nesC 1.2 Language Reference Manual

David Gay, Philip Levis, David Culler, Eric Brewer

August 2005

1 Introduction

nesC is an extension to C [3] designed to embody the structuring concepts and execution model of TinyOS [2]. 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. This manual describes v1.2 of nesC, changes from v1.0 and v1.1 are summarised in Section 2.

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 *interfaces*) 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 packet done). The interface forces a component that calls the "send packet" command to provide an implementation for the "send packet done" 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 programs.
- nesC is designed under the expectation that code will be generated by whole-program compilers. This allows for better code generation and analysis. An example of this is nesC's compile-time data race detector.
- The concurrency model of nesC is based on run-to-completion tasks, and interrupt handlers which may interrupt tasks and each other. The nesC compiler signals the potential data races caused by the interrupt handlers.

This document is a reference manual for nesC rather than a tutorial. The TinyOS tutorial[1] presents a gentler introduction to nesC.

The rest of this document is structured as follows: Section 2 summarises the new features in nesC since v1.0. Section 3 presents the notation used in the reference manual, and Section 4 the scoping and naming rules of nesC. Sections 5 and 6 present interfaces and components, while Sections 7, 8 and 9 explain how components are implemented. Section 10 presents nesC's concurrency model and data-race detection. Sections 11, 12 and 13 cover the extensions to C allowed in nesC programs. Section 14 explains how C files, nesC interfaces and components are assembled into an application and how nesC programs interact with the preprocessor and linker. Finally, Appendix A fully defines nesC's grammar (as an extension to the C grammar from Appendix A of Kernighan and Ritchie (K&R) [3, pp234–239]), and Appendix B gives a glossary of terms used in this reference manual.

2 Changes

The changes from nesC 1.1 to 1.2 are:

- Generic interfaces: interfaces can now take type parameters (allowing, e.g., a single interface definition for a queue of any type of values).
- Generic components: components can now be instantiated (at compile-time), and can take constant and type arguments (e.g., a generic queue component would take type and queue size arguments).
- Component specifications can include type and enum constant declarations; component selections and wiring statements can be interspersed in configurations; configuration implementations can refer to the types and enum constants of the components they include.
- Binary components: programs can now use components defined in binary form. The same functionality supports encapsulating a set of components as a single binary component for use in other programs.
- External types: types with a platform-independent representation and no alignment representation can now be defined in nesC (these are useful, e.g., for defining packet representations).

[1] Available with the TinyOS distribution at http://webs.cs.berkeley.edu

- Attributes: declarations may be decorated with attributes. Information on attribute use may be extracted for use in external programs. Details on this extraction process is beyond the scope of this language reference manual; see the nesC compiler documentation for details. Some predefined attributes have meaning to the nesC compiler. Use of `__attribute__` for nesC-specific features is deprecated (for details on these deprecated usages, see Section 10.3 of the nesC 1.1 reference manual).
- `includes` is deprecated and components can be preceded by arbitrary C declarations and macros. As a result, `#include` behaves in a more comprehensible fashion. For details on `includes`, see Section 9 of the nesC 1.1 reference manual.
- `return` can be used within `atomic` statements (the atomic statement is implicitly terminated by the `return`).

The changes from nesC 1.0 to 1.1 are:

1. `atomic` statements. These simplify implementation of concurrent data structures, and are understood by the new compile-time data-race detector.
2. Compile-time data-race detection gives warnings for variables that are potentially accessed concurrently by two interrupt handlers, or an interrupt handler and a task.
3. Commands and events which can safely be executed by interrupt handlers must be explicitly marked with the `async` storage class specifier.
4. The results of calls to commands or events with "fan-out" are automatically combined by new type-specific combiner functions.
5. `uniqueCount` is a new *constant function* (Section 13.2) which counts uses of `unique`.
6. The `NESC` preprocessor symbol indicates the language version. It is at least 110 for nesC 1.1, at least 120 for nesC 1.2.

3 Notation

The `typewriter` font is used for nesC code and for filenames. Single symbols in italics, with optional subscripts, are used to refer to nesC entities, e.g., "component K" or "value v".

Explanations of nesC constructs are presented along with the corresponding grammar fragments. In these fragments, we sometimes use ... to represent elided productions (irrelevant to the construct at hand). Appendix A presents the full nesC grammar.

Several examples use the `uint8_t` and `uint16_t` types (from the C99 standard `inttypes.h` file) and the standard TinyOS `result_t` type (which represents success vs failure of an operation).

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) [3, pp234–239]. 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. We do not repeat the productions from the C grammar here, but Appendix A lists and summarises the C grammar rules used by nesC.

4 Scopes and Name Spaces in nesC

nesC includes the standard C name spaces: *object*, which includes variables, functions, typedefs, and enum-constants; *label* for `goto` labels; *tag* for `struct`, `union`, `enum` tags. It adds an additional *component* name space for component and interface definitions. For simplicity, we assume that each scope contains all four name spaces, though language restrictions mean that many of these name spaces are empty (e.g., all component and interface definitions are global, so the *component* name space is empty in all but the global scope).

nesC follows the standard C scoping rules, with the following additions:

- Each interface definition introduces two scopes. The *interface parameter scope* is nested in the global scope and contains the parameters of generic interface definitions. The *interface scope* is nested in the interface parameter scope and contains the interface's commands and events.
- Each component definition introduces three new scopes. The *component parameter scope* is nested in the global scope and contains the parameters of generic component definitions. The *specification scope* is nested in the component parameter scope and contains the component's specification elements.

  The *implementation scope* is nested in the specification scope. For configurations, the implementation scope contains the names by which this component refers to its included components (Section 9.1). 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, following the usual C scoping rules.

As usual in C, scopes must not have multiple definitions of the same name within the same name space.

5 Interface and Component Specification

A nesC *interface definition* specifies a bi-directional interaction between two components, known as the *provider* and *user*. Interactions via interfaces are specified by two sets of functions: *commands* are function calls from the user to the provider component, *events* are function calls from the provider to the user component. In many cases, the provider component is providing some service (e.g., sending messages over the radio) and commands represent requests, events responses.

An interface definition has a unique name, optional C type parameters, and contains declarations for its command and event functions. An interface definition with type parameters is called a *generic interface definition*.

An *interface type* is a reference to an interface definition and, if the referenced definition is generic, corresponding type arguments. Components can only be connected via two interfaces with the same type.

A component's *specification* is the set of interfaces that it provides and uses. Each provided or used interface has a name and an interface type. Component specifications can also contain *bare* commands and events (i.e., not contained in an interface), `typedef`s and tagged type declarations; to simplify the exposition we defer discussion of these to Sections 5.4 and 5.5.

For instance, the following source code

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

interface Init<t> { // a generic interface definition
  command void doit(t x);
}

module Simple {
  provides interface Init<int> as MyInit;
  uses interface SendMsg as MyMessage;
} ...
```

shows two interface definitions, `SendMsg` and `Init`, and the specification of the `Simple` component. The specification of `Simple` has two elements: `MyInit`, a provided interface of type `Init<int>` and `MyMessage` a used interface of type `SendMsg`. `Simple` must implement the `MyInit.doit` command and the `MyMessage.sendDone` event. It can call the `MyMessage.send` command.

The rest of this section covers interface definitions, interface types and component specifications in detail. The sections on component definition (Section 6) and implementations (Sections 7 and 9) explain how commands and events are called and implemented, and how components are linked together through their interfaces.

5.1 Interface Definitions

Interface definitions have the following syntax:

interface-definition:
`interface` *identifier type-parameters*$_{opt}$ { *declaration-list* }

Interface definitions have a name (*identifier*) with global scope. This name belongs to the component name space (Section 4), so interface definitions must have a name distinct from other interface definitions and from components, however they do not conflict with regular C declarations.

The *type-parameters* is a list of optional C type parameters for this interface definition:

type-parameters:

`<` *type-parameter-list* `>`

type-parameter-list:
identifier
type-parameter-list `,` *identifier*

These parameters belong to the object name space of the interface's parameter scope (Section 4) and are therefore visible in the *declaration-list*. See Section 13.3 for how type parameters interact with C's type system (in brief, these type parameters can be used like `typedef`'d types). An interface definition with type parameters is called a *generic interface definition*.

The *declaration-list* of an interface definition specifies a set of commands and events. It must consist of function declarations with the `command` or `event` storage class:

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

The optional `async` keyword indicates that the command or event can be executed in an interrupt handler (see Section 10). The interface's commands and events belong to the object name space of the interface's scope (Section 4).

The example code above showed two simple interface definitions (`SendMsg` and `Init`). The following

```
interface Queue<t> {
  async command void push(t x);
  async command t pop();
  async command bool empty();
  async command bool full();
}
```

defines a generic interface `Queue` with a single type parameter, defining four commands which can be executed in an interrupt handler.

5.2 Interface Types

An interface type is specified by giving the name of an interface definition and, for generic interface definitions, any required type arguments:

interface-type:
`interface` *identifier* *type-arguments*$_{opt}$

type-arguments:
`<` *type-argument-list* `>`

type-argument-list:
type-name
type-argument-list `,` *type-name*

There must be as many types in *type-arguments* as there are parameters in the interface definition's type parameter list. Type arguments can not be incomplete or of function or array type.

Two interface types are the same if they refer to the same interface definition and their corresponding type arguments (if any) are of the same C type. Example interface types are `interface SendMsg` and `interface Queue<int>`.

5.3 Component Specification

The first part of a component's definition (see Section 6) is its *specification*, a declaration of provided or used specification elements, where each element is an interface, a bare command or event (Section 5.4) or a declaration (Section 5.5):

component-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 in a component specification. Multiple used or provided specification elements can be grouped in a single directive by surrounding them with { and }. For instance, these two specifications are identical:

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

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

An interface declaration has an interface type and an optional name:

specification-element:

interface-type instance-name$_{opt}$ *instance-parameters*$_{opt}$
...

instance-name:
`as` *identifier*

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

If the name is omitted, the interface's name is the same as the name of the interface definition specified by the interface type: `interface SendMsg` means the same thing as `interface SendMsg as SendMsg` and `interface Queue<int>` is the same as `interface Queue<int> as Queue`. A specification can contain independent interfaces of the same interface type, e.g.,

```
provides interface X as X1;
uses interface X as X2;
```

The interface names belong to the object name space of the specification's scope (Section 4), thus there is no confusion between interface names and interface definition names (the latter are in the component name space).

An interface declaration without *instance-parameters* (e.g., `interface X as Y`) declares a single interface to this component. A declaration with *instance-parameters* (e.g., `interface SendMsg S[uint8_t id]`) declares a *parameterised interface*, corresponding to multiple interfaces to this component, one for each distinct tuple of parameter values (so `interface SendMsg as S[uint8_t id, uint8_t id2]` declares 256 * 256 interfaces of type `SendMsg`). The types of the *parameters* must be integral types (`enum`s are not allowed at this time).

The specification for `AMStandard`, a component that dispatches messages received from the serial port and the radio to the application based on the "active message id" stored in the message, and sends messages to the radio or serial port depending on the selected destination address, is typical of many TinyOS system components:

```
module AMStandard {
  provides {
    interface StdControl;

    // The interface are parameterised by the active message id
    interface SendMsg[uint8_t id];
    interface ReceiveMsg[uint8_t id];
  }
  uses {
    interface StdControl as RadioControl;
    interface SendMsg as RadioSend;
    interface ReceiveMsg as RadioReceive;

    interface StdControl as SerialControl;
```

```
    interface SendMsg as SerialSend;
    interface ReceiveMsg as SerialReceive;
  }
} ...
```

It provides or uses nine interfaces:

- The provided interface `StdControl` of type `StdControl` supports initialisation of `AMStandard`.
- The provided parameterised interfaces of type `SendMsg` and `ReceiveMsg` (named `SendMsg` and `ReceiveMsg` respectively) support dispatching of received messages and sending of messages with a particular active message id
- The used interfaces control, send and receive messages from the radio and serial port respectively (another TinyOS component, the `GenericComm` configuration wires `AMStandard` to the lower-level components providing radio and serial port networking).

5.4 Bare Commands and Events

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 async`

It is a compile-time error if the *declaration* is not a function declaration with the `command` or `event` storage class. As in interfaces, `async` indicates that the command or event can be called from an interrupt handler. These bare command and events belong to the object name space of the specification's scope (Section 4).

As with interface declarations, bare commands (bare events) can have instance parameters; these are placed before the function's regular parameter list, e.g., `command void send[uint8_t id](int x)`:

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

If instance parameters are present, the declaration specifies a *bare, parameterised command* (*bare, parameterised event*). Note that instance parameters are not allowed on commands or events inside interface definitions.

Module `M` of Figure 1 (Section 9.6) shows an example of a component specification with a bare command.

5.5 Other Declarations in Specifications

A component specification can also include regular declarations (these belong to the specification scope):

uses-provides: also
declaration

These declarations must be either `typedef`s, or tagged type declarations. For example,

```
module Fun {
  typedef int fun_t;
  enum { MYNUMBER = 42 };
}
implementation { ... }
```

Note that declaration of an `enum` implicitly places enum constants in the component's specification scope.

5.6 Command and Event Terminology

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

A command F in a provided interface X of component K is provided command $X.F$ of K; a command F in a used interface X of K is used command $X.F$ of K; an event F in a provided interface X of K is used event $X.F$ of K; and an event F in a used interface 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 use Greek letters $\alpha, \beta, \ldots$ to refer to any command or event of a component when the distinction between bare commands (events) and commands (events) in interfaces is not relevant. Commands or events α of K are parameterised if the specification element to which they correspond is parameterised.

We will often simply refer to the "command or event α of K" when the used/provided distinction is not relevant.

6 Component Definition

A nesC component definition has a name, optional arguments, a specification and an implementation:

component:
comp-kind identifier comp-parameters$_{opt}$ *component-specification implementation*$_{opt}$

comp-kind:

```
module
configuration
component
generic module
generic configuration
```

implementation:

module-implementation
configuration-implementation

The component name belongs to the component name space of the global scope, hence must be distinct from other components and from interface definitions. There are three ways a component can be implemented: *modules* are components which are implemented with C code (Section 7), *binary components* are components which are only available in binary form (Section 8), and *configurations* are components which are implemented by assembling other components (Section 9).

Components with parameters are called *generic components*, they must be instantiated in a configuration before they can be used (Section 9). Components without parameters exist as a single instance which is implicitly instantiated. The component's definition must reflect these distinctions (the *comp-kind* rule): for instance, a generic module `A` is defined with `generic module A() {`..., a non-generic configuration `B` is defined with `configuration B {`... Binary components cannot be generic.

6.1 Generic Components

Generic component parameter lists are similar to function parameter lists, but allow for type parameters by (re)using the `typedef` keyword:

comp-parameters:

`(` *component-parameter-list* `)`

component-parameter-list:

component-parameter
component-parameter-list `,` *component-parameter*

component-parameter:

parameter-declaration
`typedef` *identifier*

The parameters belong to the object name space of the component's parameter scope (Section 4), and are hence visible both in the component's specification and implementation. Non-type parameters must be of arithmetic or `char[]` type. These parameters can be used as follows:

- Type parameters can be used as if the argument was of some unknown `typedef`'d type. Additionally, type parameters can be restricted to integral or numerical types, allowing integral or numerical operations to be used on the type. For more details, see Section 13.3.

- Non-type parameters are constants of some unknown value (for more details, see Section 13.1); they can be used in any constant expression. They cannot be assigned to.

An instantiation with arguments $a_1, \ldots, a_n$ of generic component X with parameters $p_1, \ldots, p_n$ behaves like a new, non-generic component with the specification and implementation of X where all uses of parameter p_i have been replaced by the corresponding a_i value or type.[2] Section 14.2 details when generic components get instantiated.

6.2 Examples

Some examples (with simple specifications):

```
module A { provides interface X; } ...
component B { uses interface X } // no implementation for binary components!
generic configuration B() { uses interface Y; } ...
generic module AQueue(int n, typedef t) { provides interface Queue<t>; } ...
```

`A` is a simple module, `B` a generic configuration with no arguments but which can be instantiated multiple times, `AQueue` a generic module implementing an `n` entry queue with elements of type `t`. Note how `AQueue` instantiates the generic interface `Queue` with its type parameter `t`.

7 Component Implementation: 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 [3, pp234–239]).

The top-level declarations of the module's *translation-unit* belong to the module's implementation scope (Section 4). These declarations have indefinite extent and can be: any standard C declaration or definition, a task declaration or definition (placed in the object name space), a command or event implementation.

7.1 Implementing the Module's Specification

The *translation-unit* must implement all provided commands (events) α of the module (i.e., all commands in provided interfaces, all events in used interfaces, and all bare, provided commands and events). 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:

[2] The most straightforward implementation of these semantics for generic modules is to duplicate X's code. In some cases (e.g., no arguments to X), a nesC compiler might be able to share code between the instances of X at some runtime cost.

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

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

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

The implementation of non-parameterised 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`. Additionally, the `async` keyword must be included iff it was included in α's declaration. For example, in a module that provides interface `Send` of type `SendMsg` (shown at the start of Section 5):

```
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 instance 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. These instance 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, optional interface parameters and presence or absence of the `async` keyword of a command or event does not match that given in the module's specification.

7.2 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 non-parameterised command α is called with `call` α`(...)`, a non-parameterised 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 instance parameters of type $\tau_1, \ldots, \tau_n$ is called with instance arguments $e_1, \ldots, e_n$ as follows: `call` α`[`$e_1, \ldots, e_n$`](...)` (respectively, `signal` α`[`$e_1, \ldots, e_n$`](...)`). Interface argument e_i must be assignable to type τ_i; the actual interface argument value is the value of 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 (a compile-time error occurs if a default implementation is supplied for a provided command or event). Default implementations are executed when α is not connected to any command or event implementation (see Section 9.6). 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 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 9.6 specifies what command or event implementations are actually executed and what result gets returned by `call` and `signal` expressions.

7.3 Tasks

A task is an independent locus of control defined by a function of storage class `task` returning `void` and with no arguments: `task void myTask() { ... }`.[3] A task can also have a forward declaration, e.g., `task void myTask();`.

Tasks are posted for later execution 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, 0 otherwise. The type of a post expression is `unsigned char`.

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

call-kind: also one of
`post`

Section 10, which presents nesC's concurrency model, explains when tasks get executed.

7.4 Atomic statements

Atomic statements:

atomic-stmt:
`atomic` *statement*

guarantee that the statement is executed "as-if" no other computation occurred simultaneously, and furthermore any values stored inside an atomic statement are visible inside all subsequent atomic statements. Atomic statements are used to implement mutual exclusion, for updates to concurrent data structures, etc. The following example uses `atomic` to prevent concurrent execution of `do_something`:

```
bool busy; // global

void f() { // called from an interrupt handler
  bool available;

  atomic {
    available = !busy;
    busy = TRUE;
  }
  if (available) do_something;
  atomic busy = FALSE;
}
```

Atomic sections should be short, though this is not currently enforced in any way. Except for `return` statements, control may only flow "normally" in or out of on atomic statement: any `goto`,

[3] nesC functions with no arguments are declared with `()`, not `(void)`. See Section 13.4.

`break` or `continue` that jumps in or out of an atomic statement is an error. A `return` statement is allowed inside an atomic statement; at runtime the atomic section ends after evaluating the returned expression (if any) but before actually returning from the function.

Section 10 discusses the relation between `atomic`, nesC's concurrency model, and the data-race detector.

8 Component Implementation: Binary Components

Binary components are declared with the `component` keyword and have no `implementation` section. Instead, program's using binary components must be linked with an object file providing the binary component's implementation — this object file might be the result of compiling a different nesC program.

This object file must provide definitions for the provided commands and events of the binary component, and can call its used commands and events. For more details on external linkage rules for nesC, see Section 14.5.

Note that `default` commands and events (see Sections 7 and 9.6) do not work across binary component boundaries — the used commands and events of a binary component must be fully wired.

9 Component Implementation: Configurations

Configurations implement a component specification by selecting regular components or instantiating generic components, and then connecting ("wiring") these components together. The implementation section of a configuration consists of a list of configuration elements:

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

configuration-element-list:
configuration-element
configuration-element-list configuration-element

configuration-element:
components
connection
declaration

A *components* element specifies the components that are used to build this configuration (Section 9.1), a *connection* specifies a single wiring statement (Section 9.2), and a *declaration* can declare a `typedef` or tagged type (other C declarations are compile-time errors) (Section 9.4).

A configuration C's wiring statements connects two sets of specification elements:

- C's specification elements. In this section, we refer to these as *external* specification elements.
- The specification elements of the components referred to instantiated in C. We refer to these as *internal* specification elements.

9.1 Included components

A *components* elements specifies some components used to build this configuration. These can be:

- A non-generic component X. Non-generic components are implicitly instantiated, references to X in different configurations all refer to the same component.
- An instantiation of a generic component Y. Instantiations of Y in different configurations, or multiple instantiations in the same configuration represent different components (see Section 6.1).

The syntax of *components* is as follows:

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

component-line:
component-ref instance-name$_{opt}$
component-line `,` *component-ref instance-name*$_{opt}$

instance-name:
`as` *identifier*

component-ref:
identifier
`new` *identifier* `(` *component-argument-list* `)`

component-argument-list:
component-argument
component-argument-list `,` *component-argument*

component-argument:
expression
type-name

Each *component-ref* specifies a non-generic component X by simply giving its name (a compile-time error occurs if X is generic) and a generic component Y with `new`$Y(args)$ (a compile-time error occurs if Y is not generic). The arguments to Y must match the number of parameters of Y's definition, and:

- If the parameter is a type parameter, then the argument must be a type which is not incomplete, or of function or array type.

- If the parameter is of type `char[]`, the argument must be a string constant.
- If the parameter is of arithmetic type, the argument must be a constant whose value is in the range of the parameter type.

Within a *connection*, a component specified in *components* is referred to by:

- The name explicitly specified by the `X as Y` syntax (*instance-name*). Use of `as` is necessary, e.g., when instantiating the same generic component more than once in a given configuration.
- The name of the component definition (`components new X(), Y;` is the same as `components new X() as X, Y as Y;`).

The names specified by *components* elements belong to the object name space of the component's implementation scope (Section 4).

This `NoWiring` configuration:

```
configuration NoWiring { }
implementation {
  components A, new AQueue(10, int);
  components new AQueue(20, float) as SecondQueue;
}
```

selects component `A`, and instantiates generic component `AQueue` twice. The two instances of `AQueue` are known as `AQueue` and `SecondQueue` within `NoWiring`.

9.2 Wiring

Wiring is used to connect specification elements (interfaces, commands, events) together. This section and the next (Section 9.3) define the syntax and compile-time rules for wiring. Section 9.6 details how a program's wiring statements dictate which functions get called by the `call` and `signal` expressions found in modules.

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 (either internal or external). The *argument-expression-list* optionally specifies instance arguments. We say that an endpoint is parameterised if its specification element is parameterised and the endpoint has no arguments. A compile-time error occurs if an endpoint has arguments and any of the following is true:

- Some arguments is not a constant expression.
- The endpoint's specification element is not parameterised.
- There are more (or less) arguments than there are parameters on the specification element.
- The argument's 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 9.3. This form cannot be used when arguments are specified.

  Note that a component name can hide an external specification element, preventing the element from being wired:

  ```
  configuration AA { provides interface X as Y; }
  implementation {
    components Z as Y, Z2 as Y2;

    Y /* refers to component Z, not interface X */ -> Y2.A;
  }
  ```

  Hiding specification elements will always result in a compile-time error as external specification elements must all be wired.

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 between 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 interfaces. Also, if they are commands (or events), then they must both have the same function signature. If they are interfaces they must have 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.

A configuration's external specification elements must all 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, see Section 9.6).

9.3 Implicit Connections

It is possible to write K_1 `<-` K_2`.`X or K_1`.`X `<-` K_2 (and the same with `=`, or `->`). This syntax iterates through the specification elements of K_1 (resp. K_2) to find a specification element Y such that K_1`.`Y `<-` K_2`.`X (resp. K_1`.`X `<-` 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:

```
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`.

9.4 Declarations in Configurations

As we saw above, like component specifications (Section 5.5), configurations can include `typedef` and tagged type declarations. These declarations belong to the configuration's implementation scope.

Additionally, a configuration can refer to the `typedef`s and enum constants of the components that it includes. To support this, the syntax for referring to `typedef`s is extended as follows:

typedef-name: also one of
identifier . *identifier*

where the first identifier must refer to one of the configuration's components with an appropriate `typedef` in its specification. Similarly, enum constants are referenced by extending C's field-reference syntax to allow the object to be the name of one of the configuration's components.

For example:

```
module M {
  typedef int t;
  enum { MAGIC = 54 };
} ...

configuration C { }
implementation {
  components M as Someone;

  typedef Someone.t Ct;
  enum { GREATERMAGIC = Someone.MAGIC + 1 };
}
```

9.5 Examples

The first example shows all possible wiring cases (comments within the example):

```
configuration All {
  provides interface A as ProvidedA1;
  provides interface A as ProvidedA2;
  provides interface A as ProvidedA3;
  uses interface A as UsedA1;
}
implementation {
  components new MyComponent() as Comp1, new MyComponent() as Comp2;

  // equate our interface ProvidedA1 with MyA provided by Comp1
  ProvidedA1 = Comp1.MyA;

  // the same, for ProvidedA2 and MyA of Comp2. We rely on the implicit
  // connection to avoid naming MyA
  ProvidedA2 = Comp2;

  // An equate wire connecting ProvidedA3 with UsedA1. We're just passing
  // the interface through
  ProvidedA3 = UsedA1;
```

```
  // Link some B interfaces together:
  Comp1.UsedB -> Comp2.MyB; // fully explicit connection
  Comp1 -> Comp2.MyB; // implicit equivalent of above line
  Comp1 <- Comp2.UsedB; // implicit equivalent of Comp2.UsedB -> Comp1.MyB
}

generic module MyComponent() {
  provides interface A as MyA;
  provides interface B as MyB;
  uses interface B as UsedB;
} implementation { ... }
```

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., the following example wires `C2.Y` twice:

```
configuration C { }
implementation {
  components C1, C2;

  C1.Y -> C2.Y;
}
```

```
configuration D { }
implementation {
  components C3, C2;

  C3.Y -> C2.Y;
}
```

9.6 Wiring Semantics

We first explain the semantics of wiring in the absence of parameterised interfaces. Section 9.6.1 below covers parameterised interfaces. Section 9.6.2 specifies requirements on the wiring statements of an application when viewed as a whole. We will use the simple application of Figure 1 as our running example.

For the purposes of this section, we will assume that all instantiations of generic components have been expanded into non-generic components as explained in Sections 6.1 and 14.2.

We define the meaning of wiring in terms of *intermediate functions*.[4] There is one intermediate function I_α for every command or event α of every component. For instance, in Figure 1, module

[4] nesC can be compiled without explicit intermediate functions, so the behaviour described in this section has no

```
interface X {                 module M {
  command int f();              provides interface X as P;
  event bool g(int x);          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;
}
```

Figure 1: Simple Wiring Example

M has intermediate functions $I_{\texttt{M.P.f}}$, $I_{\texttt{M.P.g}}$, $I_{\texttt{M.U.f}}$, $I_{\texttt{M.U.g}}$, $I_{\texttt{M.h}}$. In examples, we name intermediate functions based on their component, interface 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 this list's 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 `<->` I_2 for a connection between intermediate functions I_1 and I_2. For instance, configuration `C` from Figure 1 specifies the following intermediate function connections:

$I_{\texttt{C.X.f}}$ `<->` $I_{\texttt{M.P.f}}$ $I_{\texttt{M.U.f}}$ `<->` $I_{\texttt{M.P.f}}$ $I_{\texttt{C.h2}}$ `<->` $I_{\texttt{M.h}}$

$I_{\texttt{C.X.g}}$ `<->` $I_{\texttt{M.P.g}}$ $I_{\texttt{M.U.g}}$ `<->` $I_{\texttt{M.P.g}}$

In a connection I_1 `<->` 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

runtime cost beyond the actual function calls and the runtime dispatch necessary for parameterised commands or events.

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 bare, provided command or event.
- If I_1 corresponds to an external specification element that is a bare, 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 9.2 ensure that a connection I_1 `<->` I_2 cannot connect two callers or two callees. In configuration `C` from Figure 1, $I_{\texttt{C.X.f}}$, $I_{\texttt{C.h2}}$, $I_{\texttt{M.P.g}}$, $I_{\texttt{M.U.f}}$ are callers and $I_{\texttt{C.X.g}}$, $I_{\texttt{M.P.f}}$, $I_{\texttt{M.U.g}}$, $I_{\texttt{M.h}}$ are callees. Thus the connections of `C` specify that a call to $I_{\texttt{M.P.f}}$ is added to $I_{\texttt{C.X.f}}$, a call to $I_{\texttt{C.X.g}}$ is added to $I_{\texttt{M.P.g}}$, 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` $\alpha(e_1, \ldots, e_n)$ is evaluated as follows:

- The arguments $e_1, \ldots, e_n$ are evaluated, giving values $v_1, \ldots, v_n$.
- The intermediate function I corresponding to α is called with arguments $v_1, \ldots, v_n$, with results list L.
- If $L = (w)$ (a singleton list), the result of the `call` is w.
- If $L = (w_1, w_2, \ldots, w_m)$ (two or more elements), the result of the `call` depends on the result type τ of α. If $\tau =$ `void`, then the result is `void`. Otherwise, τ must have an associated *combining function* c (Section 11 shows how combining functions are associated with types), or a compile-time error occurs. The combining function takes two values of type τ and returns a result of type τ. The result of the `call` is $c(w_1, c(w_2, \ldots, c(w_{m-1}, w_m)))$ (note that the order of the elements of L was arbitrary).
- If L is empty the default implementation for α is called with arguments $v_1, \ldots, v_n$, and its result is the result of the `call`. Section 9.6.2 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.

```
list of int I_M.P.f() {
  return list(M.P.f());
}

list of int I_M.U.f() {
  return I_M.P.f();
}

list of int I_C.X.f() {
  return I_M.P.f();
}

list of void I_C.h2() {
  return I_M.h();
}
```

```
list of bool I_M.P.g(int x) {
  list of bool r1 = I_C.X.g(x);
  list of bool r1 = I_M.U.g(x);
  return list_concat(r1, r2);
}

list of bool I_M.U.g(int x) {
  return list(M.U.g(x));
}

list of bool I_C.X.g(int x) {
  return empty_list;
}

list of void I_M.h() {
  return list(M.h());
}
```

Figure 2: Intermediate Functions for Figure 1

Example Intermediate Functions Figure 2 shows the intermediate functions that are produced for the components of Figure 1, using a C-like syntax, where `list(`x`)` produces a singleton list containing x, `empty_list` is a constant for the 0 element list and `concat_list` concatenates two lists. The calls to `M.P.f`, `M.U.g`, `M.h` represent calls to the command and event implementations in module `M` (not shown).

9.6.1 Wiring and Parameterised Functions

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

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

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

- The arguments $e_1, \ldots, e_n$ are evaluated, giving values $v_1, \ldots, v_n$.
- The arguments $e'_1, \ldots, e'_m$ are evaluated, giving values $v'_1, \ldots, v'_m$.
- The v'_i values are cast to type τ_i, where τ_i is the type of the ith interface parameter of α.
- The intermediate function $I_{v'_1,\ldots,v'_m}$ corresponding to α is called with arguments $v_1, \ldots, v_n$, with results list L.[5]

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

- If L has one or more elements, the result of the `call` is produced as in the non-parameterised case.
- If L is empty the default implementation for α is called with interface parameter values $v'_1, \ldots, v'_m$ and arguments $v_1, \ldots, v_n$, and its result is the result of the `call`. Section 9.6.2 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 $v_1, \ldots, v_n$. If the endpoint corresponds to commands or events $\alpha_1, \ldots, \alpha_m$ then the corresponding intermediate functions are $I_{\alpha_1,v_1,\ldots,v_n}$, $\ldots$, $I_{\alpha_m,v_1,\ldots,v_n}$ and wiring behaves as before.
- 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 $\tau_1, \ldots, \tau_n$. If one endpoint corresponds to commands or events $\alpha_1, \ldots, \alpha_m$ and the other to corresponds to commands or events $\beta_1, \ldots, \beta_m$, then there is a connection $I_{\alpha_i,w_1,\ldots,w_n}$ `<->` $I_{\beta_i,w_1,\ldots,w_n}$ for all $1 \leq i \leq m$ and all tuples $(w_1 : \tau_1, \ldots, w_n : \tau_n)$ (i.e., the endpoints are connected for all corresponding parameter values).

9.6.2 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 an empty result list there must be a default implementation of α (the number of elements in the result list depends only on the wiring).
  - If the call is parameterised: if substitution of any values for the interface parameters of α returns an empty result list there must be a default implementation of α (the number of elements in the result list for a given parameter value tuple depends only on the wiring).

    Note that this condition does not consider the expressions used to specify interface parameter values at the call-site.

10 Concurrency in nesC

nesC's execution model is based on run-to-completion *tasks* (that typically represent the ongoing computation), and *interrupt handlers* that are signaled asynchronously by hardware. The compiler

relies on the user-provided `hwevent` and `atomic_hwevent` attributes to recognise interrupt handlers (see Section 11). A scheduler for nesC can execute tasks in any order, but must obey the run-to-completion rule (the standard TinyOS scheduler follows a FIFO policy). Because tasks are not preempted and run to completion, they are atomic with respect to each other, but are not atomic with respect to interrupt handlers.

As this is a concurrent execution model, nesC programs are susceptible to race conditions, in particular data races on the program's *shared state*, i.e., its global and module variables (nesC does not include dynamic memory allocation). Races are avoided either by accessing shared state only in tasks, or only within atomic statements. The nesC compiler reports potential data races to the programmer at compile-time.

Formally, we divide the code of a nesC program into two parts:

> **Synchronous Code (SC):** code (functions, commands, events, tasks) that is only reachable from tasks.
>
> **Asynchronous Code (AC):** code that is reachable from at least one interrupt handler.

Although non-preemption eliminates data races among tasks, there are still potential races between SC and AC, as well as between AC and AC. To prevent data races, nesC issues warnings for violations of the following rules:

> **Race-Free Invariant 1**: If a variable x is written in AC, then all accesses to x must occur in atomic sections.
>
> **Race-Free Invariant 2**: If a variable x is read in AC, then all writes to x must occur in atomic sections.

The body of a function f called from an atomic statement is considered to be "in" the atomic statement as long as all calls to f are "in" atomic statements.

It is possible to introduce a race condition that the compiler cannot detect, but it must span multiple atomic statements or tasks and use storage in intermediate variables.

nesC may report data races that cannot occur in practice, e.g., if all accesses are protected by guards on some other variable. To avoid redundant messages in this case, the programmer can annotate a variable v with the `norace` storage-class specifier to eliminate all data race warnings for v. The `norace` keyword should be used with caution.

nesC reports a compile-time error for any command or event that is AC and that was not declared with `async`. This ensures that code that was not written to execute safely in an interrupt handler is not called inadvertently.

11 Attributes

All C and nesC declarations can be decorated with *attributes* (inspired by Java 1.5's attributes [1]) that:

- Avoid reserving lots of keywords and burdening the syntax. For example, `@integer()` is used to mark generic component type arguments that must be integer types (Section 13.3).

- Allow user-specified annotations which are accessible to external tools. The mechanism by which these user-specified attributes are accessed is beyond the scope of this reference manual; please see the nesC compiler manual for details.

User-defined attributes must be declared prior to use, and have no effect on code generation. The language-defined attributes are implicitly declared; their effects are described in Section 11.

An attribute declaration is simply a `struct` declaration where the `struct`'s name is preceded by `@`:

struct-or-union-specifier: also one of
`struct @` *identifier* { *struct-declaration-list* }

A use of an attribute specifies the attribute's name and gives an initialiser (in parentheses) that must be valid for attribute's declaration:

attribute:
`@` *identifier* `(` *initializer-list* `)`

Attributes can be placed on all C and nesC declarations and definitions. Generally, attributes appear after the annotated object's name and associated arguments, but before any other syntactic elements (e.g., initialisers, function bodies, etc). See Appendix A for the full set of rules. The attributes of x are the union of all attributes on all declarations and definitions of x.

Example:

```
struct @myattr {
  int x;
  char *why;
};

extern int z @myattr(5, "fun"); // simple use

// a second attribute on z at it's definition
int z @myattr(3, "morefun") = 22;

// use on a function, with a C99-style initialiser
void f(void) @myattr(.x=5, .why="for f") {
   ...
}

// use in a module, with an empty initialiser
module X {
  provides interface I @myattr();
}
...
```

11.1 nesC Attributes

nesC includes seven predefined attributes with various effects. Except where otherwise specified, these take no arguments:

- `@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. The C `main` function is a typical example. Section 14 discusses how the nesC compiler uses the `spontaneous` attribute during compilation.

- `@hwevent()`: This attribute can be used on any function f (in modules or C code). It indicates that f is an interrupt handler, i.e., that there are spontaneous calls to f and that f is AC (Section 10). The use of `@hwevent()` implies `@spontaneous()`.

- `@atomic_hwevent()`: This attribute can be used on any function f (in modules or C code). This behaves the same as `@hwevent()`, but, additionally, informs nesC that the body of f behaves as if it were an `atomic` statement (on typical hardware this means that this interrupt handler runs with interrupts disabled). The use of `@atomic_hwevent()` implies `@spontaneous()`.

  Note that neither `@hwevent()` or `@atomic_hwevent()` provide any linkage of f with a particular interrupt handler. The mechanism by which that is achieved is platform-specific.

  Inside a function with the `@atomic_hwevent()` attribute, a call to `__nesc_enable_interrupt()` is assumed to terminate the implicit `atomic` statement. This is useful for interrupt handlers which must start with interrupts disabled, but can reenable interrupts after a little work.

- `@combine(`$fnname$`)`: This attribute specifies the combining function for a type in a `typedef` declaration. The combining function specifies how to combine the multiple results of a call to a command or event which has "fan-out". For example:

  ```
  typedef uint8_t result_t @combine("rcombine");

  result_t rcombine(result_t r1, result_t r2)
  {
    return r1 == FAIL ? FAIL : r2;
  }
  ```

  specifies logical-and-like behaviour when combining commands (or events) whose result type is `result_t`. See Section 9.6 for the detailed semantics.

  A compile-time error occurs if the combining function c for a type t does not have the following type: t `c(`t`, `t`)`.

- `@integer()`, `@number()`: declare properties of generic component type parameters. See Section 13.3.

Example of attribute use:

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

12 External Types

External types are an extension to C that allows definition of types with a platform-independent representation and no alignment restriction (i.e., an arbitrary `char` array can be cast to, and accessed via, an external type). They are intended for communication with entities external to the nesC program (e.g., other devices via a network), hence their name.

nesC has three kinds of external types:

- External base types are 2's complement integral types with a fixed size and endianness. These types are `nx_int`N`_t`, `nx_uint`N`_t`, `nxle_int`N`_t`, `nxle_uint`N`_t` for $N = 8, 16, 32, 64$. The `nx_` types are big-endian, the `nxle_` types are little endian, the `int` types are signed and the `uint` types are unsigned. Note that these types are not keywords.
- External array types are any array built from an external type, using the usual C syntax, e.g, `nx_int16_t x[10]`.

  External structures and unions are declared like C structures and unions, but using the `nx_struct` and `nx_union` keywords. An external structure can only contain external types as elements. Currently, external structures and unions cannot contain bitfields.

External types have no alignment restrictions and external structures contain no padding. External types can be used exactly like regular C types.[6]

[6]The current nesC compiler does not support using external base types in casts, as function arguments and results, or to declare initialised variables.

13 Miscellaneous

13.1 Constant Folding in nesC

There are two extensions to C's constant folding (see A.7.19 in K&R [3]) in nesC: *constant functions* and *unknown constants*. *Constant functions* are functions provided by the nesC language which return a compile-time constant. The definition of nesC's constant functions is given in Section 13.2. An *unknown constant* is a constant whose value is not known at some stage of semantic checking, e.g., non-type parameters to generic components are unknown constants when a generic component is loaded and checked. Unknown constants allow the definition of a generic component to be (mostly, see next paragraph) checked for correctness before its arguments' values are known.

An expression involving an unknown constant is considered a constant expression if the resulting expression is constant irrespective of the unknown constant's value, with the following exceptions: a/b and $a\%b$ can assume that b is not zero. Constant expressions involving unknown constants are re-checked once the values of constant expressions become known.[7] As a result, the following generic component definition is legal:

```
generic module A(int n) { }
implementation {
  int s = 20 / n;
}
```

but the following instantiation will report a compile-time error:

```
configuration B { }
implementation {
  components new A(0) as MyA;
}
```

13.2 Compile-time Constant Functions

nesC currently has three constant functions:

- `unsigned int unique(char *identifier)`
  `unsigned int uniqueN(char *identifier, unsigned int nb)`

  Given a program with k uses of `uniqueN` with the same `identifier` and values $n_1, \ldots n_k$ for `nb`, each use returns an integer u_i from the sequence $0 \ldots (\sum_{i=0}^{k} n_i) - 1$. Furthermore, the sequences $u_i \ldots u_i + n_i - 1$ do not overlap with each other. Note that $n_i = 0$ is allowed. The behaviour is undefined if $\sum_{i=0}^{k} n_i >=$ `UINT_MAX` (`UINT_MAX` from `<limits.h>`).

  Less formally, `uniqueN("S", N)` allocates a sequence of `N` consecutive numbers distinct from all other sequences allocated for identifier `S`, returns the smallest value from the sequence, and guarantees that the sequences are compact (start at 0, no gaps between sequences).

  A use of `unique(S)` is short for `uniqueN(S, 1)`.

[7] The time at which the value of unknown constants become known is unspecified by this language definition.

The expansion of `uniqueN` calls happens after generic component instantiation (Section 14.2): calls to `uniqueN` in generic components return a different value in each instantiation.

For purposes of checking constant expressions, `uniqueN(`s`, `n`)` behaves as if it were an unknown constant.

Using `unique`, a component providing a service (defined by interface `X`) can uniquely identify its clients with the following idiom:

```
module XService {
  provides interface X[uint8_t id];
} implementation { ... }

module UserOfX {
  uses interface X;
} implementation { ... }

configuration ConnectUserToService { }
implementation {
  components XService, UserOfX;

  UserOfX.X -> XService.X[unique("X")];
}
```

Each client of `XService` will be connected to interface `X` with a different `id`.

- `unsigned int uniqueCount(char *identifier)`

  `uniqueCount(`s`)` returns the sum of all `nb` parameters for all uses of `uniqueN(`s`, nb)`, or 0 if there are no calls to `uniqueN(`s`)`. For purposes of checking constant expressions, `uniqueCount(`s`)` behaves as if it were an unknown constant.

  The intended use of `uniqueCount` is for dimensioning arrays (or other data structures) which will be indexed using the numbers returned by `unique` and `uniqueN`. For instance, a `Timer` service that identifies its clients (and hence each independent timer) via a parameterised interface and `unique` can use `uniqueCount` to allocate the correct number of timer data structures.

In the following example:

```
generic module A() { }
implementation {
  int x = unique("A");
  int y = uniqueCount("A");
}
configuration B { }
implementation {
  components new A() as A1, new A() as A2;
}
```

`B.A1.y = B.A2.y = 2` and either `B.A1.x = 0, B.A2.x = 1` or `B.A1.x = 1, B.A2.x = 0`.

13.3 Type Parameters and C Type Checking

Generic interface and component definitions can have type parameters. Syntactically, type parameters behave the same as `typedef`'d identifiers. When a generic component or interface is instantiated, the type parameter will be replaced with the argument type, which cannot be incomplete, of function or of array type. The size and alignment of a type parameter are an unknown constant (Section 13.1). The rules for assignment and type equivalence for a type parameter t are simple: a value of type t is assignable to an lvalue of type t (extends A.7.17 in K&R [3]) and type t is only equivalent to itself (extends A.8.10 in K&R [3]).

If a type parameter t has the `@number()` attribute (Section 11), the corresponding argument must be a numerical (integral or floating-point) type, and all numerical operations (i.e., those valid for floating-point types) are allowed on type t.

If a type parameter t has the `@integer()` attribute (Section 11), the corresponding argument must be an integral type, and all integral operations are allowed on type t.

13.4 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).

13.5 // comments

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

14 nesC Applications

A nesC application has two executable parts: C declarations and definitions, and a set of components (non-generic components and instantiated generic components). The components are connected to each other via interfaces specified by a set of interface definitions.

The C declarations and definitions, interfaces and components that form a nesC application are determined by an on-demand loading process. The input to the nesC compiler is a single non-generic component K. The nesC compiler first loads a user-specified set of C files[8] (Section 14.1), then loads the component definition for K (Section 14.2). The resulting program contains:

- All C declarations from the initially loaded C files (Section 14.1).
- All C declarations from all component and interface definitions (Sections 14.2 and 14.3).
- All components output by the rules of Section 14.2.

[8] `ncc`, the TinyOS frontend for nesC always loads the TinyOS `tos.h` file.

Section 14.4 discusses the interactions between nesC and the C preprocessor. The external linkage rules for a compiled nesC program are given in Section 14.5. The process by which C files, nesC component and interface definitions are located is outside the scope of this reference manual; for details see the `ncc` and `nescc` man pages.

14.1 Loading C file *X.h*

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, interfaces and components.

The nesC keywords are not reserved when a C file is loaded in this fashion.

14.2 Loading Component Definition *K*

If *K* has already been loaded, nothing more is done. Otherwise, file *K*.nc is located and preprocessed. Changes made to C macros (via `#define` and `#undef`) before the `component`, `module` and `configuration` keyword are preserved and visible to all subsequently loaded files; changes made after this point are discarded. The preprocessed file is parsed using the following grammar (*translation-unit* is a list of C declarations and function definitions):

nesC-file:
translation-unit$_{opt}$ *interface*
translation-unit$_{opt}$ *module*
translation-unit$_{opt}$ *configuration*

Note that the nesC keywords are reserved while parsing the C code in *translation-unit*. If *K*.nc does not define module *K* or configuration *K*, a compile-time error is reported.

The component's definition is then processed (Sections 5.3, 9, and 7). All referenced component and interface definitions are loaded (see also Section 14.3) during this processing. C declarations and definitions from a referenced component or interface definition *D* are available after the first reference to *D*. Note however that macros defined in *D* are not available in *K* as *K* was already preprocessed (see Section 14.4 for more discussion of macros in nesC).

Finally, the set of components output by *K* is defined by the following algorithm:

Expand(*K*):

- If *K* is a generic component, no component is output.
- If *K* is a non-generic module, *K* is output.
- If *K* is a non-generic configuration: for each component instantiation `new` $L(a_1, \ldots, a_n)$ in *K*, a new component *X* is created according to the rules of Section 6.1 and Expand(*X*) is called recursively (instantiating further generic components if *L* contained component instantiations). Then *K* is output.

14.3 Loading Interface Definition *I*

If *I* has already been loaded, nothing more is done. Otherwise, file *I*.nc is located and preprocessed. Changes made to C macros (via `#define` and `#undef`) before the `interface` keyword are preserved and visible to all subsequently loaded files; changes made after this point are discarded. The preprocessed file is parsed following the *nesC-file* production above. If *I*.nc does not define `interface` *I* a compile-time error is reported. Then *I*'s definition is processed (Section 5).

14.4 nesC and the C Preprocessor

During preprocessing, nesC defines the `NESC` symbol to a number XYZ which identifies the version of the nesC language and compiler. For nesC 1.2, XYZ is at least 120.[9]

The loading of component and interface definitions is driven by syntactic rules; as a result it must happen after preprocessing. Thus if a component *X* references, e.g., an interface *I*, macros defined in *I* cannot be used in *X* even though *I*'s C declarations can be. We suggest the following structure to avoid confusion:

1. All C declarations, function definitions and macros should be placed in a `.h` file, e.g., `I.h`. This file should be wrapped in the usual `#ifndef I_H` / `#define I_H` / `#endif` way.
2. The file(s) with which the `.h` file is naturally associated (e.g., an interface `I`) should `#include "I.h"`).
3. Files which wish to use the macros defined in the `.h` file should `#include` it.
4. Files which wish to use the C declarations and definitions from the `.h` file should `#include` it if they do not reference one of the components or interfaces from point 2.

These rules are similar to how `#include` is typically used in C.

14.5 External Linkage Rules

The following rules specify the external visibility of symbols defined in a nesC program:

- The external linkage of C variable declarations is the same as for C (note that this does not include variables declared inside modules).
- All function definitions marked with `spontaneous`, `hwevent` or `atomic_hwevent` attributes (Section 11) are external definitions.
- All used commands and events of binary components are external definitions.
- All non-static C function declarations without a definition are external references.
- All provided commands and events of binary components are external references.

[9] The `NESC` symbol was not defined in versions of nesC prior to 1.1.

The external names of C declarations, and of function definitions inside modules using the `C` attribute, are the same as the corresponding C name. The external names of all other externally visible symbols is implementation-defined.[10]

The nesC compiler can assume that only code reachable from external definitions will be executed (i.e., there are no "invisible" calls to any other functions).[11]

A Grammar

Please refer to Appendix A of Kernighan and Ritchie (K&R) [3, pp234–239] while reading this grammar (see the "Imported rules", Section A.1, for a quick summary of references to the K&R grammar).

The following additional keywords are used by nesC: `as`, `atomic`, `async`, `call`, `command`, `component`, `components`, `configuration`, `event`, `generic`, `implementation`, `includes`, `interface`, `module`, `new`, `norace`, `nx_struct`, `nx_union`, `post`, `provides`, `signal`, `task`, `uses`. The following keywords are reserved for future use: `abstract` and `extends`.

nesC reserves all identifiers starting with `__nesc` for internal use.

nesC files follow the *nesC-file* production; `.h` files loaded before the program's main component (see Section 14) follow the *translation-unit* directive from K&R and do not reserve any of the nesC keywords except for `nx_struct` and `nx_union`.

New rules:

nesC-file:
translation-unit$_{opt}$ *interface-definition*
translation-unit$_{opt}$ *component*

interface-definition:
`interface` *identifier type-parameters*$_{opt}$ *attributes*$_{opt}$ { *declaration-list* }

type-parameters:
`<` *type-parameter-list* `>`

type-parameter-list:
identifier attributes$_{opt}$
type-parameter-list `,` *identifier attributes*$_{opt}$

component:
comp-kind identifier comp-parameters$_{opt}$ *attributes*$_{opt}$ *component-specification implementation*$_{opt}$

comp-kind:

[10] The current nesC compiler uses "componentname$functionname".
[11] The current nesC compiler uses this information to eliminate unreachable code.

`module`
`component`
`configuration`
`generic module`
`generic configuration`

implementation:
module-implementation
configuration-implementation

comp-parameters:
`(` *component-parameter-list* `)`

component-parameter-list:
component-parameter
component-parameter-list `,` *component-parameter*

component-parameter:
parameter-declaration
`typedef` *identifier attributes*$_{opt}$

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

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

configuration-element-list:
configuration-element
configuration-element-list configuration-element

configuration-element:
components
connection
declaration

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

component-line:
component-ref instance-name$_{opt}$
component-line `,` *component-ref instance-name*$_{opt}$

instance-name:
`as` *identifier*

component-ref:
identifier
`new` *identifier* `(` *component-argument-list* `)`

component-argument-list:
component-argument
component-argument-list `,` *component-argument*

component-argument:
expression
type-name

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

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

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

component-specification:
{ *uses-provides-list* }

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

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

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

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

specification-element:
declaration
interface-type instance-name$_{opt}$ *instance-parameters*$_{opt}$ *attributes*$_{opt}$

interface-type:
`interface` *identifier type-arguments*$_{opt}$

type-arguments:
`<` *type-argument-list* `>`

type-argument-list:
type-name
type-argument-list `,` *type-name*

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

attributes:
attributes attribute
attribute

attribute:
`@` *identifier* `(` *initializer-list* `)`

Changed rules:

typedef-name: also one of
identifier `.` *identifier*
storage-class-specifier: also one of
`command event async task norace`

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

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

struct-or-union-specifier: also one of
`struct @` *identifier* { *struct-declaration-list* }
struct-or-union identifier attributes { *struct-declaration-list* }

struct-or-union: also one of

`nx_struct`
`nx_union`

enum-specifier: also one of
`enum` *identifier attributes* { *enumerator-list* }

init-declarator: also
declarator attributes
declarator attributes `=` *initializer*

struct-declarator: also
declarator attributes
declarator `:` *constant-expression attributes*

parameter-declaration: also
declaration-specifiers declarator attributes

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

statement: also
atomic-statement

atomic-statement:
`atomic` *statement*

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`

Note that like like regular typedefs, the extended rule for *typedef-name* (to refer to types from other components) cannot be directly used in a LALR(1) parser.

A.1 Imported Rules

This list is for reference purposes only:

- *argument-expression-list*: A list of comma-separated expressions.
- *compound-stmt*: A C { } block statement.
- *declaration*: A C declaration.
- *declaration-list*: A list of C declarations.
- *declaration-specifiers*: A list of storage classes, type specifiers and type qualifiers.
- *declarator*: The part of a C declaration that specifies the array, function and pointer parts of the declared entity's type.
- *direct-declarator*: Like *declarator*, but with no leading pointer-type specification.
- *enumerator-list*: List of constant declarations inside an `enum`.
- *expression*: Any C expression.
- *identifier*: Any C identifier.
- *init-declarator*: The part of a C declaration that specifies the array, function and pointer parts of the declared entity's type, and its initialiser (if any).
- *initializer*: An initializer for a variable declaration.
- *initializer-list*: An initializer for a compound type without the enclosing {, kw}.
- *parameter-declaration*: A function parameter declaration.
- *parameter-type-list*: Specification of a function's parameters.
- *postfix-expression*: A restricted class of C expressions.
- *primary*: An identifier, constant, string or parenthesised expression.
- *statement*: Any C statement.
- *storage-class-specifier*: A storage class specification for a C declaration.
- *struct-declaration-list*: Declarations inside a `struct` or `union`.
- *translation-unit*: A list of C declarations and function definitions.
- *type-name*: A C type specification.

B Glossary

- *attribute*: a user-specified decoration that can be placed on C and nesC declarations. Attributes must be declared (see *attribute kind*).
- *attribute kind*: a declaration of an attribute, which specifies the attribute's arguments.
- *bare command, bare event*: See *command*.
- *binary component*: a component provided in binary rather than source code form. Binary components cannot be generic.
- *combining function*: C function that combines the multiple results of a command call (or event signal) in the presence of *fan-out*.
- *command*, *event*: A function that is part of a component's *specification*, either directly as a *bare* command or event, or within one of the component's *interfaces*.

  Bare commands and events have roles (*provider*, *user*) and can have *instance parameters*. When these parameters are present, the command or event is known as a *bare, parameterised* command or event. The instance parameters of a command or event are distinct from its regular function parameters.
- *compile-time error*: An error that the nesC compiler must report at compile-time.
- *component*: The basic unit of nesC programs. Components have a name and are of two kinds: *generic* components, which take type and constant parameters and must be instantiated before they can be used, and non-generic components which exist implicitly in a single instance. A component has a *specification* and an implementation. A *module* is a component whose implementation is C code; a *configuration* is a component whose implementation is built by selecting or instantiating other components, and *wiring* them together.
- *configuration*: A component whose implementation is built by selecting or instantiating other components, and *wiring* them together.
- *endpoint*: A specification of a particular specification element, and optionally some instance arguments, in a wiring statement of a configuration. A parameterised endpoint is an endpoint without instance arguments that corresponds to a parameterised specification element.
- *event*: See *command*.
- *extent*: The lifetime of a variable. nesC has the standard C extents: *indefinite*, *function*, and *block*.
- *external*: In a configuration C, describes a specification element from C's specification. See internal.
- *external type*: a special kind of type with a platform-independent representation and no alignment restrictions.
- *fan-in*: Describes a provided command or event called from more than one place.

- *fan-out*: Describes a used command or event connected to more than one command or event implementation. A *combining function* combines the results of calls to these used commands or events.

- *generic*: See *component*, *interface*.

- *interface*: An instance of a particular *interface type* in the *specification* of a component. An interface has a name, a role (*provider* or *user*), an *interface type* and, optionally, *instance parameters*. An interface with parameters is a *parameterised interface*.

- *interface definition*: An *interface definition* 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 definition has a distinct name, and may optionally take type parameters. Interface definitions with type parameters are called *generic interface definitions*. Argument types must be provided before a generic interface definition can be used as an *interface type*.

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

- *interface type*: A reference to an interface definition, along with argument types if the definition is generic. *Configurations* can only connect two interface instances if they have the same interface type and instance parameters.

- *instance parameter*: An instance parameter is a parameter added to an interface type. It has a name and must be of integral type.

  There is (conceptually) a separate interface for each distinct list of instance parameter values of a *parameterised interface* (and, similarly, separate commands or events in the case of parameterised commands or events). In a module, parameterised interfaces, commands, events allow runtime selection or a `call` or `signal` target.

- *intermediate function*: A pseudo-function that represents the behaviour of the commands and events of a component, as specified by the wiring statements of the whole application. See Section 9.6.

- *internal*: In a configuration C, describes a specification element from one of the components specified in C's component list. See external.

- *module*: A component whose implementation is provided by C code.

- *name space*: nesC has the standard C *object* (variables, functions, typedefs, enum-constants), *tag* (`struct`, `union` and `enum` tags) and *label* (goto labels) name spaces. Additionally, nesC has a *component* name space for component and interface definitions.

- *parameterised command, parameterised event, parameterised interface, parameterised endpoint*: See command, event, interface instance, endpoint.

- *provided, provider*: A role for a specification element. A module K must implement the *provided commands of* K and *provided events of* K.

- *provided command of* K: A command that is either a provided specification element of K, or a command of a provided interface of K.

- *provided event of K*: An event that is either a provided specification element of K, or an event of a used interface of K.
- *scope*: nesC has the standard C *global*, *function-parameter* and *block* scopes. Additionally there is a *component parameter*, *specification* and *implementation* scope for each component and an *interface parameter* and *interface* scope for each interface. Scopes are divided into *name spaces*.
- *specification*: A list of *specification elements* that specifies the interaction of a component with other components.
- *specification element*: An *interface*, *bare command* or *bare event* in a specification. Specification elements are either *provided* or *used*.
- *task*: A TinyOS task representing an independent thread of control whose execution is requested by the application and initiated by the TinyOS scheduler.
- *used, user*: A role for a specification element.
- *used command of K*: A command that is either a used specification element of K, or a command of a used interface of K.
- *used event of K*: An event that is either a used specification element of K, or an event of a provided interface of K.
- *wiring*: The connections between component's specification elements specified by a configuration.

References

[1] JSR 175: A Metadata Facility for the Java Programming Language. `http://jcp.org/aboutJava/communityprocess/review/jsr175/`.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[3] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988.