts behind nesC are:

- Separation of construction and composition: programs are built out of *components*, which are assembled (“wired”) to form whole programs. Components define two scopes, one for their specification (containing the names of their *interface instances*) and one for their implementation. Components have internal concurrency in the form of *tasks*. Threads of control may pass into a component through its *interfaces*. These threads are rooted either in a task or a hardware interrupt.
- Specification of component behaviour in terms of set of *interfaces*. Interfaces may be 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). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the `send` command unless it provides an implementation of the `sendDone` event. Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts (the nature of this binding is system-dependent, so is not described further in this reference manual).
- Components are statically linked to each other via their interfaces. This increases runtime efficiency, encourages robust design, 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.

The rest of this document is structured as follows: Sections 2 and 3 present the notation and terminology used in the reference manual. Sections ??, ??, ??, and ?? present nesC interfaces and components. Section ?? explains how C files, nesC interfaces and components are assembled into an application. Section ?? covers the remaining miscellaneous features of nesC. Finally, Section ?? fully defines nesC's grammar (as an extension to the C grammar from Appendix A of Kernighan and Ritchie (K&R) [?, pp234–239]).

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).

Several examples use the `uint8_t` and `uint16_t` types from the C99 standard `inttypes.h` file.

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 *specification elements*.
- *specification element*: An *interface instance*, *command* or *event* in a specification that is either *provided* or *used*.
- *module*: A component whose implementation is provided by C code.
- *configuration*: A component whose implementation is provided by a composition of other components with a specific wiring.
- *interface type*: An *interface type* specifies the interaction between two components, *the provider* and *the user*. This specification takes the form of a set of *commands* and *events*. Each interface type has a distinct name.

Interfaces are bi-directional: the provider of an interface implements its commands, the user of an interface implements its events.

- *command, event*: A function that is part of a component’s specification, either directly as a specification element or within one of the component’s interface instances.

When used directly as specification elements, commands and events can have *interface parameters*. As with interface instances, we distinguish between *simple commands (events)* without interface parameters and *parameterised commands (events)* with interface parameters. The interface parameters of a command or event are distinct from its regular function parameters.

- *interface instance*: An instance of a particular interface type in the specification of a component. An interface instance has an instance 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 has an interface parameter name and 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 (and, similarly, separate simple commands or events in the case of parameterised commands or events). Parameterised interface instances allow runtime selection based on parameter values between a set of commands (or between a set of events).

- *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, function-parameter and block scopes. Additionally there are per-component-specification, per-component-implementation and per-interface-type scopes.
- *namespaces*: nesC has the standard C variable (also used for functions and *typedefs*), tagged type (**struct**, **union** and **enum** tag names) and label namespaces. Additionally, nesC has a component and interface type namespace for component and interface type names.
- *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 a multi-function interaction channel between 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 ?? explains how components specify the interfaces they provide and use, Section ?? explains how commands and events are called from C code and Section ?? 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*. This identifier has global scope and belongs to the component and interface type namespace (so all interface types have names distinct from each other and from all components).

Each interface type has a separate scope for the declarations in *declaration-list*. This *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 ??).

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 the **send** command, while users must implement the **sendDone** event.

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*. This identifier has global scope and belongs to the component and interface type namespace. A component introduces two per-component scopes: a specification scope, nested in the C global scope, and an implementation scope nested in the specification scope.

A component can optionally include C files via the *includes-list* (see Section ??).

The *specification* lists the specification elements (interface instances, commands or events) used or provided by this component. Typically, commands call down towards the hardware components and events call up towards application components. A thread of control crosses components only through its specification elements.

Each specification element has a name (interface instance name, command name or event name). These names belong to the variable namespace of the per-component-specification scope.

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:

<pre>module A1 { uses interface X; uses interface Y; } ...</pre>	<pre>module A1 { uses { interface X; interface Y; } } ...</pre>
--	---

An interface instance is specified as follows:

specification-element:

```

interface renamed-identifier parametersopt
...

```

renamed-identifier:

```

identifier
identifier as identifier

```

interface-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 *interface-parameters* are omitted, then **interface X as Y** declares a simple interface instance, corresponding to a single interface to this component. If the *interface-parameters* are present (e.g., **interface SendMsg S[uint8_t id]**) then this is a declaration of a parameterised interface instance, corresponding to multiple interfaces to this component, one for each distinct tuple of parameter values (so **interface SendMsg S[uint8_t id]** declares 256 interfaces of type **SendMsg**). The types of the *parameters* must be integral types (**enums** are not allowed at this time).

Commands or events can be included directly as specification elements by including 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 are simple commands or events if no interface parameters are specified, or parameterised commands or events if interface parameters are specified. The *interface-parameters* are placed before the function's regular parameter list, e.g., **command void send[uint8_t id](int x):**

direct-declarator: also

```

direct-declarator interface-parameters ( parameter-type-list )
...

```

Note that interface parameters are only allowed on commands or events within component specifications, not within interface types.

Here is a full specification example:

```

configuration GenericComm {

```

```

provides {
  interface StdControl as Control;
  interface SendVarLenPacket;

  // 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 `SendVarLenPacket` of type `SendVarLenPacket`.
- Provides parameterised instances of interface type `SendMsg` and `ReceiveMsg`; the parameterised instances are named `SendMsg` and `ReceiveMsg` respectively.
- Uses event `sendDone`.

We say that a command (event) F provided in the specification of component K is *provided command (event) F* of K ; similarly, a command (event) used in the specification of component K is *used command (event) F* of K .

A command F in a provided interface instance X of component K is provided command $X.F$ of K ; a command F in a used interface instance X of K is used command $X.F$ of K ; an event F in a provided interface instance X of K is used event $X.F$ of K ; and an event F in a used interface instance X of K is provided event $X.F$ of K (note the reversal of used and provided for events due to the bidirectional nature of interfaces).

We will often simply refer to the “command or event α of K ” when the used/provided distinction is not relevant. Commands or events α of K may be parameterised or simple, depending on the parameterised or simple status of the specification element to which they correspond.

6 Modules

Modules implement a component specification with C code:

```

module-implementation:
    implementation { translation-unit }

```

where *translation-unit* is a list of C declarations and definitions (see K&R [?, pp234–239]).

The top-level declarations of the module's *translation-unit* belong to the module's per-component-implementation scope. These declarations have indefinite extent and can be: any standard C declaration or definition, a TinyOS task declaration or definition, a commands or event implementation.

6.1 Implementing the Module's Specification

The *translation-unit* must implement the all provided commands (events) α of the module (i.e., all directly provided commands and events, all commands in provided interfaces and all events in used interfaces). A module can call any of its commands and signal any of its events.

These command and event implementations are specified 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 *interface-parameters* (*parameter-type-list*)

The implementation of simple command or event α has the syntax of a C function definition for α (note the extension to *direct-declarator* to allow . in function names) 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 implementation of parameterised command or event α with interface parameters P has the syntax of a C function definition for α with storage class `command` or `event` where the function's regular parameter list is prefixed with the parameters P within square brackets (this is the same syntax as parameterised command or event declarations within a component specification). These interface parameter declarations P belong to α 's function-parameter scope and have the same extent as regular function parameters. 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:

- There is no implementation for a provided command or event.
- The type signature (and optional interface 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-kindopt primary ( argument-expression-listopt )
...
```

call-kind: one of

```
call signal post
```

A simple command α is called with `call α (...)`, a simple event α is signaled with `signal α (...)`. For instance, in a module that uses interface `Send` of type `SendMsg`: `call Send.send(1, sizeof(Message), msg1)`.

A parameterised command α (respectively, an event) with n parameters of type τ_1, \dots, τ_n is called with interface parameter expressions e_1, \dots, e_n as follows: `call α [e_1, \dots, e_n](...)` (respectively, `signal α [e_1, \dots, e_n](...)`). Interface parameter expression e_i must be assignable to type τ_i ; the actual interface parameter value is e_i cast to type τ_i . For instance, in a module that uses interface `Send[uint8_t id]` of type `SendMsg`:

```
int x = ...;
call Send.send[x + 1](1, sizeof(Message), &msg1);
```

Execution of commands and events is immediate, i.e., `call` and `signal` behave similarly to function calls. The actual command or event implementations executed by a `call` or `signal` expression depend on the wiring statements in the program’s configurations. These wiring statements may specify that 0, 1 or more implementations are to be executed. When more than 1 implementation is executed, we say that the module’s command or event has “fan-out”.

A module can specify a default implementation for a used command or event α that it calls or signals. A compile-time error occurs for default implementations of provided commands or events. Default implementations are executed when α is not connected to any command or event implementation. A default command or event is defined by prefixing a command or event implementation with the `default` keyword:

declaration-specifiers: also

```
default declaration-specifiers
```

For instance, 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) ...
```

Section ?? specifies what command or event implementations are actually executed and what result gets returned in all cases.

6.3 Tasks

A TinyOS task is an independent locus of control defined by a function of storage class `task` returning `void` and with no arguments: `task void myTask() { ... }`.¹ A task can also have a forward declaration, e.g., `task void myTask();`.

Tasks are posted by prefixing a call to the task with `post`, e.g., `post myTask()`. `Post` returns immediately; its return value is 1 if the task was successfully posted for independent execution, 0 otherwise. The type of a post expression is `unsigned char`.

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

call-kind: also one of
`post`

7 Configurations

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

configuration-implementation:
`implementation { component-listopt connection-list }`

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

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*.

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

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). The names chosen for components belong to the component's implementation scope.

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 *K* is used in two different configurations (or even twice within the same configuration) there is still only instance of *K* (and its variables) in the program.

7.2 Wiring

Wiring is used to connect specification elements (interfaces, commands, events) together. This section and the next (Section ??) define the syntax and compile-time rules for wiring. Section ?? details how a program's wiring statements dictate which functions get called at each **call** and **signal** expression.

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 interface 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 one the three following forms:

- *X*, where *X* names an external specification element.
- *K.X* where *K* is a component from the *component-list* and *X* is a specification element of *K*.
- *K* where *K* is a some component name from the *component-list*. This form is used in implicit connections, discussed in Section ???. 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.

- $endpoint_1 \leftarrow endpoint_2$ is equivalent to $endpoint_2 \rightarrow endpoint_1$.

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.

The same specification element may be connected multiple times, e.g.,:

```
configuration C {
  provides interface X;
} implementation {
  components C1, C2;

  X = C1.X;
  X = C2.X;
}
```

In this example, the multiple wiring will lead to multiple signalers (“fan-in”) for the events in interface *X* and for multiple functions being executed (“fan-out”) when commands in interface *X* are called. Note that multiple wiring can also happen when two configurations independently wire the same interface, e.g.:

<pre>configuration C { } implementation { components C1, C2; C1.Y -> C2.Y; }</pre>	<pre>configuration D { } implementation { components C3, C2; C3.Y -> C2.Y; }</pre>
--	--

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 implementations).

7.3 Implicit Connections

It is possible to write $K_1 \leftarrow K_2.X$ or $K_1.X \leftarrow K_2$ (and the same with $=$, or \rightarrow). This syntax iterates through the specification elements of K_1 (resp. K_2) to find a specification element Y such that $K_1.Y \leftarrow K_2.X$ (resp. $K_1.X \leftarrow K_2.Y$) forms a valid connection. If exactly one such Y can be found, then the connection is made, otherwise a compile-time error occurs.

For instance, with:

<pre>module M1 {</pre>	<pre>module M2 {</pre>
------------------------	------------------------

```

    provides interface StdControl;
} ...

    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

We first explain the semantics of wiring in the absence of parameterised interfaces. Section ?? below covers parameterised interfaces. Finally, Section ?? specifies requirements on the wiring statements of an application when viewed as a whole.

We define the meaning of wiring in terms of *intermediate functions*.² There is one intermediate function I_α for every command or event α of every component. For instance, with

```

interface X {
    command void f();
    event void g();
}

module M {
    provides interface X as P;
    uses interface X as U;
    provides command void h();
} implementation { ... }

```

module `M` has intermediate functions `M.P.f`, `M.P.g`, `M.U.f`, `M.U.g`, `M.h`. In examples in the rest of this section we name intermediate functions based on their component, optional interface instance name and function name.

An intermediate function is either used or provided. Each intermediate function takes the same arguments as the corresponding command or event in the component’s specification. The body of an intermediate function I is a list of calls (executed sequentially) to other intermediate functions. These other intermediate functions are the functions to which I is connected by the application’s wiring statements. The arguments I receives are passed on to the called intermediate functions unchanged. The result of I is a list of results (the type of list elements is the result type of the command or event corresponding to I), built by concatenating the result lists of the called intermediate functions. An intermediate function which returns an empty result list corresponds to an unconnected command or event; an intermediate function which returns a list of two or more elements corresponds to “fan-out”.

Intermediate Functions and Configurations The wiring statements in a configuration specify the body of intermediate functions. We first expand the wiring statements to refer to intermediate functions rather than specification elements, and we suppress the distinction between `=` and `->` wiring statements. We write $I_1 \leftrightarrow I_2$ for a connection between intermediate functions I_1 and I_2 . For instance, in:

²nesC can be compiled without explicit intermediate functions, so the behaviour described in this section has no runtime cost beyond the actual function calls and the runtime dispatch necessary for parameterised commands or events.

```

interface X {
  command void f();
  event void g();
}

module M {
  provides interface X as P;
  uses interface X as U;
  provides command void h();
} implementation { ... }

configuration C {
  provides interface X;
  provides command void h2();
}

implementation {
  components M;
  X = M.P;
  M.U -> M.P;
  h2 = M.h;
}

```

we have the following intermediate function connections:

```

C.X.f <-> M.P.f  M.U.f <-> M.P.f  C.h2 <-> M.h
C.X.g <-> M.P.g  M.U.g <-> M.P.g

```

In a connection $I_1 \leftrightarrow I_2$ from a configuration C one of the two intermediate functions is the *callee* and the other is the *caller*. The connection simply specifies that a call to the callee is added to the body of the caller. I_1 (similarly, I_2) is a callee if any of the following conditions hold (we use the internal, external terminology for specification elements with respect to the configuration C containing the connection):

- If I_1 corresponds to an internal specification element that is a provided command or event.
- If I_1 corresponds to an external specification element that is a used command or event.
- If I_1 corresponds to a command of interface instance X , and X is an internal, provided or external, used specification element.
- If I_1 corresponds to an event of interface instance X , and X is an external, provided or internal, used specification element.

If none of these conditions hold, I_1 is a caller. The rules for wiring in Section ?? ensure that a connection $I_1 \leftrightarrow I_2$ cannot connect two callers or two callees. In our previous example, in configuration C , $C.X.f$, $C.h2$, $M.P.g$, $M.U.f$ are callers and $C.X.g$, $M.P.f$, $M.U.g$, $M.h$ are callees. Thus the connections of C specify that a call to $M.P.f$ is added to $C.X.f$, a call to $C.X.g$ is added to $M.P.g$, etc.

Intermediate Functions and Modules The C code in modules calls, and is called by, intermediate functions.

The intermediate function I for provided command or event α of module M contains a single call to the implementation of α in M . Its result is the singleton list of this call's result.

The expression `call $\alpha(e_1, \dots, e_n)$` is evaluated as follows:

- The arguments e_1, \dots, e_n are evaluated, giving values v_1, \dots, v_n .

- The intermediate function I corresponding to α is called with arguments v_1, \dots, v_n , with results list L .
- If L has one or more elements, the result of the `call` is an arbitrary element chosen from L .
- If L is empty the default implementation for α is called with arguments v_1, \dots, v_n , and its result is the result of the `call`. Section ?? specifies that a compile-time error occurs if L can be empty and there is no default implementation for α .

The rules for `signal` expressions are identical.

7.4.1 Wiring and Parameterised Functions

If a command or event α of component K is parameterised with interface parameters of type τ_1, \dots, τ_n then there is an intermediate function $I_{\alpha, v_1, \dots, v_n}$ for every distinct tuple $(v_1 : \tau_1, \dots, v_n : \tau_n)$.

In modules, if intermediate function I_{v_1, \dots, v_n} corresponds to parameterised, provided command (or event) α then the call in I_{v_1, \dots, v_n} to α 's implementation passes values v_1, \dots, v_n as the values for α 's interface parameters.

The expression `call` $\alpha[e'_1, \dots, e'_m](e_1, \dots, e_n)$ is evaluated as follows:

- The arguments e_1, \dots, e_n are evaluated, giving values v_1, \dots, v_n .
- The arguments e'_1, \dots, e'_m are evaluated, giving values v'_1, \dots, v'_m .
- The v'_i values are cast to type τ_i , where τ_i is the type of the i th interface parameter of α .
- The intermediate function $I_{v'_1, \dots, v'_m}$ corresponding to α is called with arguments v_1, \dots, v_n , with results list L .³
- If L has one or more elements, the result of the `call` is an arbitrary element chosen from L .
- If L is empty the default implementation for α is called with interface parameter values v'_1, \dots, v'_m and arguments v_1, \dots, v_n , and its result is the result of the `call`. Section ?? specifies that a compile-time error occurs if L can be empty and there is no default implementation for α .

The rules for `signal` expressions are identical.

There are two cases when an endpoint in a wiring statement refers to a parameterised specification element:

- The endpoint specifies parameter values v_1, \dots, v_n . If the endpoint corresponds to commands or events $\alpha_1, \dots, \alpha_m$ then the corresponding intermediate functions are $I_{\alpha_1, v_1, \dots, v_n}, \dots, I_{\alpha_m, v_1, \dots, v_n}$ and wiring behaves as before.

³This call typically involves a runtime selection between several command implementations - this is the only place where intermediate functions have a runtime cost.

- The endpoint does not specify parameter values. In this case, both endpoints in the wiring statement correspond to parameterised specification elements, with identical interface parameter types τ_1, \dots, τ_n . If one endpoint corresponds to commands or events $\alpha_1, \dots, \alpha_m$ and the other to corresponds to commands or events β_1, \dots, β_m , then there is a connection $I_{\alpha_i, w_1, \dots, w_n} \leftrightarrow I_{\beta_i, w_1, \dots, w_n}$ for all $1 \leq i \leq m$ and all tuples $(w_1 : \tau_1, \dots, w_n : \tau_n)$ (i.e., the endpoints are connected for all corresponding parameter values).

7.4.2 Application-level Requirements

There are two requirement that the wiring statements of an application must satisfy, or a compile-time error occurs:

- There must be no infinite loop involving only intermediate functions.
- At every `call α` (or `signal α`) expression in the application's modules:
 - If the call is unparameterised: if the call returns no results there must be a default implementation of α .
 - If the call is parameterised: if substitution of any values for the interface parameters of α returns no results there must be a default implementation of α (note that this condition does not consider the expressions used to specify interface parameter values at the call-site).

8 nesC Applications

A nesC application has three parts: a list of C declarations and definitions, a set of interface types and a set of components. The naming environment of nesC applications is structured as follows:

- An outermost, global scope with three namespaces: a C variable and a C tag namespace for the C declarations and definitions, and a component and interface type namespace for the nesC interface types and components.
- C declarations and definitions may introduce their own nested scopes within the global scope, as usual (for function declarations and definitions, code blocks within functions, etc).
- Each interface type introduces a scope that holds the interface's commands or events. This scope is nested in the global scope, therefore command and event definitions can refer to C types and tags defined in the global scope.
- Each component introduces two new scopes. The specification scope, nested in the global scope, contains a variable namespace which holds the component's specification elements. The implementation scope, nested in the specification scope, contains a variable and a tag namespace.

For configurations, the implementation's scope variable namespace contains the names by which this component refers to its included components (Section ??). For modules, the implementation scope holds the tasks, C declarations and definitions that form the module's body. These declarations, etc may introduce their own nested scopes within the implementation scope (for function bodies, code blocks, etc). As a result of the scope nesting structure,

code in modules has access to the C declarations and definitions in the global scope, but not to any declarations or definitions in other components.

The C declarations and definitions, interface types and components that form a nesC application are determined by an on-demand loading process. The input to the nesC compiler is a single component K . The nesC compiler first loads C file `tos` (Section ??), then loads component K (Section ??). The code for the application is all the code loaded as part of the process of loading these two files. A nesC compiler can assume that all calls to functions, commands or events not marked with the `spontaneous` attribute (Section ??) occur in the loaded code (i.e., there are no “invisible” calls to non-`spontaneous` functions).⁴

Part of the process of loading a C file, nesC component or interface type involves locating the corresponding source file. The mechanism used to locate files is outside the scope of this reference manual; for details on how this works in the current compiler please see the `ncc` man page.

8.1 Loading C file X

If X has already been loaded, nothing more is done. Otherwise, file $X.h$ is located and preprocessed. Changes made to C macros (via `#define` and `#undef`) are visible to all subsequently preprocessed files. The C declarations and definitions from the preprocessed $X.h$ file are entered into the C global scope, and are therefore visible to all subsequently processed C files, interface types and components.

8.2 Loading Component K

If K has already been loaded, nothing more is done. Otherwise, file $X.nc$ is located and preprocessed. Changes made to C macros (via `#define` and `#undef`) are discarded. The preprocessed file is parsed using the following grammar:

nesC-file:

```
includes-listopt interface  
includes-listopt module  
includes-listopt configuration
```

includes-list:

```
includes  
includes-list includes
```

includes:

```
includes identifier-list ;
```

If $X.nc$ does not define `module K` or `configuration K` , a compile-time error is reported. Otherwise, all C files specified by the *includes-list* are loaded (Section ??). Then all interface types used in the component’s specification are loaded (Section ??). Next, the component specification

⁴For instance, the current nesC compiler uses this information to eliminate unreachable code.

is processed (Section ??). If K is a configuration, all components specified (Section ??) by K are loaded (Section ??). Finally, K 's implementation is processed (Sections ?? and ??)..

8.3 Loading Interface Type I

If I has already been loaded, nothing more is done. Otherwise, file $X.nc$ is located and preprocessed. Changes made to C macros (via `#define` and `#undef`) are discarded. The preprocessed file is parsed following the *nesC-file* production above. If $X.nc$ does not define `interface I` a compile-time error is reported. Otherwise, all C files specified by the *includes-list* are loaded (Section ??). Then I 's definition is processed (Section ??).

As an example of including C files in components or interfaces, interface type `Bar` might include C file `BarTypes.h` which defines types used in `Bar`:

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

The definition of interface `Bar` can refer to `BarType`, as can any component that uses or provides interface `Bar` (interface `Bar`, and hence `BarTypes.h`, are loaded before any such component's specification or implementation are processed).

9 Miscellaneous

9.1 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).

9.2 `//` comments

nesC allows `//` comments in C, interface type and component files.

9.3 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).⁵ The attributes of a function f are the union of all attributes on all declarations and definitions of f .

The attribute syntax in nesC is:

⁵gcc doesn't allow attributes after the parameter list in function definitions.

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 C declaration or definition *d* at the top-level of a module (it is ignored for all other declarations). It specifies that *d*'s should appear in the global C scope rather than in the module's per-component-implementation scope. This allows *d* to be used (e.g., called if it is a function) 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 ?? discusses how the nesC compiler uses the **spontaneous** attribute during compilation.

Example of attribute use: in file `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**).

9.4 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 call 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.

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*