TinyOS Programming

Philip Levis

June 28, 2006

Revision 1.2 (June 28 2006)

Contents

Code Examples

Chapter 1

Preface

This book provides a brief introduction to TinyOS programming for TinyOS 2.0. While it goes into greater depth than the tutorials, there are several topics that are outside its scope, such as the structure and implementation of radio stacks or existing TinyOS libraries. It focuses on how to write nesC code, and explains the concepts and reasons behind many of the nesC and TinyOS design decisions. If you are interested in a brief introduction to TinyOS programming, then you should probably start with the tutorials, and if you're interested in details on particular TinyOS subsystems you should probably consult TEPs. Both of these can be found in the `doc/html` directory of a TinyOS distribution.

While some of the contents of this book are useful for 1.x versions of TinyOS, they do have several differences from 2.0 which can lead to different programming practices. If in doubt, referring to the TEP on the subject is probably the best bet, as TEPs often discuss in detail the differences between 1.x and 2.0.

For someone who has experience with C or C++, writing simple nesC programs is fairly simple: all you need to do is implement one or two modules and wire them together. The difficulty (and intellectual challenge) comes when building larger applications. TinyOS modules are fairly analogous to C coding, but configurations – which stitch together components – are not.

This book is a first attempt to explain how nesC relates to and differs from other C dialects, stepping through how the differences lead to very different coding styles and approaches. As a starting point, this book assumes that

1. you know C, C++, or Java reasonably well, understand pointers and that

2. you have taken an undergraduate level operating systems class (or equivalent) and know about concurrency, interrupts and preemption.

Of course, this book is as much a description of nesC as it is an argument for a particular way of using the language to achieve software engineering goals. In this respect, it is the product of thousands of hours of work by many people, as they learned and explored the use of the language. In particular, David Gay and Cory Sharp have always pushed the boundaries of nesC programming in order to better understand which practices lead to the simplest, most efficient, and robust code. In particular, Chapter 8 is an edited version of a paper David Gay and I wrote together, while using structs as a compile-time checking mechanism in interfaces (as Timer does) is an approach that invented by Cory.

Chapter 2

Programming Hints, Condensed

Throughout the text, there are a number of programming hints that touch on ways to avoid common issues that arise in nesC programs. They are collected here for easy reference.

Programming Hint 1: It's dangerous to signal events from commands, as you might cause a very long call loop, corrupt memory and crash your program.

Programming Hint 2: Keep tasks short.

Programming Hint 3: Keep code synchronous when you can. Code should be async only if its timing is very important or if it might be used by something whose timing is important.

Programming Hint 4: Keep atomic sections short, and have as few of them as possible. Be careful about calling out to other components from within an atomic section.

Programming Hint 5: Only one component should be able to modify a pointer's data at any time. In the best case, only one component should be storing the pointer at any time.

Programming Hint 6: Allocate all state in components. If your application requirements necessitate a dynamic memory pool, encapsulate it in a component and try to limit the set of users.

Programming Hint 7: Conserve memory by using enums rather than const variables for integer constants, and don't declare variables with an enum type.

Programming Hint 8: In the top-level configuration of a software abstraction, auto-wire Init to MainC. This removes the burden of wiring Init from the programmer, which removes unnecessary work from the boot sequence and removes the possibility of bugs from forgetting to wire.

Programming Hint 9: If a component is a usable abstraction by itself, its name should end with C. If it is

intended to be an internal and private part of a larger abstraction, its name should end with P. Never wire to P components from outside your package (directory).

Programming Hint 10: Use the *as* keyword liberally.

Programming Hint 11: Never ignore combine warnings.

Programming Hint 12: If a function has an argument which is one of a small number of constants, consider defining it as a few separate functions to prevent bugs. If the functions of an interface all have an argument that's almost always a constant within a large range, consider using a parameterized interface to save code space. If the functions of an interface all have an argument that's a constant within a large range but only certain valid values, implement it as a parameterized interface but expose it as individual interfaces, to both minimize code size and prevent bugs.

Programming Hint 13: If a component depends on unique, then #define a string to use in a header file, to prevent bugs from string typos.

Programming Hint 14: Never, ever use the "packed" attribute.

Programming Hint 15: Always use platform independent types when defining message formats.

Programming Hint 16: If you have to perform significant computation on a platform independent type or access it many (hundreds or more) times, then temporarily copying it to a native type can be a good idea.

Chapter 3

Linking and Namespaces

Programming TinyOS can be challenging because it requires using a new language, nesC. On one hand, nesC appears to be very similar to C. Implementing a new system or protocol doesn't involve climbing a steep learning curve. Instead, the difficulties begin when trying to incorporate new codes with existing ones. The place where nesC differs greatly from C is in its linking model. The challenge and complexity isn't in writing software components, but rather in combining a set of components into a working application. In order to understand why this is often difficult, it's useful for us revisit how linking works in C, C++, and Java and how this affects the structure of well-written code in those languages. Then, when we examine how linking works in nesC, the differences will be apparent. The purpose here isn't to go into the details of symbol tables and virtual functions – linking from a systems standpoint – rather to talk about linking from a programmer standpoint.

3.1 C

C programs have a single global namespace for functions and variables (we'll just call both of them variables for simplicity). A source file can name a variable in one of three ways: declaration, definition, or a reference. A *declaration* states that a variable exists and gives, among other things, information about its type. A variable can declared multiple times as long as the declarations agree: For example, this is a declaration of a basic POSIX system call:

```
int read(int fd, void* buf, size_t count);
```

Listing 3.1: The POSIX read(2) function

A declaration is not an implementation: it merely states that a variable exists somewhere, and so other code can reference it. Calling a function, assignment, taking an address, or loading are all *references*. This code, for example, references the variables result, read, and val:

```
result = read(fd, &val, 1);
```

Listing 3.2: Calling read(2)

A C compiler generally expects variables to be declared before being referenced. There are some exceptions (some compilers only issue a warning if you reference an undeclared function and hope they'll come across the declaration later), but referenced undeclared variables is usually just bad C programming.

Definition is the final kind of naming. A declaration claims that a variable exists, a reference uses the variable, and a definition actually creates it. A function definition is its implementation, while a variable definition is its allocation. A variable can be declared many times and referenced many times, but defined only once. For example, this code declares the read function, defines the readByte and myFd variables, and references many more:

```
int read(int fd, void* buf, size_t count);
int myFd; // initialized elsewhere
char readByte() {
   char val = 0;
   read(myFd, &val, 1);
   return val;
}
```

Listing 3.3: Reading a byte with read(2)

Defining a variable introduces a name into the program's namespace. A C program namespace has many *scopes*. At the top level, visible to all, is the global scope: any code can reference a variable in global scope. Global variables and non-static functions are in global scope. A program has a tree of naming scopes, whose root is the global scope. Curly braces — { and } — define child scopes, and code in a scope can only reference variables named in scopes above it in the tree. For example, here's a snippet of C and a figure showing its scope tree:

```
// Scope A
int var;
void foo() { // Scope B
  int test = 0;
  if (var) { // Scope C
    int randVal = rand();
    test = randVal;
```

```
    var = randVal;
  }
}
int bar() { // Scope D
  int test;
  return 4;
}
int snark() { // Scope E
  int var;
  return 5;
}
```

Listing 3.4: Naming scopes in C

In this example, the code in scope C can reference randVal, test, and var, while the code in scope B can only reference test and var. While both scope B and scope D introduce the same name, test, they do not conflict. Scope E, however, introduces a variable with the name var, which conflicts with the var in global scope. This is commonly called shadowing and while legal, usually causes a compiler to issue a warning. There is one additional level of scope, provided by the static keyword: file scope. A variable with the static keyword can be referenced by code in the same file, but does not enter the global scope.

Header files are not a special C construct: they are just a kind of C source file whose use has evolved from programming practice. Specifically, C header files generally contain only variable declarations. These declarations match the definitions of the corresponding implementation file or object file.

When a C program is broken up into many files, those files can only reference each other through declarations introduced into global scope. If an object file references but does not define a variable (a symbol), then in the linking stage it has to be linked with another object file that defines it. This can be a library or a standard object file. Because a declaration can only have one definition, this means that when code references a function, for example, it references an implementation. Two source files that reference the same function name reference the same function, introducing an unforeseen dependency between the two codes, because if you want to change the implementation that one uses, you have to change the implementation the other uses as well.

Function pointers are the common approach to avoid this undesirable binding. Rather than reference a specific function, a code can reference a variable that stores a pointer to any function. This allows the code to resolve the binding at runtime, by choosing what pointer to store in that variable. Using function pointers allows C software to call new extensions that it could not name when it was compiled, which is critical for any system with callbacks.

For example, C GUI toolkits need to be able to call functions in response to user events. The function with the behavior is part of the application code, but the pre-compiled GUI toolkit needs to be able to call

it. So when the application creates a button, for example, it gives the button a function pointer for it to call when clicked. The button structure stores this pointer in RAM. There is no other way to do this effectively in C: because the toolkit cannot name the function it must call, a function pointer must be used, and this pointer must be assigned at runtime.

Another example of this is the virtual file system that UNIX/Linux kernels use. The OS might need to access any number of file systems, and statically compiling all of them into the core kernel is problematic (imagine having to recompile Windows just because USB storage devices become popular). So what the OS does is interact with file systems through a VFS — virtual file system — interface. This is a set of basic operations, such as read() and write(), which are stored in a structure. Every file system implementation populates the structure with its own functions, then passes the structure to the core kernel. When a user program tries to read from a CD, for example, the OS gets the VFS structure associated with a CD file and calls the read() function pointer, allowing the core kernel to remain independent of specific file systems. The basic approach is the same as GUI toolkits. Because of C's naming approach, the OS has to use run-time allocation and function pointers in order to enable flexible composition and extension.

3.2 C++

C++ is similar to C, except that it richer namespaces and inheritance. In C++ the :: (double colon) generally refers to a namespace hierarchy. For example, you can define a namespace, or a hierarchy of namespaces:

```
namespace test;
namespace test::more;
```

Listing 3.5: C++ namespaces

A class also defines a namespace:

```
void MyClass::doSomething() {}
```

Listing 3.6: The namespace of a C++ class

While classes can use a variety of access modifiers on their fields and methods, these do not operate at the namespace level. In this way they are different than the static keyword in C. In C, static means that the variable or function does not exist in the global scope, and so cannot be named: other files that try to do so will get a "no such variable exists" error. In contrast, specifying a variable to be private in a C++ class means that other codes can reference it, but doing so generates a compile error. For example:

```
decl.c:
static int x;

use.c:
x = 5;   // compile error: no such variable!

NewClass.C:
class NewClass {
  private int x;
}

OtherClass.C:
NewClass* nc = new NewClass();
nc.x = 4; // compile error: access violation
```

Listing 3.7: Private fields in C++

Inheritance provides a way to extend functionality more easily than is possible than in C. Rather than using a structure of function pointers, a program can use a class to represent an extensible abstraction. In practice, each object has a pointer to a function table of its functions, but this table can be in read-only memory and the programmer does not have to maintain it. Rather than use function pointers, a C++ system can use objects to extend functionality. As with C, however, the association occurs at runtime.

C's name binding leads to design patterns such as the factory being common object oriented software engineering practice. A factory decouples an object from its instantiation. For example, a Tetris game might need to create game pieces. A simple way to do this would be to have the game engine allocate the pieces directly:

```
Piece* piece = new JPiece();
```

Listing 3.8: Instantiating a Piece object in C++

But this binds the game engine to a specific class, JPiece, as well as the parameter list of JPiece's constructor. This binding is unnecessary and can be problematic when, for example, it turns out that the desired class name changes to JTetrisPiece. Rather than binding itself to a specific set of piece classes, the game engine can define an abstract class Piece and just ask a factory to create the pieces for it:

```
class PieceFactory {
 public:
  Piece* createIPiece();
  Piece* createJPiece();
  Piece* createBlockPiece();
  Piece* createSPiece();
  Piece* createZPiece();
  Piece* createTPiece();
```

```
}
```

Listing 3.9: A C++ Piece factory

Now the game engine is independent of the piece implementations and their names: changing the set of pieces used only requires changing the factory, and as there is a single allocation point any changes to the constructors can be localized to the factory class. Design patterns such as the factory introduce a level of naming indirection that breaks undesirable coupling between specific classes.

3.3 Java

With regards to naming and name binding, Java has many similarities to C and C++. Because instantiation points name a specific class, it's common to use design patterns such as the factory in order to decouple classes. Similarly, even more so than C++, many classes take objects as parameters. For example, java.io.BufferedInputStream takes a java.io.InputStream as a parameter in its constructor. This allows a program to associate a BufferedInputStream with any sort of input stream.

As with C and C++, this association occurs at runtime. Consider, for example, this very common piece of Java code for very simple GUIs:

```
JButton b = new JButton("Quit");
b.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    System.exit(0);
  }
}
```

Listing 3.10: Function binding in Java

On one hand, your class always instantiates a JButton, and always makes its action to call System.exit(): this logic and composition is static. However, Java naming in the C and C++ tradition, requires this association to be dynamic: the only good way to add functionality is through dynamically adding an action listener (there's no "clicked" method that you can overload).

3.4 Components and Interfaces

nesC (network embedded system C) is a component-based C dialect. In some ways, nesC components are similar to objects. For example, they encapsulate state and couple state with functionality. The principal distinction lies in their naming scope. Unlike C++ and Java objects, which refer to functions and variables

in a global namespace, nesC components use a purely *local namespace*. This means that in addition to declaring the functions that it implements, a component must also declare the functions that it calls. The names that a component uses to call these functions is completely local: the name it references does not have to be the same that implements the function. When a component A declares that it calls a function B, it is essentially introducing the name A.B into a global namespace. A different component, C, that calls a function B introduces C.B into the global namespace. Even though both A and C refer to the function B, they might be referring to completely different implementations.

Every component has a *specification*, a code block that declares the functions it *provides* (implements) and the functions that it *uses* (calls). For example, this is the specification for a fictional component SmoothingFilterC, which smoothes raw data:

```
module SmoothingFilterC {
  provides command uint8_t topRead(uint8_t* array, uint8_t len);
  uses command uint8_t bottomRead(uint8_t* array, uint8_t len);
}
```

Listing 3.11: SmoothingFilterC, a simple nesC module

Because SmoothingFilterC provides the function topRead, it must define it and other components can call it. Conversely, because SmoothingFilterC uses bottomRead, it can reference the function and so depends on some other component to define it. Components can always reference functions that they define: SmoothingFilterC can call topRead on itself.

In practice, components very rarely declare individual functions in their specification. Instead, nesC has *interfaces*, which are collections of related functions. Component specifications are almost always in terms of interfaces. For example, power management and configuration issues mean that applications often need to be able to start and stop system abstractions and services, such as turning on a sensor to take a reading or turning on the radio stack to hear packets. The StdControl interface is a common way to express this functionality:

```
interface StdControl {
  command error_t start();
  command error_t stop();
}
```

Listing 3.12: The StdControl interface

A component representing an abstraction or service that can be turned on or off provides StdControl, while a component that needs to turn others on and off uses StdControl. This is often a hierarchical rela-

tionship. For example, a routing layer needs to start a data link packet layer, which in turn needs to start and stop idle channel detection:

```
module RoutingLayerC {
  provides interface StdControl;
  uses interface StdControl as SubControl;
}

module PacketLayerC {
  provides interface StdControl;
}
```

Listing 3.13: Interfaces in component signatures

Connecting providers and users together is called *wiring*. For example, RoutingLayerC's code has function calls to SubControl.start() and SubControl.stop(). Unless SubControl is wired to a provider, these functions are undefined symbols: they are not bound to any actual code. However, if SubControl is wired to PacketLayerC's StdControl, then when RoutingLayerC calls SubControl.start(), it will invoke PacketLayerC's StdControl.start(). This means that the reference RoutingLayerC.SubControl.start points to the definition PacketLayerC.StdControl.start. The two components RoutingLayerC and PacketLayerC are completely decoupled, and are only bound together when wired.

3.5 Why?

Having a global namespace requires C, C++, and Java to reply on dynamic composition. Referencing a function requires referencing a unique name in the global namespace, so code uses a level of indirection — function pointers or factories — in order to decouple one implementation from another.

nesC takes a different approach. First, code is broken up into components, discrete units of functionality. A component can only reference variables from its own, local namespace. A component can't name a variable in another component. However, a component can declare that it uses a function defined by another component. It can also declare that it provides a function that another component can call. Building a nesC program involves writing components and wiring users to providers. Because this composition occurs at compile-time, it doesn't require runtime allocation or storing function pointers in RAM. Additionally, since a program does not have these levels of indirection, the nesC compiler knows the complete call graph. Before wiring, a component could be calling any other component, and so is completely decoupled from whom it calls. However, in an application, the call is wired to a specific endpoint, and so the nesC compiler can optimize across the call boundary. In practice, nesC often inlines a call that crosses five or six components

into a flat instruction stream with no function calls.

TinyOS and nesC can take this approach because, unlike end-user computers, which need to be able to dynamically load new programs in response to user input, sensor networks are composed of embedded computers, which have well-defined and tightly specified uses. While these might evolve over time, this evolution is very slow in comparison to how often a PC loads new programs. There's an additional side to embedment which motivates using static, rather than dynamic approaches. Because traditional computers usually have some sort of user interaction, faults are more easily detected. If the server is acting funny, you notice, and reboot it. If your word processor is slow, you close it and reopen it. The same is not true for embedded systems, which operate, for the most part, unattended. For example, consider traditional computers that have roles resembling embedded systems, such as a dedicated mail server. If a logging failure causes the mail server to start losing some mail messages, it might — and often does — go unnoticed for a long time.

At the level of programming actual behavior or algorithms, nesC is usually quite accessible if you know one of the above languages. It is, after all, a C dialect. A component can only reference functions in its specification and only reference variables declared within it, but this usually isn't too hard to get a handle on. Understandably, the part that most people new to the language find most challenging is wiring, as there's no clear analogue in other system languages. Composing nesC components into working, flexible, and reusable systems requires learning a new language. While the language itself is very simple (it only has two operators, = and −>), the considerations are foreign to most programmers who are used to global namespaces. Therefore, while this book does go over implementations, it is mostly dedicated to wiring and composition, as these are what most nesC programmers find challenging.

Chapter 4

Interfaces and Modules

nesC has two kinds of components: configurations and modules. Configurations, discussed in Chapter 5, are how components are wired together. Modules, in contrast, are implementations. Configurations connect the declarations of different components, while modules define functions and allocate state.

4.1 Split Phase

Because sensor nodes have a broad range of hardware capabilities, one of the goals of TinyOS is to have a flexible hardware/software boundary. An application that encrypts packets should be able to interchangeably use hardware or software implementations. Hardware, however, is almost always *split-phase* rather than blocking. It is split-phase in that completion of a request is a callback. For example, to acquire a sensor reading with an analog-to-digital converter (ADC), software writes to a few configuration registers to start a sample. When the ADC sample completes, the hardware issues an interrupt, and the software reads the value out of a data register.

Now, let's say that rather than directly sampling, the sensor implementation actually samples periodically and when queried gives a cached value. This may be necessary if the sensor needs to continually calibrate itself. Magnetometer drivers sometimes do this due to the effect of the Earth's magnetic field, as two sensors oriented differently might have very different magnetometer floors. Drivers estimate the floor and essentially return a measure of recent change, rather than an absolute value. From a querying standpoint, the implementation of the sensor is entirely in software. This fact should not be apparent to the caller. For ease of composition, sampling a self-calibrating magnetometer should be the same as a simple photoresistor. But the magnetometer is a synchronous operation (it can return the result immediately) while the ADC is split-phase.

The basic solution to this problem is to make one of the two look like the other: either give the magnetometer a split-phase interface, or make the ADC synchronous by blocking. If the ADC interrupt is very fast, the ADC driver might be able to get away with a simple spin loop to wait until it fires. If the interrupt is slow, then this wastes a lot of CPU cycles and energy. The traditional solution for this latter case (e.g., in traditional OSes) is to use multiple threads. When the code requests an ADC sample, the OS sets up the request, puts the calling thread on a wait queue, starts the operation, and then schedules another thread to run. When the interrupt comes in, the driver resumes the waiting thread and puts it on the OS ready queue.

The problem with threads in embedded systems is that they require a good deal of RAM. Each thread has its own private stack which has to be stored when a thread is waiting or idle. E.g., when a thread samples a blocking ADC and is put on the wait queue, the memory of its entire call stack has to remain untouched so that when it resumes it can continue execution. RAM is a very tight resource on current sensor node platforms. Early versions of TinyOS ran in 512 bytes of RAM. When a thread is idle, its stack is wasted storage, and allocating the right sized stack for all of the threads in the system can be a tricky business. Additionally, while it is easy to layer threads on top of a split-phase interface, it is very difficult to do the opposite. Because it's a one-way street, while increasing amounts of RAM might allow threads at an application level, the bottom levels of TinyOS — the core operating system — can't require them, as they preclude chips with leaner RAM resources than high-end microcontrollers.

TinyOS therefore takes the opposite approach. Rather than make everything synchronous through threads, operations that are split-phase in hardware are split-phase in software as well. This means that many common operations, such as sampling sensors and sending packets, are split-phase. An important characteristic of split-phase interfaces is that they are bidirectional: there is a downcall to start the operation, and an upcall that signifies the operation is complete. In nesC, downcalls are generally *commands*, while upcalls are *events*. An interface specifies both sides of this relationship. For example, this is the basic TinyOS packet send interface, Send:

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t error);

  command error_t cancel(message_t* msg);
  command void* getPayload(message_t* msg);
  command uint8_t maxPayloadLength(message_t* msg);
}
```

Listing 4.1: The split–phase Send interface

Whether a component provides or uses the Send interface defines which side of the split-phase operation

it represents. A provider of Send defines the send and cancel functions and can *signal* the sendDone event. Conversely, a user of Send needs to define the sendDone event and can *call* the send and cancel commands. When a call to send returns SUCCESS, the msg parameter has been passed to the provider, which will try to send the packet. When the send completes, the provider signals sendDone, passing the pointer back to the user.

4.2 Interfaces with Arguments

Interfaces can take types as arguments. For example, Read is a simple interface for acquiring sensor readings:

```
interface Read<val_t> {
  command error_t read();
  event void readDone(error_t err, val_t t);
}
```

Listing 4.2: The Read interface

Type arguments to interfaces are in angle brackets. The Read interface has a single argument, which defines the type of the data value that it produces. For example, a magnetometer component that produces a 16-bit reading might look like this:

```
module MagnetometerC {
  provides interface StdControl;
  provides interface Read<uint16_t>;
}
```

Listing 4.3: Using the Read interface in MagnetometerC

When wiring providers and users of interfaces that have type arguments, they types must match. For example, you cannot wire a Read<uint8_t> to a Read<uint16_t>. Sometimes, arguments are used to enforce type checking that does not actually pertain to the arguments in commands or events. For example, the Timer interface takes a single parameter that is not in any of its functions:

```
interface Timer<precision_tag> {
  command void startPeriodic(uint32_t dt);
  command void startOneShot(uint32_t dt);
  command void stop();
  event void fired();

  command bool isRunning();
  command bool isOneShot();
```

```
  command void startPeriodicAt(uint32_t t0, uint32_t dt);
  command void startOneShotAt(uint32_t t0, uint32_t dt);
  command uint32_t getNow();
  command uint32_t gett0();
  command uint32_t getdt();
}
```

Listing 4.4: The Timer interface and its typing

The precision_tag argument is not used anywhere in the interface. Instead, it is used as a type check when wiring: the argument specifies the time units. Three standard types are TMilli, T32khz, and TMicro, for millisecond, 32khz, and microsecond timers. These types are each defined as C structs. This argument provides type checking because a component specification that includes

```
uses interface Timer<TMilli>;
```

Listing 4.5: Using the Timer interface

must be wired to a component that

```
provides interface<TMilli>;
```

Listing 4.6: Providing the Timer interface

Wiring to this would be a compile-time error:

```
provides interface<TMicro>;
```

Listing 4.7: Providing a Timer with microsecond precision

4.3 Module Implementation

Tying this all together, consider a fictional component PeriodicReaderC, which samples a 16-bit sensor value every few seconds. It provides one interface, StdControl, and uses two, Timer and Read:

```
module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
```

Listing 4.8: PeriodicReaderC signature

Every component has an implementation block after its signature. For modules, this implementation is similar to an object: it has variables and functions. A module must implement every command of interfaces it provides and every event of interfaces it uses. For example, this is a simple possible implementation of PeriodicReaderC:

```
module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
implementation {
  uint16_t lastVal = 0;

  command error_t StdControl.start() {
    return call Timer.startPeriodic(1024);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t err, uint16_t val) {
    if (err == SUCCESS) {
      lastVal = val;
    }
  }
}
```

Listing 4.9: Complete PeriodicReaderC component

This component periodically samples a sensor and stores the last valid reading in a local variable. Note how it achieves this with split-phase interfaces. The call to StdControl.start will start the timer. One second later, Timer.fired is signaled, the component calls Read.read and returns. At some point later, depending on the latency of the read operation, the data source signals Read.readDone, passing the reading as an argument.

As written, this component doesn't do anything with the reading besides store it. You could imagine, however, using an interface like Send in order to put a reading into a packet and send it:

```
module PeriodicSenseAndSendC {
  provides interface StdControl;
  uses {
    interface Timer<TMilli>;
    interface Read<uint16_t>;
    interface Send;
    interface Packet;
  }
}
```

```
implementation {

  message_t packet;
  bool busy = FALSE;

  command error_t StdControl.start() {
    return call Timer.startPeriodic(1024);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t err, uint16_t val) {
    if (err != SUCCESS || busy) {
      return;
    }
    else {
      uint8_t payloadLen;
      uint16_t* payload = (uint16_t*)call Packet.getPayload(\&packet, \&payloadLen);
      if (payloadLen >= sizeof(uint16_t)) {
        *payload = val;
        if (call Send.send(\&packet, sizeof(uint16_t)) {
          busy = TRUE;
        }
      }
    }
  }

  event void Send.sendDone(message_t* msg, error_t error) {
    busy = FALSE;
  }
}
```

Listing 4.10: PeriodicSenseAndSendC component

The Packet interface is an interface that lets a component get a pointer to where its payload region is within the packet. This allows a component to use underlying packet layers without having to know the size of their headers. Note the use of the busy flag: this is used to protect the message buffer from race conditions. For example, a very long sending delay might cause Timer.fired to be signaled again before Send.sendDone is signaled. In this case, if the busy flag were not used, then the payload of the packet might be modified (in Read.readDone) while in the midst of being sent, possibly corrupting the packet.

While modules have some similarities to objects, they also have significant differences. First and foremost, modules are, by default, singletons: you can't instantiate them. For example, there is only one PeriodicSenseAndSendC. This ties back to the goal of hiding whether a component is hardware or software. Hardware resources are singletons: you can't instantiate control registers or output pins. Therefore, software

generally is as well. There are such things as generic modules, which can be instantiated: they are presented and discussed in Chapter 7.

4.4 Tasks

Returning to the magnetometer/ADC example, depending on split-phase operations means that the magnetometer driver has to issue a callback. On one hand, it could just signal the event from within the call. However, signaling an event from within a command is generally a bad idea, because it can easily cause call loops. For example, consider this code:

```
module FilterMagC {
  provides interface StdControl;
  provides interface Read<uint16_t>;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t> as RawRead;
}
implementation {...}

module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
implementation {
  uint16_t filterVal = 0;
  uint16_t lastVal = 0;

  command error_t StdControl.start() {
    return call Timer.startPeriodic(10);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call Read.read();
  }
  event void Read.readDone(error_t err, uint16_t val) {
    if (err == SUCCESS) {
      lastVal = val;
      filterVal *=  9;
      filterVal /= 10;
      filterVal += lastVal / 10;
    }
  }

  command error_t Read.read() {
    signal Read.readDone(SUCCESS, filterVal);
  }
}
```

Listing 4.11: A troublesome implementation of PeriodicReaderC

On one hand, this approach is very simple and fast. On the other, it can lead to significant problems with the stack. Imagine, for example, a component, FastSamplerC, that wants to sample a sensor many times quickly (acquire a high frequency signal). It does this by calling Read.read in its Read.readDone handler:

```
event void Read.readDone(error_t err, uint16_t val) {
  buffer[index] = val;
  index++;
  if (index < BUFFER_SIZE) {
   call Read.read();
  }
}
```

Listing 4.12: Signal handler that can lead to an infinite loop

This means that there will be a long call loop between read and readDone. If the compiler can't optimize the function calls away, this will cause the stack to grow significantly. Given that motes often have limited RAM and no hardware memory protection, exploding the stack like this can corrupt data memory and cause the program to crash.

Programming Hint 1: It's dangerous to signal events from commands, as you might cause a very long call loop, corrupt memory and crash your program.

Of course, acquiring a high-frequency signal from our example Read implementation is a bit silly. As the implementation is caching a value, sampling it more than once isn't very helpful. But this call pattern — issuing a new request in an event signaling request completion — is a common one.

The problems caused by this signaling raise the question of how PeriodicReaderC is going to signal the readDone event. It needs a way to schedule a function to be called later (like an interrupt).

The right way to do this is with a *task*, a deferred procedure call. A module can *post* a task to the TinyOS scheduler. At some point later, the scheduler will execute the task. Because the task isn't called immediately, there is no return value. Also, because a task executes within the naming scope of a component, it doesn't take any parameters: any parameter you want to pass can just be stored in the component. Tasks, like functions, can be predeclared. This is what a declaration for a task looks like:

```
task void readDoneTask();
```

Listing 4.13: Declaring a task

A definition is like the declaration, but also includes a function body. A component posts a task to the TinyOS scheduler with the post keyword:

```
post readDoneTask();
```

Listing 4.14: Posting a task

This is how our data filter component might look like implemented with a task:

```
module FilterMagC {
  provides interface StdControl;
  provides interface Read<uint16_t>;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t> as RawRead;
}
module PeriodicReaderC {
  provides interface StdControl;
  uses interface Timer<TMilli>;
  uses interface Read<uint16_t>;
}
implementation {
  uint16_t filterVal = 0;
  uint16_t lastVal = 0;

  task void readDoneTask();

  command error_t StdControl.start() {
    return call Timer.startPeriodic(10);
  }
  command error_t StdControl.stop() {
    return call Timer.stop();
  }
  event void Timer.fired() {
    call RawRead.read();
  }
  event void RawRead.readDone(error_t err, uint16_t val) {
    if (err == SUCCESS) {
      lastVal = val;
      filterVal *=  9;
      filterVal /= 10;
      filterVal += lastVal / 10;
    }
  }

  command error_t Read.read() {
    post readDoneTask();
    return SUCCESS;
  }

  task void readDoneTask() {
    signal Read.readDone(SUCCESS, filterVal);
  }
```

```
}
```

Listing 4.15: An improved implementation of PeriodicReaderC

When FilterMagC.Read.read is called, FilterMagC posts treadDoneTask and returns immediately. At some point later, TinyOS runs the task, which signals Read.readDone.

Tasks are non-preemptive. This means that only one task runs at any time, and TinyOS doesn't interrupt one task to run another. Once a task starts running, no other task runs until it completes. This means that tasks run atomically with respect to one another. This has the nice property that you don't need to worry about tasks interfering with one another and corrupting each other's data. However, it also means that tasks should usually be reasonably short. If a component has a very long computation to do, it should break it up into multiple tasks. A task can post itself. For example, the basic execution loop of the Mate bytecode interpreter is a task that executes a few instructions of a thread and reposts itself.

It takes about 80 microcontroller clock cycles to post and execute a task. Generally, keeping task run times to at most a few milliseconds is a good idea. Because tasks are run to completion, then a long-running task or large number of not-so-long-running tasks can introduce significant latency (tens of milliseconds) between a task post and its execution. This usually isn't a big deal with application-level components. But there are lower-level components, such as radio stacks, that use tasks. For example, if the packet reception rate is limited by how quickly the radio can post tasks to signal reception, then a latency of 10ms will limit the system to 100 packets per second.

Consider these two cases. In both, there are five processing components and a radio stack. The mote processor runs at 8MHz. Each processing component needs to do a lot of CPU work. In the first case, the processing components post tasks that run for 5ms and respost themselves to continue the work. In the second case, the processing components post tasks that run for 500us and repost themselves to continue the work.

In the first case, the task posting overhead is 0.05%: 80 cycles overhead on 40,000 cycles of execution. In the second case, the task posting overhead is 0.5%: 80 cycles overhead on 4,000 cycles of execution. So the time to complete the executions isn't significantly different. However, consider the task queue latency. In the first case, when the radio stack posts a task to signal that a packet has been received, it expects to wait around 25ms (5 processing tasks x 5ms each), limiting the system to 40 packets per second. In the second case, when the radio stack posts the task, it expects to wait around 2.5ms (5 processing tasks x 500 us each), limiting the system to 400 packets per second. Because the task posting cost is so low, using lots of short running tasks improves the responsiveness of the system without introducing significant CPU overhead.

Of course, there's often a tradeoff between lots of short tasks and the amount of state you have to allocate in a component. For example, let's say you want to encrypt a chunk of data. If the encryption operation takes a while (e.g., 10 milliseconds), then splitting it into multiple task executions would improve the overall system responsiveness. However, if you execute it in a single task, then you can allocate all of the state and scratch space you need on the stack. In contrast, splitting it across tasks would require keeping this state and scratch space in the component. There is no hard rule on this tradeoff. But generally, long running tasks can cause other parts of the OS to perform poorly, so should be avoided when necessary.

Programming Hint 2: Keep tasks short.

The post operation returns an error_t. If the task is not already in the task queue, post returns SUCCESS. If the task is in the task queue (has been posted but has not run yet), post returns FAIL. In either case, the task will run in the future. Generally, if a component needs a task to run multiple times, it should have the task repost itself.

Note: one of the largest differences between TinyOS 1.x and 2.0 is the task model. In 1.x, tasks have a shared, fixed-size queue, and components can post a task multiple times. In 2.0, each task has a reserved slot. This means that in 1.x, a task post can return FAIL if the task is not already in the queue, and two posts back-to-back can both return SUCCESS.

Returning to the call to read, here are two possible implementations, which differ only slightly, but demonstrate very different semantics:

```
command error_t Read.read() {
  return post readDoneTask();
}
```

Listing 4.16: One–to–one binding of a split–phase call

versus

```
command error_t Read.read() {
  post readDoneTask();
  return SUCCESS;
}
```

Listing 4.17: Many–to–one binding of a split–phase call

The first represents a calling semantics where there is a one-to-one mapping between successful calls to read and the readDone event. The second represents a many-to-one calling semantics, where a single readDone event can correspond to many read requests. The question is whether the user of the interface is responsible for queueing based on whether Read.read returns SUCCESS, or whether it keeps internal queueing state. In the former approach, the user can't distinguish between "I'll be free when you get an event" and "I'm not free now, try later" unless it keeps state on whether it has a request pending. In the latter approach, if the user wants to queue it also needs to keep state on whether it has a request pending, as issuing another Read.read() can confuse it (number of command calls != number of event signals). While I personally prefer the latter approach, it's a matter of taste. There are plenty of developers who prefer the former approach. The important part is that an interface precisely states which semantics it follows.

4.5 Concurrency

Tasks allow software components to emulate the split-phase behavior of hardware. But they have much greater utility than that. They also provide a mechanism to manage preemption in the system. Because tasks run atomically with respect to one another, code that runs only in tasks can be rather simple: there's no danger of another execution suddenly taking over and modifying data under you. However, interrupts do exactly that: they interrupt the current execution and start running preemptively.

In nesC and TinyOS, functions that can run preemptively, from outside task context, are labeled with the async keyword: they run asynchronously with regards to tasks. A rule of nesC is that commands an async function calls and events an async function signals must be async as well. That is, it can't call a command or event that isn't async. A function that isn't asynchronous is synchronous (often call "sync" for short). By default, commands and events are sync: the async keyword specifies if they aren't. Interface definitions specify whether their commands and events are async or sync. For example, the Send interface is purely synchronous:

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t error);

  command error_t cancel(message_t* msg);
  command void* getPayload(message_t* msg);
  command uint8_t maxPayloadLength(message_t* msg);
}
```

Listing 4.18: The Send interface

In contrast, the Leds interface is purely asynchronous:

```
interface Leds {
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();

  async command void led1On();
  async command void led1Off();
  async command void led1Toggle();

  async command void led2On();
  async command void led2Off();
  async command void led2Toggle();

  async command uint8_t get();
  async command void set(uint8_t val);
}
```

Listing 4.19: The Leds interface

All interrupt handlers are async, and so they cannot include any sync functions in their call graph. The one and only way that an interrupt handler can execute a sync function is to post a task. A task post is an async operation, while a task running is sync.

For example, consider a packet layer on top of a UART. When the UART receives a byte, it signals an interrupt. In the interrupt handler, software reads the byte out of the data register and puts it in a buffer. When the last byte of a packet is received, the software needs to signal packet reception. But the receive event of the Receive interface is sync. So in the interrupt handler of the final byte, the component posts a task to signal packet reception.

This raises the question: If tasks introduce latency, why use them at all? Why not make everything async? The reason is simple: race conditions. The basic problem with preemptive execution is that it can modify state underneath an ongoing computation, which can cause a system to enter an inconsistent state. For example, consider this command, toggle, which flips the state bit and returns the old one:

```
bool state;
async command bool toggle() {
  if (state == 0) {
    state = 1;
    return 1;
  }
  if (state == 1) {
    state = 0;
    return 0;
  }
}
```

Listing 4.20: Toggling a state variable

Now imagine this execution, which starts with state = 0:

```
toggle()
  state = 1;
     -> interrupt
     toggle()
        state = 0
        return 0;
  return 1;
```

Listing 4.21: A call sequence that could corrupt a variable

In this execution, when the first toggle returns, the calling component will think that state is equal to 1. But the last assignment (in the interrupt) was to 0.

This problem can be much worse when a single statement can be interrupted. For example, on micaZ or Telos motes, writing or reading a 32 bit number takes more than one instruction. It's possible that an interrupt executes in between two instructions, so that part of the number read is of an old value while another part is of a new value.

This problem — data races — is particularly pronounced with state variables. For example, imagine this is a snippet of code from AMStandard, the basic packet abstraction in TinyOS 1.x, with a bunch of details omitted. The state variable indicates whether the component is busy.

```
command result_t SendMsg.send ... {
  if (!state) {
    state = TRUE;
    // send a packet
    return SUCCESS;
  }
  return FAIL;
}
```

Listing 4.22: State transition that is not async–safe

If this command were async, then it's possible between the conditional "if (!state)" and the assignment "state = TRUE" that another component jumps in and tries to send as well. This second call will see state to be false, set state to true, start a send and return SUCCESS. But then the first caller will result, send state to true again, start a send, and return SUCCESS. Only one of the two packets will be sent successfully, but barring additional error checks in the call path, it can be hard to find out which one, and this might introduce all kinds of bugs in the calling components. Note that the command isn't async.

Programming Hint 3: Keep code synchronous when you can. Code should be async only if its timing is very important or if it might be used by something whose timing is important.

The problems interrupts introduce means that programs need a way to execute snippets of code that won't be preempted. NesC provides this functionality through atomic statements. For example:

```
command bool increment() {
  atomic {
    a++;
    b = a + 1;
  }
}
```

Listing 4.23: Incrementing with an atomic section

The atomic block promises that these variables can be read and written atomically. Note that this does not promise that the atomic block won't be preempted. Even with atomic blocks, two code segments that do not touch any of the same variables can preempt one another. For example:

```
async command bool a() {
  atomic {
    a++;
    b = a + 1;
  }
}
async command bool c() {
  atomic {
    c++;
    d = c + 1;
  }
}
```

Listing 4.24: Incrementing with two independent atomic sections

In this example, c can (theoretically) preempt a without violating atomicity. But a can't preempt itself, nor can c preempt itself.

nesC goes further than providing atomic sections: it also checks to see whether variables aren't protected properly and issues warnings when this is the case. For example, if b and c from the prior example didn't have atomic sections, then nesC would issue a warning because of possible self-preemption. The rule for when a variable has to be protected by an atomic section is simple: if it is accessed from an async function, then it must be protected. nesC's analsyis is flow sensitive. This means that if you have a function that does not include an atomic block, but is always called from within an atomic block, the compiler won't issue a warning. Otherwise, you might have lots of unnecessarily nested atomic blocks. Usually, an atomic block

involves some kind of execution (e.g.. disabling an interrupt), so unnecessary atomics are a waste of CPU cycles. Furthermore, nesC removes redundant atomic blocks.

While you can make data race warnings go away by liberally sprinkling your code with atomic blocks, you should do so carefully. On one hand, an atomic block does have a CPU cost, so you want to minimize how many you have. On the other, shorter atomic blocks delay interrupts less and so improve system concurrency. The question of how long an atomic block runs is a tricky one, especially when your component has to call another component.

For example, the SPI bus implementation on the Atmega128 has a resource arbiter to manage access to the bus. The arbiter allows different clients to request the resource (the bus) and informs them when they've been granted it. However the SPI implementation doesn't want to specify the arbiter policy (e.g., first come first served vs. priority), so it has to be wired to an arbiter. This decomposition has implications for power management. The SPI turns itself off when it has no users, but it can't know when that is without calling the arbiter (or replicating arbiter state). This means that the SPI has to atomically see if it's being used, and if not, turn itself off:

```
atomic {
  if (!call ArbiterInfo.inUse()) {
    stopSpi();
  }
}
```

Listing 4.25: Atomically turning off the SPI bus

In this case, the call to isUse() is expected to be very short (in practice, it's probably reading a state variable). If someone wired an arbiter whose inUse() command took 1ms, then this could be a problem. The implementation assumes this isn't the case. Sometimes (like this case), you have to make these assumptions, but it's good to make as few as possible.

The most basic use of atomic blocks is for state transitions within a component. Usually, a state transition has two parts, both of which are determined by the existing state and the call: the first is changing to a new state, the second is taking some kind of action. Returning to the AMStandard example, it looks something like this:

```
if (!state) {
  state = TRUE;
  // send a packet
  return SUCCESS;
}
else {
```

```
  return FAIL;
}
```

Listing 4.26: State transition that requires a large atomic section

If state is touched by an async function, then you need to make the state transition atomic. But you don't want to put the entire block within an atomic section, as sending a packet could take a long enough time that it causes the system to miss an interrupt. So the code does something like this:

```
uint8_t oldState;
atomic {
  oldState = state;
  state = TRUE;
}
if (!oldState) {
  //send a packet
  return SUCCESS;
}
else {
  return FAIL;
}
```

Listing 4.27: A fast and atomic state transition

If state were already true, it doesn't hurt to just set it true. This takes fewer CPU cycles than the somewhat redundant statement of

```
if (state != TRUE) {state = TRUE;}
```

Listing 4.28: An unnecessary conditional

In this example, the state transition occurs in the atomic block, but then the actual processing occurs outside it, based on the state the component started in.

Let's look at a real example. This component is CC2420ControlP, which is part of the TinyOS 2.x CC2420 radio stack. CC2420ControlP is responsible for configuring the radio's various IO options, as well as turning it on and off. Turning the CC2420 radio has four steps:

1. Turn on the voltage regulator (0.6ms)
2. Acquire the SPI bus to the radio (depends on contention)
3. Start the radio's oscillator by sending a command over the bus (0.86ms)
4. Put the radio in RX mode (0.2ms)

Some of the steps that take time are split-phase and have async completion events (particularly, 1 and 3). The actual call to start this series of events, however, is SplitControl.start(), which is sync. One way to implement this series of steps is to assign each step a state and use a state variable to keep track of where you are. However, this turns out to not be necessary. Once the start sequence begins, it continues until it completes. So the only state variable you need is whether you're starting or not. After that point, every completion event is implicitly part of a state. E.g., the startOscillatorDone() event implicitly means that the radio is in state 3. Because SplitControl.start() is sync, the state variable can be modified without any atomic sections:

```
command error_t SplitControl.start() {
  if ( m_state != S_STOPPED )
    return FAIL;

  m_state = S_STARTING;
  m_dsn = call Random.rand16();
  call CC2420Config.startVReg();
  return SUCCESS;
}
```

Listing 4.29: The first step of starting the CC2420 radio

The startVReg() starts the voltage regulator. This is an async command. In its completion event, the radio tries to acquire the SPI bus:

```
async event void CC2420Config.startVRegDone() {
  call Resource.request();
}
```

Listing 4.30: The handler that the first step of starting the CC2420 is complete

In the completion event (when it receives the bus), it sends a command to start the oscillator:

```
event void Resource.granted() {
  call CC2420Config.startOscillator();
}
```

Listing 4.31: The handler that the second step of starting the CC2420 is complete

Finally, when the oscillator completion event is signaled, the component tells the radio to enter RX mode and posts a task to signal the startDone() event. It has to post a task because oscillatorDone is async, while startDone is sync. Note that the component also releases the bus for other users.

```
async event void CC2420Config.startOscillatorDone() {
```

```
  call SubControl.start();
  call CC2420Config.rxOn();
  call Resource.release();
  post startDone_task();
}
```

Listing 4.32: Handler that the third step of starting the CC2420 radio is complete

Finally, the task changes the radio's state from STARTING to STARTED:

```
task void startDone_task() {
  m_state = S_STARTED;
  signal SplitControl.startDone( SUCCESS );
}
```

Listing 4.33: State transition so components can send and receive packets

An alternative implementation could have been to put the following code in the startOscillatorDone() event:

```
atomic {
  m_state = S_STARTED;
}
```

Listing 4.34: An alternative state transition implementation

The only possible benefit in doing so is that the radio could theoretically accept requests earlier. But since components shouldn't be calling the radio until the startDone event is signaled, this would be a bit problematic. There's no chance of another task sneaking in between the change in state and signaling the event when both are done in the startDone_task.

Programming Hint 4: Keep atomic sections short, and have as few of them as possible. Be careful about calling out to other components from within an atomic section.

4.6 Allocation

Besides power, the most valuable resource to mote systems is RAM. Power means that the radio and CPU have to be off almost all the time. Of course, there are situations which need a lot of CPU or a lot of bandwidth (e.g., cryptography or binary dissemination), but by necessity they have to be rare occurances. In contrast, the entire point of RAM is that it's always there. The sleep current of the microcontrollers most motes use today is, for the most part, determined by RAM.

Modules can allocate variables. Following the naming scope rules of nesC, these variables are completely private to a component. For example, the PeriodicReaderC component allocated a lastVal and a filterVal, both of which were two bytes, for a total cost of 4 bytes of RAM. Because tasks run to completion, TinyOS does not have an equivalent abstraction to a thread or process. More specifically, there is no execution entity that maintains execution state beyond what is stored in components. When a TinyOS system is quiescent, component variables represent the entire software state of the system.

The only way that components can share state is through function calls, which are (hopefully) part of interfaces. Just as in C, are two basic ways that components can pass parameters: by value and by reference (pointer). In the first case, the data is copied onto the stack, and so the callee can modify it or cache it freely. In the second case, the caller and callee share a pointer to the data, and so components need to carefully manage access to the data in order to prevent memory corruption and memory leaks. While it's fine to pass pointers as arguments, you have to be very careful about storing pointers in a component. The general idea is that, at any time, every pointer should have a clear owner, and only the owner can modify the corresponding memory.

For example, abstract data types (ADTs) in TinyOS are usually represented one of two ways: generic modules or through an interface with by-reference commands. With a generic module, the module allocates the ADT state and provides accessors to its internal state. For example, many TinyOS components needs to maintain bit vectors, and so in tos/system there's a generic module BitVectorC that takes the number of bits as a parameter:

```
generic module BitVectorC( uint16_t max_bits ) {
  provides interface Init;
  provides interface BitVector;
}
```

Listing 4.35: Signature of BitVectorC

This component allocates the bit vector internally and provides the BitVector interface to access it:

```
interface BitVector {
  async command void clearAll();
  async command void setAll();
  async command bool get(uint16_t bitnum );
  async command void set(uint16_t bitnum );
  async command void clear(uint16_t bitnum );
  async command void toggle(uint16_t bitnum );
  async command void assign(uint16_t bitnum, bool value );
  async command uint16_t size();
}
```

Listing 4.36: The BitVector interface

With this kind of encapsulation, assuring that accesses to the data type are race-free is reasonably easy, as the internal implementation can use atomic sections appropriately. There is always the possibility that preempting modifications will lead to temporal inconsistencies with accessors. E.g., in BitVector, it's possible that, after a bit has been fetched for get() but before it returns, an interrupt fires whose handler calls set() on that same bit. In this case, get() returns after set(), but its return value is the value before set(). If this kind of interlacing is a problem for your code, then you should call get() from within an atomic section. Generic modules are discussed in depth in Chapter 7.

TinyOS 1.x uses only the second approach, passing a parameter by reference, because it does not have generic modules. For example, the Mat virtual machine supports scripting languages with typed variables, and provides functionality for checking and setting types. In this case, the ADT is a script variable. In the interface MateTypes below, a MateContext* is a thread and a MateStackVariable* is a variable:

```
interface MateTypes {
  command bool checkTypes(MateContext* context, MateStackVariable* var, uint8_t type);
  command bool checkMatch(MateContext* context, MateStackVariable* v1, MateStackVariable* v2);
  command bool checkValue(MateContext* context, MateStackVariable* var);
  command bool checkInteger(MateContext* context, MateStackVariable* var);
  command bool isInteger(MateContext* context, MateStackVariable* var);
  command bool isValue(MateContext* context, MateStackVariable* var);
  command bool isType(MateContext* context, MateStackVariable* var, uint8_t type);
}
```

Listing 4.37: Representing an ADT though an interface in TinyOS 1.x

When a component implements an ADT in this way, callers have to be careful to not corrupt the the data type. Between when a call is made to the ADT's component and when that call returns, a component should not modify the variable (i.e., call the ADT component again). In the MateTypes example above, this is easy, as all of its commands are synchronous: no code that can preempt the call (async) can itself call MateTypes.

ADTs represent the simple case of when pointers are used: they are inevitably single-phase calls. You don't for example, expect a MateTypes.isType() command to have a MateTypes.isTypeDone() event. The much trickier situation for pointers is when they involve a split-phase call. Because the called component probably needs access to the pointer while the operation is executing, it has to store it in a local variable. For example, consider the basic Send interface:

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
```

```
  event void sendDone(message_t* msg, error_t error);

  command error_t cancel(message_t* msg);
  command void* getPayload(message_t* msg);
  command uint8_t maxPayloadLength(message_t* msg);
}
```

Listing 4.38: The Send interface

The important pair of functions in this example is send/sendDone. To send a packet, a component calls send. If send returns SUCCESS, then the caller has passed the packet to a communication stack to use, and must not modify the packet. The callee stores the pointer in a variable, enacts a state change, and returns immediately. If the interface user modifies the packet after passing it to the interface provider, the packet could be corrupted. For example, the radio stack might compute a checksum over the entire packet, then start sending it out. If the caller modifies the packet after the checksum has been calculated, then the data and checksum won't match up and a receiver will reject the packet. When a split-phase interface has this kind of "pass" semantics, the completion event should have the passed pointer as one of its parameters.

Programming Hint 5: Only one component should be able to modify a pointer's data at any time. In the best case, only one component should be storing the pointer at any time.

One of the trickiest examples of this pass approach is the Receive interface. At first glance, the interface seems very simple:

```
interface Receive {
  event message_t* receive(message_t* msg, void* payload, uint8_t len);
  command uint8_t getPayloadLength(message_t* msg);
  command void * getPayload(message_t* msg, uint8_t* len);
}
```

Listing 4.39: The Receive interface

The receive event is rather different than most events: it has a message_t* as both a parameter and a return value. When the communication layer receives a packet, it passes that packet to the higher layer as a parameter. However, it also expects the higher layer to return it a message_t* back. The basic idea behind this is simple: if the communication layer doesn't have a message_t*, it can't receive packets, as it has nowhere to put them. Therefore, the higher layer always has to return a message_t*, which is the next buffer the radio stack will use to receive into. This return value can be the same as the parameter, but it does not have to be. For example, this is perfectly reasonable, if a bit feature-free, code:

```
event message_t* Receive.receive(message_t* msg, void* payload, uint8_t len) {
  return msg;
}
```

Listing 4.40: The simplest receive handler

A receive handler can always copy needed data out of the packet and just returned the passed buffer. There are, however, situations when this is undesirable. One common example is a routing queue. If the node has to forward the packet it just received, then copying it into another buffer is wasteful. Instead, a queue allocates a bunch of packets, and in addition to a send queue, keeps a free list. When the routing layer receives a packet to forward, it sees if there are any packets left in the free list. If so, it puts the received packet into the send queue and returns a packet from the free list, giving the radio stack a buffer to receive the next packet into. If there are no packets left in the free list, then the queue can't accept the packet and so just returns it back to the radio for re-use. The pseudocode looks something like this:

receive (m): if I'm not the next hop, return m if my free list is empty, return m else put m on forwarding queue, return entry from free list

One of the most common mistakes early TinyOS programmers encounter is misusing the Receive interface. For example, imagine a protocol that does this:

```
event message_t* LowerReceive.receive(message_t* m, void* payload, uint8_t len) {
  processPacket(m);
  if (amDestimation(m)) {
    signal UpperReceive.receive(m, payload, len);
  }
  return m;
}
```

Listing 4.41: A broken receive handler that doesn't respect buffer swapping

The problem with this code is that it ignores the return value from the signal to UpperReceive.receive. If the component that handles this event performs a buffer swap — e.g., it has a forwarding queue — then the packet it returns is lost. Furthermore, the packet that it has put on the queue has also been returned to the radio for the next packet reception. This means that, when the packet reaches the end of the queue, the node may send something completely different than what it decided to forward (e.g., a packet for a completely different protocol).

The buffer swap approach of the Receive interface provides isolation between different communication components. Imagine, for example, a more traditional approach, where the radio dynamically allocates a packet buffer when it needs one. It allocates buffers and passes them to components on packet reception. What happens if a component holds on to its buffers for a very long time? Ultimately, the radio stack will

run out of memory to allocate from, and will cease being able to receive packets at all. By pushing the allocation policy up into the communication components, protocols that have no free memory left are forced to drop packets, while other protocols continue unaffected.

This approach speaks more generally of how TinyOS components generally handle memory allocation. All state is allocated in one of two places: components, or the stack. A shared dynamic memory pool across components makes it much easier for one bad component to cause others to fail. That is not to say that dynamic allocation is never used. For example, the TinyDB system and the motlle language of the Mat virtual machine both maintain a dynamic memory pool. Both of them have a component that allocates a block of memory and provides an interface to allocate and free chunks within that block. However, both allocators are shared by a relatively small set of components that are designed to work together: this is a much more limited, and safer, approach than having routing layers, signal processing modules, and applications all share a memory pool.

Programming Hint 6: Allocate all state in components. If your application requirements necessitate a dynamic memory pool, encapsulate it in a component and try to limit the set of users.

Modules often need constants of one kind or another, such as a retransmit count or a threshold. Using a literal constant is problematic, as you'd like to be able to reuse a consistent value. This means that in C-like languages, you generally use something like this:

```
const int MAX_RETRANSMIT = 5;

if (txCount < MAX_RETRANSMIT) {
 ...
}
```

Listing 4.42: Wasting memory by defining a constant as an integer

The problem with doing this in nesC/TinyOS is that a const int might allocate RAM, depending on the compiler (good compilers will place it in program memory). You can get the exact same effect by defining an enum:

```
enum {
  MAX_RETRANSMIT = 5
};
```

Listing 4.43: Defining a constant as an enum

This allows the component to use a name to maintain a consistent value and does not store the value

either in RAM or program memory. This can even improve performance, as rather than a memory load, the architecture can just load a constant. It's also better than a #define, as it exists in the debugging symbol table and application metadata.

Note, however, that using enum types in variable declarations can waste memory, as enums default to integer width. For example, imagine this enum:

```
typedef enum {
  STATE_OFF = 0,
  STATE_STARTING = 1,
  STATE_ON = 2,
  STATE_STOPPING = 3
} state_t;
```

Listing 4.44: An example enum

Here are two different ways you might allocate the state variable in question:

```
state_t state; // platform int size (e.g., 2-4 bytes)
uint8_t state; // one byte
```

Listing 4.45: Allocating a state variable

Even though the valid range of values is 0-3, the former will allocate a native integer, which on a microcontroller is usually 2 bytes, but could be 4 bytes on low power microprocessors. The second will allocate a single byte. So you should use enums to declare constants, but avoid declaring variables of an enum type.

Programming Hint 7: Conserve memory by using enums rather than const variables for integer constants, and and don't declare variables with an enum type.

Chapter 5

Configurations and Wiring

The previous two chapters dealt with modules, which are the basic building blocks of a TinyOS program. Modules allocate state and implement executable logic. However, like all components, they can only name functions and variables within their local namespaces, as defined by their signatures. For one module to be able to call another, we have to map a set of names in one component — generally, an interface — to a set of names in another component. In nesC, connecting two components in this way is called wiring. In addition to modules, nesC has a second kind of component, configurations, whose implementation is component wirings. Modules implement program logic: configurations compose modules into larger abstractions.

In a TinyOS program, there are usually more configurations than modules. There are two reasons for this. First, except low-level hardware abstractions, any given component is built on top of a set of other abstractions, which are encapsulated in configurations. For example, a routing stack depends on a single-hop packet layer, which is a configuration. This single-hop configuration wires the actual protocol implementation module (e.g., setting header fields) to a raw packet layer on top of the radio. This raw packet layer is a configuration that wires the module which sends bytes out to the bus over which it sends bytes. The bus, in turn, is a configuration. These layers of encapsulation generally reach very low in the system.

Essentially, encapsulating an abstraction A in a configuration means that it can be ready-to-use: all we need to do is wire to A's functionality. In contrast, if it were a module that uses interfaces, then we'd need to wire up A's dependencies and requirements as well. That's sort of like having to link the Java libraries against the C libraries every time you want to compile a Java program. For example, a radio stack can use a really wide range of resources, including buses, timers, random number generators, cryptographic support, and hardware pins. Rather than expecting a programmer to connect the stack up to all of these things, the entire stack can be encapsulated in a single component. This component connects all of the subcomponents

to the abstractions they need.

In addition to wiring one component to another, configurations also need to export interfaces. This is another kind of wiring, except that, rather than connect two end points — a provider and a user — an export (also called pass-through wiring) maps one name to another. This idea is confusing at first, and is best explained after a few examples, so we'll return to it later.

Configurations look very similar to modules. They have a specification and an implementation. This is the configuration LedsC, which presents the TinyOS abstraction of the ubiquitious 3 LEDs[1]:

```
configuration LedsC {
  provides interface Init @atleastonce();
  provides interface Leds;
}
implementation {
  components LedsP, PlatformLedsC;
  Init = LedsP;
  Leds = LedsP;

  LedsP.Led0 -> PlatformLedsC.Led0;
  LedsP.Led1 -> PlatformLedsC.Led1;
  LedsP.Led2 -> PlatformLedsC.Led2;
}
```

Listing 5.1: The LedsC configuration

Syntactically, configurations are very simple. They have three operators: -¿, ¡- and =. The first two are for basic wiring: the arrow points from the user to the provider. For example, the following two lines are identical:

```
MyComponent.Random -> RandomC.Random;
RandomC.Random <- MyComponent.Random;
```

Listing 5.2: Wiring directionality

A direct wiring (a -¿ or ¡-) always goes from a user to a provider, and resolves the call paths in both directions. That is, once an interface is linked with a -¿ operator, it is considered connected. Here's a simple example, the Blink application:

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
```

[1]Don't worry about the @atleastonce attribute; attributes are discussed in Chapterch:advanced

```
  uses interface Boot;
}
```

Listing 5.3: The signature of the BlinkC module

When BlinkC calls Leds.led0Toggle(), it names a function in its own local scope (BlinkC.Leds.led0Toggle). The LedsC component provides the Leds interface[2]:

```
configuration LedsC {
  provides interface Init @atleastonce();
  provides interface Leds;
}
```

Listing 5.4: The signature of the LedsC configuration

BlinkC calls the function BlinkC.Leds.led0Toggle. LedsC provides the function LedsC.Leds.led0Toggle(). Wiring the two maps the first to the second:

```
configuration BlinkAppC {}
implementation {
  components MainC, BlinkC, LedsC;
  // Some code elided
  BlinkC.Leds -> LedsC;
}
```

Listing 5.5: The BlinkAppC configuration that wires BlinkC to LedsC

This means that when BlinkC calls BlinkC.Leds.led0Toggle, it actually calls LedsC.Leds.led0Toggle. The same is true for other calls of the Leds interface, such as Leds.led1On. The configuration BlinkAppC provides a mapping between the local namespaces of the two components. The -¿ operator maps between two components that a configuration names, and is always from a user to a provider.

From the perspective of someone using a component, it shouldn't be relevant whether it is a module or a configuration. Just like modules, configurations can provide and use interfaces. But as they have no code, these interfaces must be defined in terms of other components. Take, for example, ActiveMessageC, the HIL for packet-level communication. Every hardware platform defines a component ActiveMessageC, which the basic packet components (AMSenderC, AMReceiverC, etc.) wire to. Generally, ActiveMessageC is just a configuration that renames a particular radio chip's active message layer. For example, this is the ActiveMessageC of the Telos platform[3]:

[2]Don't worry about the @atleastonce attribute on Init; we discuss it in Chapter 9. It's basically a way to make sure somebody initializes the Leds.

[3]Don't worry about the array brackets on AMSend, Receive, and Snoop: they represent what's called a *parameterized interface*, which is covered in a later chapter.

```
configuration ActiveMessageC {
  provides {
    interface Init;
    interface SplitControl;

    interface AMSend[uint8_t id]
    interface Receive[uint8_t id];
    interface Receive as Snoop[uint8_t id];

    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements;
  }
}
implementation {
  components CC2420ActiveMessageC as AM;

  Init         = AM;
  SplitControl = AM;

  AMSend       = AM;
  Receive      = AM.Receive;
  Snoop        = AM.Snoop;
  Packet       = AM;
  AMPacket     = AM;
  PacketAcknowledgements = AM;
}
```

Listing 5.6: The ActiveMessageC Configuration

All ActiveMessageC does is take CC2420ActiveMessageC and present all of its interfaces with a different name. Another option could have been for the CC2420 (a radio chip) code to define an ActiveMessageC. From an OS standpoint, the problem with this approach is dealing with the situation when a platform has two radio chips: they both define an ActiveMessageC, and since this is a global name, you need some way to determine which one.

ActiveMessageC uses the other configuration operator, =, which exports interfaces. While the -¿ operator maps between the interfaces of components that a configuration names, the = operator maps between a configuration's own interfaces and components that it names, exporting interfaces of components within the configuration out to the configuration's namespace. Take, for example, RandomC, the component that defines the standard TinyOS random number generator:

```
configuration RandomC {
  provides interface Init;
  provides interface ParameterInit<uint16_t> as SeedInit;
  provides interface Random as Random;
}
```

```
implementation {
  components RandomMlcgC;
  components MainC;

  Init = RandomMlcgC;   // Allow for re-initialization
  MainC.SoftwareInit -> RandomMlcgC; // Auto-initialize

  SeedInit = RandomMlcgC;
  Random = RandomMlcgC;
}
```

Listing 5.7: The RandomC configuration

In the default case, RandomC is a wrapper around RandomMlcgC. There's another implementation, RandomLfsrC, that is about twice as fast but produces not nearly as good random numbers. Platforms or applications that need to use RandomLfsrC can redefine RandomC to encapsulate RandomLfsrC instead.

RandomMlcgC provides the Init interface. Calling RandomMlcgC.Init.init seeds the random number generator with the node's local address. The SeedInit interface allows a component to start the generator with a specific seed. RandomC wires RandomMlcgC.Init in two different ways:

```
  Init = RandomMlcgC;   // Allow for re-initialization
  MainC.SoftwareInit -> RandomMlcgC; // Auto-initialize
```

Listing 5.8: RandomC's wiring of Init for auto–initialization and re–initialization

In the first, it equates its own RandomC.Init with RandomMlcg.Init. If a component calls RandomC.Init.init(), it actually calls RandomMlcgC.Init.init(). In the second, RandomC wires RandomMlcgC.Init to the TinyOS boot sequence (MainC). When TinyOS boots, it calls MainC.SoftwareInit.init (see TEP 107 for the full boot sequence), and so it calls RandomMlcgC.Init.init(). This means that, before an application starts, RandomC has made sure that its underlying random number generator has been properly seeded. If the application re-seeds it by calling RandomC.Init or RandomC.SeedInit, no harm is done. But by wiring to MainC, RandomC makes sure that an application (or protocol, or system) doesn't have to remember to initialize RandomC.

This technique — "auto-wiring" initialization — is used in many TinyOS 2.x abstractions. One very common bug in TinyOS 1.x is to forget to initialize. This usually happens because *many* components might be initializing the same component. This approach is wasteful, but since initialization only happens once, it's not a huge issue. The bigger issue is that a component often relies on someone else initializing. For example, imagine two radio stacks, A and B. A initializes the timer system, B does not. A programmer writes an application using radio stack A and forgets to initialize the timer system. Because radio stack A does, everything works fine. The programmer then decides to switch to radio stack B, and nothing

works: neither the application nor the stack initialize the timers, and so the system just hangs. For software initialization setting fields, etc. – generally doesn't matter (Init is not supposed to call anything besides Init). Hardware initialization is a much trickier problem, and is generally handled on a per-platform basis. Refer to TEP 107 for more details.

Programming Hint 8: In the top-level configuration of a software abstraction, auto-wire Init to MainC. This removes the burden of wiring Init from the programmer, which removes unnecessary work from the boot sequence and removes the possibility of bugs from forgetting to wire.

From an implementation standpoint, the two configuration operators have two very different purposes. The = operator defines how a configuration's interfaces are implemented. Like a module, a configuration is an abstractions defined by a signature. A module directly implements the functions it needs to (events from its used interfaces, commands from its provided interfaces). A configuration, in contrast, delegates the implementation to another component using the = operator. For example, RandomC delegates the implementation of the Random interface to RandomMlcgC. In contrast, the -¿ operator combines existing components, completing existing signatures.

TinyOS component names all end in either C or P. C stands for Component and means that it represents a usable abstraction. P stands for Private, and generally means that you shouldn't wire to it: instead, there's usually a C that encapsulates it in some way to make it useful. Once you have written the signature for a C component, changing it is very hard: any number of other components might depend on it, and changing it will cause compilation errors. In contrast, because a P component is only wired to by higher-level configurations within that software abstraction, their signatures are much more flexible. E.g., changing the signature of AMSenderC would break almost all TinyOS code, but an internal change to CC2420ReceiveP (and changing its wiring in CC2420ReceiveC) should not be apparent to the user.

The distinction between C (an externally usable abstraction) and P (an internal implementation) is particularly important in nesC because of the component model. In languages such as C, an implementation can directly reference what it depends on (e.g., library calls). In nesC, a configuration needs to resolve those dependencies. Let's look at a complete (but very simple) example of how all of these issues can be resolved: RandomC.

Programming Hint 9: If a component is a usable abstraction by itself, its name should end with C. If it is intended to be an internal and private part of a larger abstraction, its name should end with P. Never wire to P components from outside your package (directory).

As mentioned above, RandomC is name for the standard TinyOS random number generator. It is a

configuration with this signature:

```
configuration RandomC {
  provides interface Init;
  provides interface ParameterInit<uint16_t> as SeedInit;
  provides interface Random as Random;
}
```

Listing 5.9: The RandomC signature

The default implementation of RandomC lives in tos/system. As shown above, maps RandomC to a specific implementation, RandomMlcgC, while auto-wiring to the boot sequence. RandomMlcgC is itself a (trivial) configuration:

```
configuration RandomMlcgC {
  provides interface Init;
  provides interface ParameterInit<uint16_t> as SeedInit;
  provides interface Random as Random;
}
implementation {
  components RandomMlcgP;

  Init = RandomMlcgP;
  SeedInit = RandomMlcgP;
  Random = RandomMlcgP;
}
```

Listing 5.10: The RandomMlcgC signature

RandomMlcgC represents a complete random number generator abstraction that is multiplicative linear congruential generator (MLCG). RandomMlcgP is a particular implementation of such a generator. In this case, it's completely in software. A platform that has a hardware random number generator could have a different RandomMlcgP. Because this different implementation might have a different signature — e.g., it might require accessing registers through an HPL — it also requires a different RandomMlcgC that resolves these dependencies to present a complete abstraction.

In short, the configuration RandomC maps the standard number generator to a specific algorithm, RandomMlcgC. The configuration RandomMlcgC encapsulates a specific implementation as a complete abstraction. RandomMlcgP is an implementation of the multiplicative linear congruential generator. Similarly, there is also a RandomLfsrC, which is a linear feed shift register random number generator. RandomLfsrC is a configuration that just exports the interfaces of RandomLfsrP, the software implementation. This hierarchy of names means that a system can wire to a specific random number generator if it cares which one it uses, or wire to the general one that TinyOS provides (RandomC). An application can change what the

default random number generator is by defining its own RandomC, which maps to a different algorithm.

The as keyword and other namespace tricks

Components sometimes name two instances of an interface in their signature:

```
// A greatly elided signature of ActiveMessageC
configuration ActiveMessageC {
  provides interface Receive[am_id_t];
  provides interface Receive as Snoop[am_id_t];
}
```

Listing 5.11: A truncated ActiveMessageC configuration

The as keyword allows you to rename an interface in a signature. The Snoop interface above, for example, is still of type Receive: you can wire any "uses interface Receive" to it. However, its name allows you to distinguish between Snoop (packets not destined for the local node) and Receive (packets destined for you). Technically, the statement

```
uses interface StdControl;
```

Listing 5.12: Using StdControl

is actually

```
uses interface StdControl as StdControl;
```

Listing 5.13: The implicit as keyword in uses and provides

That is, the first StdControl is the type, and the second is the name. Because this is so common, nesC allows you to use the shorthand.

The as keyword can also be used within configurations. Because nesC components are in a global namespace, sometimes they have very long and descriptive names. For example, the lowest level (byte) SPI bus abstraction on the Atmega128 is HplAtm128SpiP, which means, "This is the private hardware presentation layer component of the Atmega128 SPI bus." Typing that in a configuration is a real pain, and it's not very easy to read. So, the slightly higher level abstraction, the configuration Atm128SpiC, names it like this:

```
HplAtm128SpiC as HplSpi;
```

Listing 5.14: Using the as keyword with components

which makes the wiring a good deal more comprehensible. Similarly, CC2420ReceiveC, the receive path of the CC2420 radio, is a configuration that wires packet logic to things like interrupts and status pins:

```
configuration CC2420ReceiveC {...}
implementation {
  components CC2420ReceiveP;
  components new CC2420SpiC() as Spi;

  components HplCC2420PinsC as Pins;
  components HplCC2420InterruptsC as InterruptsC;
  // rest of the implementation elided
}
```

Listing 5.15: CC2420ReceiveC's use of the as keyword

Because all interfaces are types, when wiring you can sometimes elide one of the interface names. You've actually seen this a lot in the previous examples, such as RandomC:

```
MainC.SoftwareInit -> RandomMlcgC; // Auto-initialize
```

Listing 5.16: Autoinitializing RandomMlcgC

On the left side, MainC.SoftwareInit is an instance of the Init interface. On the right side is RandomMlcgC, without an interface name. Because RandomMlcgC only provides one instance of the Init interface, nesC assumes that this is the one you mean. So technically, this line is

```
MainC.SoftwareInit -> RandomMlcgC.Init;
```

Listing 5.17: Expanding implicit interface wiring

If it's an export wiring, then the component name is implicit on one side, so you always have to name the interface. For example,

```
Init = RandomMlcgC;   // Allow for re-initialization
```

Listing 5.18: Enabling re–initialization of RandomMlcgC

means "wire Init of this component to the interface of type Init of RandomMlcgC." This form of shorthand works in terms of types, not names. It would work just as well if RandomMlcgC provided Init as "RandomInit". However, you can't do this:

```
= RandomMlcgC.Init;
```

Listing 5.19: An illegal implicit wiring

If a component has two instances of the same interface, then you have to name which instance you mean. For example, this is ActiveMessageC for the telos platforms:

```
configuration ActiveMessageC {
  provides {
    interface Init;
    interface Receive[uint8_t id];
    interface Receive as Snoop[uint8_t id];
  }
}
implementation {
  components CC2420ActiveMessageC as AM;
  Init         = AM;
  ...
  Receive      = AM.Receive;
  Snoop        = AM.Snoop;
  ...
}
```

Listing 5.20: Using the as keyword to distinguish interface instances

Because CC2420ActiveMessageC provides two instances of the Receive interface, ActiveMessageC has to name them. Basically, wiring has to be precise and unambiguous, but if nesC lets you use shorthand in the common cases of redundancy.

The as keyword make code more readable and comprehensible. Because there is a flat component namespace, some components have long and complex names which can be easily summarized, and using the as keyword with interfaces can add greater semantic information on the role of that interface. Additionally, by using the as keyword, you create a level of indirection. E.g., if a configuration uses the as keyword to rename a component, then changing the component only requires changing that one line. Without the keyword, you have to change every place it's named in the configuration.

Programming Hint 10: Use the "as" keyword liberally.

5.1 Pass Through Wiring

Sometimes you don't want a configuration to specify the endpoint of an interface. Instead, you need a configuration to act as a renaming mechanism, or as a thin shim which interposes on some (but not all) of the interfaces of a given abstraction. You don't want the component using the shim component to know which are interposed. This practice is very rare in TinyOS 2.0 (there isn't a single instance of it in the core),

but it was used some times in 1.x, and so it's here for completeness sake.

Pass through wiring is when a configuration wires two interfaces in its signature together. It must wire a uses to a provides, and it does so with the = operator. For example, this is a configuration that does nothing except introduce a name change on the interface:

```
configuration NameChangeC {
  provides interface Send as SpecialSend;
  uses interface Send as SubSend;
}
implementation {
  SpecialSend = SubSend;
}
```

Listing 5.21: Using a pass–through wiring to rename an abstraction

A component that wires to NameChangeC.SpecialSend wires to whatever NameChangeC.SubSend has been wired to.

Multiple Wirings, Fan-in, and Fan-out

Not all wirings are one-to-one. For example, this is part of the component CC2420TransmitC, a configuration that encapsulates the transmit path of the CC2420 radio (there's also a CC2420ReceiveC):

```
configuration CC2420TransmitC {
  provides interface Init;
  provides interface AsyncControl;
  provides interface CC2420Transmit;
  provides interface CSMABackoff;
  provides interface RadioTimeStamping;
}
implementation {
  components CC2420TransmitP;
  components AlarmMultiplexC as Alarm;
  Init = Alarm;
  Init = CC2420TransmitP;
  // further wirings elided
}
```

Listing 5.22: Fan–out on CC2420TransmitC's Init

This wiring means that CC2420TransmitC.Init maps both to Alarm.Init and CC2420TransmitP.Init. What does that mean? There certainly isn't any analogue in C-like languages. In nesC, a multiple-wiring like this means that when a component calls CC2420TransmitC.Init.init(), it calls both Alarm.Init.init() and CC2420TransmitP.Init.init(). The order of the two calls is not defined.

This ability to multiply wire might seem strange. In this case, you have a single call point, CC2420TransmitC.Init.init, which fans-out to two callees. There are also fan-ins, which are really just a fancy name for "multiple people

call the same function." But the similarity of the names "fan-in" and "fan-out" is important, as nesC interfaces are bidirectional. For example, coming from C, wiring two components to RandomC.Random doesn't seem strange: two different components might need to generate random numbers. In this case, as Random only has commands, all of the functions are fan-in, as there are multiple callers for a single callee, just like a library function.

But as nesC interfaces are bidirectional. if there is fan-in on the command of an interface, then when that component signals an event on the interface, there are multiple callees. Take, for example, the power control interfaces, StdControl and SplitControl. StdControl is single-phase: it only has commands. SplitControl, as its name suggests, is split-phase: the commands have completion events. In this wiring,

```
components A, B, C;
A.StdControl -> C;
B.StdControl -> C;
```

Listing 5.23: Fan–in on StdControl

Then either A or B can call StdControl to start or stop C. However, in this wiring, there are also completion events:

```
components A, B, C;
A.SplitControl -> C;
B.SplitControl -> C;
```

Listing 5.24: Fan–out and fan–in on SplitControl

Either A or B can call SplitControl.start. When C issues the SplitControl.startDone() event, though, both of them are wired to it, so both A.SplitControl.startDone and B.SplitControl.startDone are called. The implementation has no way of determining which called the start command.[4]

Interfaces are not a one-to-one relationship. Instead, they are an n-to-k relationship, where n is the number of users and k is the number of providers. Any provider signaling will invoke the event handler on all n users, and any user calling a command will invoke the command on all k providers.

Anecdote: Historically, multiple wirings come from the idea that TinyOS components can be thought of as hardware chips. In this model, an interface is a set of pins on the chip. The term wiring comes from this idea: connecting the pins on one chip to those of another. In hardware, though, you can easily connect N pins together. For example, a given GPIO pin on a chip might have multiple possible triggers, or

[4]There are ways to disambiguate this, through parameterized interfaces, which are covered in the next chapter.

a bus have have multiple end devices that are controlled with chip select pins. It turns out that taking this metaphor literally has several issues. When TinyOS moved to nesC, these problems were done away with. Specifically, consider this configuration:

```
configuration A {
  uses interface StdControl;
}

configuration B {
  provides interface StdControl;
  uses interface StdControl as SubControl; // Called in StdControl
}

configuration C {
  provides interface StdControl;
}

A -> B.StdControl;
A -> C.StdControl;
B.SubControl -> C;
```

Listing 5.25: Why the metaphor of "wires" is only a metaphor

If you take the multiple wiring metaphor literally, then the wiring of B to C joins it with the wiring of A to B and C. That is, they all form a single "wire." The problem is that B's call to C is the same wire as A's call to B. B enters an infinite recursion loop, as it calls SubControl, which calls StdControl, which calls SubControl, and so on and so on. Therefore, nesC does not take the metaphor literally. Instead, the wirings from one interface to another are considered separately. So the code

```
A -> B.StdControl;
A -> C.StdControl;
B.SubControl -> C;
```

Listing 5.26: How nesC handles multiple wirings

Makes it so that when A calls StdControl.start it calls B and C, and when B calls SubControl.start it calls C.

In practice, multiple wirings allow an implementation to be independent of the number of components it depends on. Consider, for example, MainC, which presents the abstraction of the boot sequence to software systems:

```
configuration MainC {
  provides interface Boot;
```

```
  uses interface Init as SoftwareInit;
}
```

Listing 5.27: MainC

It only has two interfaces. The first, SoftwareInit, it calls when booting so that software components which need so can be sure they're initialized before execution begins. The second, Boot, signals an event once the entire boot sequence is over. Many components need initialization. For example, in the very simple application RadioCountToLeds, there are ten components wired to MainC.SoftwareInit. Rather than use many Init interfaces and call them in some order, MainC just calls SoftwareInit once and this call forks out to all of the components that have wired to it.[5]

5.2 Combine Functions

Fan-out raises an interesting question: if

```
call SoftwareInit.init()
```

Listing 5.28: Calling SoftwareInit.init()

actually calls ten different functions, then what is its return value?

nesC provides the mechanism of combine functions to specify the return value. A data type can have an associated combine function. Because a fan-out always involves calling N functions with identical signatures, the caller's return value is the result of applying the combine function to the return values of all of the callees. When nesC compiles the application, it autogenerates a fan-out function which applies the combine function.

For example, error_t's combine function is ecombine (defined in types/TinyError.h):

```
error_t ecombine(error_t e1, error_t e2) {
  return (e1 == e2)? e1: FAIL;
}
```

Listing 5.29: The combine function for error_t

If both calls return the same value, ecombine returns that value. Otherwise, as only one of them could be SUCCESS, it returns FAIL. This combine function is bound to error_t with a C attribute:

[5] Another approach could have been to use a parameterized interface (covered in the next chapter), but as the calls to Init are supposed to be very self-contained, the idea is that the increased complexity wouldn't be worth it.

```
typedef uint8_t error_t __attribute__((combine(ecombine)));
```

Listing 5.30: Associating a combine function with a type

When asked to compile the following configuration

```
configuration InitExample {}
implementation {
  components MainC;
  components AppA, AppB;

  MainC.SoftwareInit -> AppA;
  MainC.SoftwareInit -> AppB;
}
```

Listing 5.31: Fan–out on SoftwareInit

ncc will generate something like the following code[6]:

```
error_t MainC\$SoftwareInit\$init() {
  error_t result;
  result = AppA\$SoftwareInit\$init();
  result = ecombine(result, AppB\$SoftwareInit\$init());
  return result;
}
```

Listing 5.32: Resulting code from fan–out on SoftwareInit

Some return values don't have combine functions, either due to programmer oversight or the semantics of the data type. Examples of the latter include things like data pointers: if both calls return a pointer, say, to a packet, there isn't a clear way to combine them into a single pointer. If your program has fan-out on a call whose return value can't be combined, the nesC compiler will issue a warning along the lines of

"calls to Receive.receive in CC2420ActiveMessageP are uncombined"

or

"calls to Receive.receive in CC2420ActiveMessageP fan out, but there is no combine function fpecified for the return value."

Programming Hint 11: Never ignore combine warnings.

[6]The nesC compiler actually compiles to C, which it then passes to a native C compiler. Generally, it uses $ as the delimiter between component, interface, and function names. Because nesC does not allow $, this allows the compiler to enforce component encapsulation (there's no way to call a function with a $ from within nesC and break the component boundaries).

Chapter 6

Parameterized Wiring

Sometimes, a component wants to provide many instances of an interface. For example, the basic timer implementation component HilTimerMilliC[1] doesn't provide just one timer: it needs to provide many timers. One way it could do so is by having a long signature:

```
configuration HilTimerMilliC {
  provides interface Timer<TMilli> as Timer0;
  provides interface Timer<TMilli> as Timer1;
  provides interface Timer<TMilli> as Timer2;
  provides interface Timer<TMilli> as Timer3;
  ...
  provides interface Timer<TMilli> as Timer100;
}
```

Listing 6.1: Timers without parameterized interfaces

While this works, it's kind of a pain and leads to a lot of repeated code. Every instance needs to have its own implementation. That is, there will be 100 different startPeriodic functions, even though they're almost completely identical. Another approach could be to have a call parameter to the Timer interface that specifies which timer is being changed, sort of like a file descriptor in POSIX file system calls. In this case, HilTimerMilliC would look like this

```
configuration HilTimerMilliC {
  provides interface Timer;
}
```

Listing 6.2: Timers with a single interface

[1] See TEP 102 for details.

Components that use timers would have some way of generating unique timer identifiers, and would pass them in every call:

```
call Timer.startPeriodic(timerDescriptor, 1024); // Fire at 1Hz
```

Listing 6.3: Starting a timer with a run–time parameter

While this approach works it doesn't lead to multiple implementations passing the parameter is generally unnecessary, in that components generally allocate some number of timers and then only use those timers. That is, the set of timers a component uses and the size of the set are generally known at compile time. Making the caller pass the parameter at runtime is therefore unnecessary, and could possibly introduce bugs (e.g., if it were, due to laziness, stored in a variable).

There are other situations when a component wants to provide a large number of interfaces, such as communication. Active messages have an 8-bit type field, which is essentially a protocol identifier. In the Internet, the valid protocol identifiers for IP are well specified[2], and many port numbers for TCP are well established. When a node receives an IP packet with protocol identifier 6, it knows that this is a TCP packet and dispatches it to the TCP stack. Active messages need to perform a similar function, albeit without the standardization of IANA: a network protocol needs to be able to register to send and receive certain AM types. Like timers, with basic interfaces there are two ways to approach this: code redundancy or run-time parameters. That is, you could either have a configuration like this

```
configuration NetworkProtocolC {...}
implementation {
  components NetworkProtocolP, PacketLayerC;
  NetworkProtocolP.Send -> PacketLayerC.Send15;
}
```

Listing 6.4: Wiring to AM type 15 by name

or the network protocol code could look like this:

```
call Send.send(15, msg, sizeof(payload_t));
```

Listing 6.5: Calling AM type 15 with a compile–time parameter

Neither of these solutions is very appealing. The first leads to a lot of redundant code, wasting code memory. Also, as the wiring is by name, it is also difficult to wire to. That is, there is no way to manipulate constants in order to control the wiring. For example, if a sensor filter and a routing stack both wire to

[2]http://www.iana.org/assignments/protocol-numbers

Timer3, there's no way to separate them without changing the code text of one of them to read "Timer4." One way to manage the namespace would be to have components leave their timers unwired and then expect the application to resolve all of them. But this places a large burden on an application developer. For example, a small application that builds on top of a lot of large libraries might have to wire eight different timers. Additionally, it means that the components it includes aren't self-contained, working abstractions: they have remaining dependencies that an application developer needs to resolve.

The second approach is superior to the first at first glance, but it turns out to have even more significant problems. First, in many cases the identifier is a compile-time constant. Requiring the caller to pass it as a run-time parameter is unnecessary and is a possible source of bugs. Second, and more importantly, it pushes identifier management into the caller. For example, let's return to the timer example:

```
call Timer.startPeriodic(timerDescriptor, 1024); // Fire at 1Hz
```

Listing 6.6: Starting a timer with a run–time parameter

From the calling component's perspective, it doesn't care which timer it's using. All it cares is that it has its own timer. By making the identifier part of the call, this forces the module to know (and manage) the name of the identifier. The third and largest problem, however, isn't with calls out to other components: it's with calls in from other components. In Timer, for example, how does the timer service signal a fired() event? Because the identifier is a runtime parameter, the only way is for Timer.fired() fan-out to all timers, and have them all check the identifier.

To support abstractions that have sets of interfaces, nesC has parameterized interfaces. You've seen them in a few of the earlier example signatures. A parameterized interface is essentially an array of interfaces, and the array index is the parameter. For example, this is the signature of ActiveMessageC:

```
configuration ActiveMessageC {
  provides {
    interface Init;
    interface SplitControl;

    interface AMSend[uint8_t id];
    interface Receive[uint8_t id];
    interface Receive as Snoop[uint8_t id];

    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements;
  }
}
```

Listing 6.7: ActiveMessageC signature

AMSend, Receive, and Snoop are all parameterized interfaces. Their parameter is the AM type of the message (the protocol identifier). Normally, components don't wire directly to ActiveMessageC. Instead, they use AMSenderC, AMReceiverC, and the other virtualized abstractions. [3] However, there are some test applications for the basic AM abstraction, such as TestAM. The module TestAMC sends and receives packets:

```
module TestAMC {
  uses {
    ...
    interface Receive;
    interface AMSend;
    ...
  }
}
```

Listing 6.8: Signature of TestAMC

TestAMAppC is the configuration that wires up the TestAMC module:

```
configuration TestAMAppC {}
implementation {
  components MainC, TestAMC as App;
  components ActiveMessageC;

  MainC.SoftwareInit -> ActiveMessageC;
  App.Receive -> ActiveMessageC.Receive[240];
  App.AMSend -> ActiveMessageC.AMSend[240];
  ...
}
```

Listing 6.9: Wiring TestAMC to ActiveMessageC

Note that TestAM has to wire SoftwareInit to ActiveMessageC because it doesn't use the standard abstractions, which auto-wire it. This configuration means that when TestAMC calls AMSend.send, it calls ActiveMessageC.AMSend number 240, so packets with protocol ID 240. Similarly, TestAMC receives packets with protocol ID 240. Because these constants are specified in the configuration, they are not bound in the module: from the module's perspective, they don't even exist. That is, from TestAMC's perspective, these two lines of code are identical:

[3] See TEP 116: Packet Protocols, for details.

```
TestAMC.AMSend -> ActiveMessageC.AMSend240; // Not real TinyOS code
TestAMC.AMSend -> ActiveMessageC.AMSend[240];
```

Listing 6.10: Wiring to a single interface versus an instance of a parameterized interface

The different lies in the component with the parameterized interface. The parameter is essentially another argument in functions of that interface. In ActiveMessageC.AMSend, for example, the parameter is an argument passed to it in calls to send() and which it must pass in signals of sendDone(). But the parameterized interface gives you two key things. First, it automatically fills in this parameter when TestAMC calls send (nesC generates a stub function to do so, and inlining makes the cost negligible). Second, it automatically dispatches on the parameter when ActiveMessageC signals sendDone (nesC generates a switch table based on the identifier).

In reality, ActiveMessageC is a configuration that encapsulates a particular chip, such as CC2420ActiveMessageC, which encapsulates that chip's implementation, such as CC2420ActiveMessageP:

```
module CC2420ActiveMessageP {
  provides {
    interface AMSend[am_id_t id];
    ...
  }
}
```

Listing 6.11: A possible module underneath ActiveMessageC

Within CC2420ActiveMessageP, this is what the parameterized interface looks like:

```
command error_t AMSend.send[am_id_t id](am_addr_t addr, message_t* msg, uint8_t len) {
  cc2420_header_t* header = getHeader( msg );
  header->type = id;
  ...
}
```

Listing 6.12: Parameterized interface syntax

The interface parameter precedes the function argument list, and the implementation can treat it like any other argument. Basically, it is a function argument that the nesC compiler fills in when components are composed. When CC2420ActiveMessageP wants to signal sendDone, it pulls the protocol ID back out of the packet and uses that as the interface parameter:

```
event void SubSend.sendDone(message_t* msg, error_t result) {
  signal AMSend.sendDone[call AMPacket.type(msg)](msg, result);
}
```

Listing 6.13: Dispatching on a parameterized interface

If the AM type of the packet is 240, then the dispatch code nesC generates will cause this line of code to signal the sendDone() wired to ActiveMessageC.AMSend[240], which in this case is TestAMC.AMSend.sendDone.

CC2420ActiveMessageP.Receive looks similar to sendDone. The AM implementation receives a packet from a lower level component and dispatches on the AM type to deliver it to the correct component. Depending on whether the packet is destined to the local node, it signals either Receive.receive or Snoop.receive:

```
event message_t* SubReceive.receive(message_t* msg, void* payload, uint8_t len) {
  if (call AMPacket.isForMe(msg)) {
    return signal Receive.receive[call AMPacket.type(msg)](msg, payload, len - CC2420_SIZE);
 }
  else {
    return signal Snoop.receive[call AMPacket.type(msg)](msg, payload, len - CC2420_SIZE);
  }
}
```

Listing 6.14: How active message implementations decide on whether to signal to Receive or Snoop

The subtraction of CC2420_SIZE is because the lower layer has reported the entire size of the packet, while to layers above AM the size of the packet is only the data payload (the entire size minus the size of headers and footers, that is, CC2420_SIZE).

Parameterized interfaces get the best of both worlds. Unlike the name-based approach (e.g. Send240) described above, there is a single implementation of the call. Additionally, since the parameter is a value, unlike a name it can be configured and set. E.g., a component can do something like this:

```
\#ifndef ROUTING_TYPE
\#define ROUTING_TYPE 201
\#endif

RouterP.AMSend -> PacketSenderC.AMSend[ROUTING_TYPE];
```

Listing 6.15: Defining a parameter

It also avoids the pitfalls of the runtime parameter approach. Because the constant is set at compile-time, nesC can automatically fill it in and dispatch based on it, simplifying the code and improving the efficiency of outgoing function invocations.

Note that you can also wire entire parameterized interfaces:

```
configuration CC2420ActiveMessageC {
  provides interface AMSend[am_id_t id];
```

```
} {...}
configuration ActiveMessageC {
  provides interface AMSend[am_id_t];
}
implementation {
  components CC2420ActiveMessageC;
  AMSend = CC2420ActiveMessageC;
}
```

Listing 6.16: Wiring full parameterized interface sets

Programming Hint 12: If a function has an argument which is one of a small number of constants, consider defining it as a few separate functions to prevent bugs. If the functions of an interface all have an argument that's almost always a constant within a large range, consider using a parameterized interface to save code space. If the functions of an interface all have an argument that's a constant within a large range but only certain valid values, implement it as a parameterized interface but expose it as individual interfaces, to both minimize code size and prevent bugs.

Parameterized interfaces aren't limited to a single parameter. For example, this is valid code:

provides interface Timer[uint8_t x][uint8_t y];

In practice, however, this leads to large and inefficienct code (nested swich statements), and so components rarely (if ever) use it.

6.1 Defaults

Because a module's call points are resolved in configurations, a common compile error in nesC is to forget to wire something. The equivalent in C is to forget to include a library in the link path, and in Java it's to include a jar. Usually, a dangling wire represents a bug in the program. With parameterized interfaces, however, often they don't.

Take, for example, the Receive interface of ActiveMessageC. Most applications receive a few AM types, maybe 15 at most: they don't respond to or use every protocol ever developed. However, there's these call in CC2420ActiveMessageP:

```
return signal Receive.receive[call AMPacket.type(msg)](msg, payload, len - CC2420_SIZE);
```

Listing 6.17: Signaling Receive.receive

On one hand, if all of the nodes in the network run the same executable, it's possible that none of them will ever send a packet of, say, AM type 144. However, if there are other nodes nearby, or if packets are

corrupted in memory before being sent (or after being received), then it's very possible that a node which doesn't care about protocol 144 will receive a packet of this type. Therefore nesC expects the receive event to have a handler: it needs to execute a function when this happens. But the application doesn't wire to Receive[144], and making a developer wire to all of the unwired instances is unreasonable, especially as they're all null functions (in the case of Receive.receive, the handler just returns the packet passed to it).

nesC therefore has default handlers. A default handler is an implementation of a function that's used if no implementation is wired in. If a component wires to the interface, then that implementation is used. Otherwise, the call (or signal) goes to the default handler. For example, CC2420ActiveMessageP has the following default handlers:

```
 default event message_t* Receive.receive[am_id_t id](message_t* msg, void* payload, uint8_t len) {
   return msg;
 }

 default event message_t* Snoop.receive[am_id_t id](message_t* msg, void* payload, uint8_t len) {
   return msg;
 }

default event void AMSend.sendDone[uint8_t id](message_t* msg, error_t err) {
  return;
}
```

Listing 6.18: Default events in an active message implementation

In the TestAM application, TestAMAppC wires TestAMC to ActiveMessageC.Receive[240]. Therefore, on the telos or micaz platform, when the radio receives a packet of AM type 240, it signals TestAMC.Receive.receive(). Since the application doesn't use any other protocols, when it receives an active message of any other type it signals CC2420ActiveMessageP's default handler.

Default handlers are dangerous, in that using them improperly can cause your code to stop working. For example, while CC2420ActiveMessageP has a default handler for Send.sendDone, TestAMC does not have a default handler for Send.send. Otherwise, you could forget to wire TestAMC.Send and the program would compile fine. That is, defaults should only be used when an interface is not necessary for the proper execution of a component. This almost always involves parameterized interfaces, as it's rare that all of the parameter values are used.

6.2 unique() and uniqueCount()

Parameterized interfaces were originally intended to support abstractions like active messaging. It turns out, however, that they are much more powerful than that. If you look at the structure of most basic TinyOS 2.0 abstractions, there's a parameterized interface in there somewhere. The ability to leave specify compile-time constants outside of modules, combined with dispatch, means that we can use parameterized interfaces to distinguish between many different callers. A component can provide a service through a parameterized interface, and every client that needs to use the service can wire to a different parameter ID. For split-phase calls, this means that you can avoid fan-out on the completion event. Consider these two examples:

```
components RouterC, SourceAC, SourceBC;
SourceAC.Send -> RouterC;
SourceBC.Send -> RouterC;
```

Listing 6.19: Single–wiring a Send

versus

```
components RouterC, SourceAC, SourceBC;
SourceAC.Send -> RouterC.Send[0];
SourceBC.Send -> RouterC.Send[1];
```

Listing 6.20: Wiring to a parameterized Send

In both cases, SourceAC and SourceBC can call Send.send. In the first case, when RouterC signals Send.sendDone, that signal will fan-out to both SourceAC and SourceBC, who will have to determine — by examining the message pointer, or internal state variables — whether the event is intended for them or someone else. In the second case, however, if RouterC keeps the parameter ID passed in the call to Send, then it can signal the appropriate completion event. E.g., SourceBC calls Send.send, RouterC stores the ID 1, and when it signals sendDone it signals it on Send.sendDone[1](...).

Let's return to the timer example, where this abstraction is particularly powerful. The timer component HilTimerMilliC has the following signature:

```
configuration HilTimerMilliC {
  provides interface Timer<TMilli>[uint8_t];
}
```

Listing 6.21: Partial HilTimerMilliC signature

Because Timer is parameterized, many different components can wire to separate interface instances. When a component calls Timer.startPeriodic, nesC fills in the parameter ID, which the timer implementation can use to keep track of which timer is being told to start. Similarly, the timer implementation can signal Timer.fired on specific timer instances.

For things such as network protocols, where the parameter to an interface is a basis for communication and interoperability, the actual parameter used is important. For example, if you have two different compilations of the same application, but one wires a protocol with

```
RouterC.Send -> ActiveMessageC.Send[210];
```

Listing 6.22: Wiring to Send instance 210

while the other wires it with

```
RouterC.Send -> ActiveMessageC.Send[211];
```

Listing 6.23: Wiring to Send instance 211

then they will not be able to communicate. In these cases, the parameter used is shared across nodes, and so needs to be globally consistent. Similarly, if you had two protocols wire to the same AM type, then this is a basic conflict that an application developer is going to have to resolve. Generally, protocols use named constants (enums) to avoid these kinds of typos.

With timers and the Send client example above, however, there is no such restriction. The parameter represents a unique client ID, rather than a piece of shared data. A client doesn't care which timer it wires to, as long as it wires to one that nobody else does. For this case, rather than force clients to guess Ids and hope there is no collision, nesC provides a special compile-time function, unique().

It is a compile-time function because it is resolved at compile time. When nesC compiles an application, it transforms all calls to unique() into an integer identifier. The unique function takes a string key as an argument, and promises that every instance of the function with the same key will return a unique value. Two calls to unique with different keys can return the same value. So if two components, AppOneC and AppTwoC, both want timers, they could do this

```
AppOneC.Timer -> HilTimerMilliC.Timer[unique("Timer")];
AppTwoC.Timer -> HilTimerMilliC.Timer[unique("Timer")];
```

Listing 6.24: Generating a unique parameter for the HilTimerMilliC's Timer interface

and be assured that they will have distinct timer instances.[4] If there are n calls to unique, then the unique values will be in the range of 0 – (n-1).

The combination of parameterized interfaces and the unique function allow services to provide a limited form of isolation between their clients (i.e., no fan-out on completion events). For example, in TinyOS 2.0, there are several situations when more than one component wants to access a shared resource. For timing reasons, fully virtualizing the resource (i.e., using a request queue) isn't feasible. Instead, the components need to be able to request the resource and know when it has been granted to them. TinyOS provides this mechanism through the Resource interface:

```
interface Resource {
  async command error_t request();
  async command error_t immediateRequest();
  event void granted();
  async command void release();
  async command uint8_t getId();
}
```

Listing 6.25: The Resource Interface

A component can request the resource either through request() or requestImmediate(). The latter returns SUCCESS only if the user was able to acquire the resource at that time and otherwise does nothing (it is a single-phase call). The request() call, in contrast, is split-phase, with the granted indicating that it is safe to use the resource.

Resource is for when multiple clients need to share the component. So TinyOS has Arbiters, which are components that institute a sharing policy between different clients. An arbiter providers a parameterized Resource interface:

```
configuration FcfsArbiterC {
  provides interface Resource[uint8_t id];
  ...
}
```

Listing 6.26: The First–Come–First–Served arbiter

Each client wires to a unique instance of the Resource interface, and the arbiter uses the client Ids to keep track of who has the resource.

[4] In practice, clients rarely call unique() directly. Instead, these calls are encapsulated inside generic components, which are discussed in the next chapter. One common problem with unique() encountered in TinyOS 1.x is that a mistyped key will generate a non-unique value and possibly cause very strange behavior.

In these examples – Timer and Resource – there is an additional factor to consider: each client requires the component to store some amount of state. For example, arbiters have to keep track of which clients have pending requests, and timer systems have to keep track of the period of each timer, how long until it fires, and whether it's active. Because the calls to unique define the set of valid client Ids, nesC has a second compile-time function, uniqueCount(). This function also takes a string key. If there are n calls to unique with a given key (returning values 0...n-1), then uniqueCount returns n, and this is resolved at compile-time.

Being able to count the number of unique clients allows a component to allocate the right amount of state to support them. Early versions of nesC didn't have the uniqueCount function: components were forced to allocate a fixed amount of state. If there were more clients than the state could support, one or more would fail at runtime. If there were fewer clients than the state could support, then there was wasted RAM. Because a component can count the number of clients and know the set of client Ids that will be used, it can promise that each client will be able to work and use the minimum amount of RAM needed. Returning to the timer example from above:

```
AppOneC.Timer -> HilTimerMilliC.Timer[unique("Timer")];
AppTwoC.Timer -> HilTimerMilliC.Timer[unique("Timer")];
```

Listing 6.27: Wiring to HilTimerMilliC with unique parameters

and HilTimerMilliC could allocate state for each client:

```
timer_t timers[uniqueCount("Timer")];
```

Listing 6.28: Counting how many clients have wired to HilTimerMilliC

Assuming the above two were the only timers, then HilTimerMilliC would allocate two timer structures. If we assume that AppOneC.Timer was assigned ID 0 and AppTwoC.Timer was assigned ID 1, then HilTimerMilliC can directly use the parameters as an index into the state array. This isn't how HilTimerMilliC works: it's actually a bit more complicated, as it uses generic components, which the next chapter discusses.

Chapter 7

Generic Components

Generic components (and typed interfaces) are the biggest addition in nesC 1.2. They are, for the most part, what lead TinyOS 2.0 to be significantly different than 1.x. Normally, components are singletons. That is, a component's name is a single entity in a global namespace. When two different configurations reference MainC, they are both referencing the same piece of code and state. In the world of C++ and Java, a singleton is similar (but not identical) to a static class.

Generic components are not singletons. They can be instantiated within a configuration. Take, for example, something like a bit vector. Many components need and use bit vectors. By having a single component that provides this abstraction, we prevent bugs by reducing code repetition. If you only have singletons, then every bit vector has to be a different component, each of which has a separate implementation. By making a bit vector a generic component, we can write it once and use it many times. This is the signature of BitVectorC, which we saw in an earlier chapter:

```
generic module BitVectorC( uint16_t max_bits ) {
  provides interface Init;
  provides interface BitVector;
}
```

Listing 7.1: The BitVectorC generic module

A configuration instantiates a generic component with the keyword *new*. This is constrast to singleton components, which are just named. Instantiation is private to the configuration, so every time new is used an instance is created. Instantiating modules copies their code. Therefore, while generic modules can simplify the source code of an application and prevent bugs from code copying, they do not inherently reduce the size of the final executable.

For example, the code

```
configuration ExampleVectorC {}
implementation {
  components new BitVectorC(10);
}
```

Listing 7.2: Instantiating a BitVectorC

creates a BitVectorC of size 10. This creates a copy of the BitVectorC code where every instance of the argument `max_bits` is replaced by the constant 10.

Generic components use a code-copying approach for two reasons: simplicity and types. If generic modules did not use a code-copying approach, then there would be a single copy of the code that works for all instances of the component. This is difficult when a generic component can take a type as a argument, as allocation size, offsets, and other considerations can make a truly single copy unfeasible (C++ templates, for example, create a single copy of code for each template type). Code sharing within identical types requires adding an argument, such as a pointer, to all of the functions. This argument indicates which instance is executing. Additionally, all variable accesses would have to offset from this pointer. In essence, the execution time and costs of functions could change significantly (offsets rather than constant accesses). In order to provide simple, easy to understand and run-time efficient components, nesC uses a code-copying approach, sacrificing possible reductions in code size.

A generic component is private to the configuration that instantiates it, as no other configuration can name it. However, the instantiator can make the generic accessible to other components by exporting its interfaces. If the exporting configuration is a singleton, then it can provide a globally accessible name to the component within (the section below on the TinyOS 2.0 timer system describes an example of this approach).

Code-copying applies to configurations as well as modules. One (reasonably accurate) way to think of a generic component is that it literally creates a copy of the source file. In the case of modules, this is executable code and variables that will be in the final application. In the case of configurations, however, this copying can instantiate other components and create wirings. A generic module defines a piece of repeatable executable logic: a generic configuration defines a repeatable pattern of composition between components.

Generic components differ in syntax from singleton components by having an argument list. Generics with no arguments have an empty list, just like a function. For example:

```
configuration TimerMilliImplC {...}
generic configuration TimerMilliC() {...}

module Msp430ClockP {...}
```

```
generic module VirtualizeTimerC(typedef precision_tag, int max_timers) {...}
```

Listing 7.3: Generic Component Syntax

Both generics have a argument list. TimerMilliC takes no arguments, however, so the list is empty. VirtualizeTimerC, in contrast, takes two arguments, a type (used to check precision) and a count of the number of timers it should provide. Generic components support three types for arguments:

1. Types: these can be arguments to typed interfaces
2. Numeric constants
3. Constant strings

7.1 Generic Modules

BitVectorC is a generic module with a single argument, a uint16_t. If an argument is a type, then it is declared with the typedef keyword. For example, HilTimerMilliC is often built on top of a single timer. Components in the timer library (lib/timer) virtualize the single underlying timer into many timers. The timer interface, however, has a type as an argument to ensure that the precision requirements are met. This means that a component which virtualizes timers must have this type passed into it. This is the signature of VirtualizeTimerC:

```
generic module VirtualizeTimerC( typedef precision_tag, int max_timers ) {
  provides interface Timer<precision_tag> as Timer[ uint8_t num ];
  uses interface Timer<precision_tag> as TimerFrom;
}
```

Listing 7.4: The VirtualizeTimerC generic module

It is a generic module with two arguments. The first argument is the timer precision tag, which is a type and therefore has a type of typedef. In the case of HilTimerMilliC, this is TMilli. This precision tag provides additional type checking for an interface. Rather than have separate Timer interfaces for different time fidelities, TinyOS uses a single Timer interface with the fidelity as an argument. The argument itself is an empty struct that is never allocated; because it is a struct, it is very unlikely that there will be inadvertent implicit casts.

The second argument is the number of virtualized timers the component provides. This is usually computed with a uniqueCount(). Because VirtualizeTimerC is a module, instantiating one will allocate the necessary state. As generic modules create code copies, the lines

```
components new VirtualizeTimerC(TMilli, 3) as TimerA;
components new VirtualizeTimerC(TMilli, 4) as TimerB;
```

Listing 7.5: Instantiating two VirtualizeTimerC components

generate two copies of the VirtualizeTimerC's code, one of which allocates three timers, the other of which allocates four. Again, nesC does this because different instances of VirtualizeTimerC might have different types and different constants. For example, the max_timers argument can be used in loops, say, when checking if timers are pending. Rather than go for an object-oriented (or C++ template-like) approach of passing data pointers around, nesC just creates copies of the code. Because all of these copies come from a single source file, they are all consistent and don't create maintenance problems in the way that multiple source files can.

7.2 HilTimerMilliC: An Example Use of Generic Components

Implementing a solid and efficient timer subsystem is very difficult. TinyOS 2.x makes the task simpler by having a library of reusable components (`lib/timer`) that provide many of the needed pieces of functionality. Each supported microcontroller provides a timer system by layering these library components on top of some low-level hardware abstractions. Because many microcontrollers have several clock sources, most of these library components are generic components, so that a platform can readily provide several timer systems of different fidelities.

For example, this is the full code for HilTimerMilliC on the micaZ platform (defined in platform/mica):

```
\#include "Timer.h"

configuration HilTimerMilliC {
  provides interface Init;
  provides interface Timer<TMilli> as TimerMilli[uint8_t num];
  provides interface LocalTime<TMilli>;
}
implementation {

  enum {
    TIMER_COUNT = uniqueCount(UQ_TIMER_MILLI)
  };

  components AlarmCounterMilliP, new AlarmToTimerC(TMilli),
    new VirtualizeTimerC(TMilli, TIMER_COUNT),
    new CounterToLocalTimeC(TMilli);

  Init = AlarmCounterMilliP;
```

```
  TimerMilli = VirtualizeTimerC;
  VirtualizeTimerC.TimerFrom -> AlarmToTimerC;
  AlarmToTimerC.Alarm -> AlarmCounterMilliP;

  LocalTime = CounterToLocalTimeC;
  CounterToLocalTimeC.Counter -> AlarmCounterMilliP;
}
```

Listing 7.6: The full code of HilTimerMilliC

The only singleton component in this configuration is AlarmCounterMilliP, which is an abstraction of a low-level microcontroller timer. HilTimerMilliC uses three generic components on top of AlarmCounterMilliP to provide a full timer system:

- CounterToLocalTimerC turns a hardware counter into a local timebase;
- AlarmToTimerC turns an Alarm interface, which provides an async one-shot timer, into a Timer interface, which provides a sync timer with greater functionality;
- VirtualizeTimerC vitualizes a single Timer into many Timers, specified by an argument to the component.

All of the generics have the type TMilli as one of their arguments. These type arguments make sure that timer fidelities are not accidentally changed. E.g., VirtualizeTimerC takes a single timer of fidelity precision_tag and virtualizes it into timer_count timers of precision_tag. The timer library has components that translate between precisions.

UQ_TIMER_MILLI is a #define (from Timer.h) for the string "HilTimerMilliC.Timer". This is a common approach in TinyOS 2.x. Using a #define makes it harder to run into bugs caused by errors in the string: chances are that a typo in the define will be a compile time error. This is generally good practice for components that depend on unique strings.

Programming Hint 13: If a component depends on unique, then #define a string to use in a header file, to prevent bugs from string typos.

The first thing HilTimerMilliC does is define an enum for the number of timers being used. It assumes that each timer has wired to TimerMilli with a call to unique(UQ_TIMER_MILLI). It takes an async, hardware timer — AlarmCounterMilliP — and turns it into a virtualized timer. It does this with three steps. The first step turns the Alarm (the async timer) into a Timer, with the generic component AlarmToTimerC:

```
AlarmToTimerC.Alarm -> AlarmCounterMilliP;
```

Listing 7.7: Turning an Alarm into a Timer

The second step virtualizes a single timer into many timers:

```
VirtualizeTimerC.TimerFrom -> AlarmToTimerC;
```

Listing 7.8: Virtualizing a Timer

It then exports the parameterized timer interface:

```
TimerMilli = VirtualizeTimerC;
```

Listing 7.9: Exporting the virtualized Timer interface

Additionally, some aspects of the timer system require being able to access a time base, for example, to specify when in the future a timer fires. So HilTimerMilliC takes a hardware counter and turns it into a local time component,

```
CounterToLocalTimeC.Counter -> AlarmCounterMilliP;
```

Listing 7.10: Wiring to a counter for a time base

then exports the interface:

```
LocalTime = CounterToLocalTimeC;
```

Listing 7.11: Exporting the LocalTime interface

Many of the components in the timer library are generics because a platform might need to provide a wide range of timers. For example, depending on the number of counters, compare registers, and their width, a platform might provide millisecond, microsecond, and 32kHz timers. The variants of the MSP430 chip family that some platforms use, for example, can easily provide millisecond and 32kHz timers with a very low interrupt load: their compare registers are 16 bits, so even at 32kHz they only fire one interrupt every two seconds.

Generic modules work very well for abstractions that have to allocate per-client state, such as timers or resource arbiters. A generic module allows you to specify the size — the number of clients — in the configuration that instantiates the module, rather than within the module itself. For example, if VirtualizeTimerC were not a generic, then inside its code there would have to be a uniqueCount() with the proper key.

Unlike standard components, generics can only be named by the configuration that instantiates them. For example, in the case of HilTimerMilliC, no other component can wire to the VirtualizeTimerC that it instantiates. The generic is private to HilTimerMilliC. The only way it can be made accessible is to export its interfaces (which HilTimerMilliC does). This is how you can make an instance that many components can wire to. You create a singleton by writing a configuration with the identical signature and just exporting all of the interfaces. For example, let's say you needed a bit vector to keep track of which system services are running or not. You want many components to be able to access this vector, but BitVectorC is a generic. So you write a component like this:

```
configuration SystemServiceVectorC {
  provides interface BitVector;
}
implementation {
  components MainC, new BitVectorC(uniqueCount(UQ_SYSTEM_SERVICE));
  MainC.SoftwareInit -> BitVectorC;
  BitVector = BitVectorC;
}
```

Listing 7.12: The fictional component SystemServiceVectorC

Now many components can refer to this particular bit vector. While you can make a singleton out of a generic by instantiating it within one, the opposite is not true: a component is either instantiable or not.

7.3 Generic Configurations

Generic modules are a way to reuse code and separate common abstractions into well-tested building blocks (there only needs to be one FIFO queue implementation, for example). nesC also has generic configurations, which are a very powerful tool for building TinyOS abstractions and services. However, just as configurations are harder for a novice programmer to understand than modules, generic configurations are a bit more challenging than generic modules.

The best way to describe what role a generic configuration can play in a software design is to start from first principles:

A module is a component that contains executable code; A configuration defines relationships between components to form a higher-level abstraction; A generic module is reusable piece of executable code; therefore, A generic configuration is a reusable set of relationships that form a higher-level abstraction.

Several examples in this book have mentioned and described HilTimerMilliC. But if you look at TinyOS code, there is only one component that references it. Although it is a very important component, programs

never directly name it. It is the core part of the timer service, but applications that need timers instantiate a generic component named TimerMilliC.

Before delving into generic configurations, however, let's consider what code looks like without them. Let's say we have HilTimerMilliC, and nothing more. Many components need timers; HilTimerMilliC enables this through its parameterized interface. Remember that HilTimerMilliC encapsulates an instance of VirtualizeTimerC, whose size parameter is a call to unique(UQ_TIMER_MILLI). This means that if a component AppP needs a timer, then its configuration AppC must wire it like this:

```
configuration AppP {...}
implementation {
  components AppP, HilTimerMilliC;
  AppP.Timer -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

Listing 7.13: Wiring directly to HilTimerMilliC

Now, let's say that AppP actually needs three timers. The code would look like this:

```
configuration AppP {...}
implementation {
  components AppP, HilTimerMilliC;
  AppP.Timer1 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
  AppP.Timer2 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
  AppP.Timer3 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

Listing 7.14: Wiring directly to HilTimerMilliC three times

This approach can work fine: it's how TimerC in TinyOS 1.x works. But it does have some issues. First, there are references to UQ_TIMER_MILLI is sprinkled throughout many components in the system: changing the identifier used is not really possible. This is especially true because a call to unique() with the incorrect (but valid) parameter will not return an error. For example, if a component did this

```
AppP.Timer1 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MICRO)];
```

Listing 7.15: A buggy wiring to HilTimerMilliC

by accident, then there will be two components wiring to the same instance of Timer and the program will probably exhibit really troublesome behavior. The issue is that a detail about the internal implementation of the timer system — the key for unique that it uses — has to be exposed to other components. Usually, all a component wants to do is allocate a new timer. It doesn't care — and shouldn't have to care — about how it is implemented.

Generic configurations simplify this by defining a wiring pattern that can be instantiated. For example, all of above users of timer just want to wire to a millisecond timer; they shouldn't have to worry about unique keys, which provide details of the HilTimerMilliC implementation and are an easy source of bugs. The basic wiring pattern is this:

```
AppP.Timer1 -> HilTimerMilliC.TimerMilli[unique(UQ_TIMER_MILLI)];
```

Listing 7.16: The wiring pattern to HilTimerMilliC

The next section describes how TimerMilliC codifies this pattern to simplify using timers. It turns out that generic configurations have a much broader (and more powerful) set of uses than simple unique key management, and the next section covers some of these as well.

7.4 Examples

Because generic configurations are a challenging concept, we present four examples of their use for basic abstractions in the TinyOS core. The examples increase in complexity.

7.4.1 TimerMilliC

The standard millisecond timer abstraction, TimerMilliC, provides this abstraction. TimerMilliC is a generic configuration that provides a single Timer interface. Its implementation wires this interface to an instance of the underlying parameterized Timer interface using the right unique key. This means that unique() is called in only one file; as long as all components allocate timers with TimerMilliC, there is no chance of a key match mistake. TimerMilliC's implementation is very simple:

```
generic configuration TimerMilliC() {
  provides interface Timer<TMilli>;
}
implementation {
  components TimerMilliP;
  Timer = TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];
}
```

Listing 7.17: The TimerMilliC generic configuration

TimerMilliP is a singleton configuration that auto-wires HilTimerMilliC to the boot sequence and exports HilTimerMilliC's parameterized interface:

```
configuration TimerMilliP {
  provides interface Timer<TMilli> as TimerMilli[uint8_t id];
}
implementation {
  components HilTimerMilliC, MainC;
  MainC.SoftwareInit -> HilTimerMilliC;
  TimerMilli = HilTimerMilliC;
}
```

Listing 7.18: TimerMilliP auto–wires HilTimerMilliC to Main.SoftwareInit

TimerMilliC encapsulates a wiring pattern — wiring to the timer service with a call to unique — for other components to use. When a component instantiates a TimerMilliC, it creates a copy of the TimerMilliC code, which includes a call to unique(UQ_TIMER_MILLI). The line of code

```
components X, new TimerMilliC();
X.Timer -> TimerMilliC;
```

Listing 7.19: Instantiating a TimerMilliC

is essentially this:

```
components X, TimerMilliP;
X.Timer -> TimerMilliP.TimerMilli[unique(UQ_TIMER_MILLI)];
```

Listing 7.20: Expanding a TimerMilliC Instantiation

TimerMilliP is itself a configuration, which wires to HilTimerMilliC, which is a configuration. When a component calls Timer.start() on a TimerMilliC, the actual function it invokes is Timer.start() on VirtualizeTimerC. Let's step through the complete wiring path for an application that creates a timer. BlinkAppC wires the BlinkC module to its three timers:

```
configuration BlinkAppC{}
implementation {
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  MainC.SoftwareInit -> LedsC;

  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

Listing 7.21: The Blink application

Wiring BlinkC.Timer0 to Timer0 establishes this wiring chain (the key to unique, UQ_TIMER_MILLI, is elided for readability):

```
BlinkC.Timer0 -> Timer0.Timer
Timer0.Timer = TimerMilliP.TimerMilli[unique(...)]
TimerMilliP.TimerMilli[unique(...)] = HilTimerMilliC[unique(...)]
HilTimerMilliC[unique(...)] = VirtualizeTimerC.Timer[unique(...)]
```

Listing 7.22: The full module–to–module wiring chain in Blink (BlinkC to VirtualizeTimerC)

BlinkC and VirtualizeTimerC are the two modules; the intervening components are all configurations. When nesC compiles this code, all of the intermediate layers will be stripped away, and BlinkC.Timer0.start will be a direct function call on VirtualizeTimerC.Timer[...].start.

Many of TinyOS's basic services use this pattern of a generic configuration to managing a keyspace for a parameterized interface. For example, one of the non-volatile storage abstractions in TinyOS is BlockStorageC (covered in TEP 103). This abstraction is intended for reading and writing large objects in a random access fashion. This abstraction provides the BlockRead and BlockWrite interfaces. The abstraction supports there being multiple readers and writers with a similar pattern to what Timer uses, although unlike Timer only one read or write can be outstanding at any time. The underlying implementation therefore keeps track of who's outstanding and enqueues other requests.

7.4.2 AMSenderC

TimerMilliC is reasonably simple: all it really does is encapsulate a wiring with unique() in order to make sure there aren't client collisions and in order to simplify wiring. Because HilTimerMilliC has to know the state of all of the outstanding timers in order to do its job well, it provides a virtualized abstraction, which TimerMilliC can just export.

Active messages are slightly different. The basic platform active message component, ActiveMessageC, provides AMSend, parameterized by the AM id. However, ActiveMessageC can only have a single packet outstanding at any time. If it is already sending a packet and a component calls SendAM.send, ActiveMessageC returns FAIL or EBUSY. From the perspective of a caller, this is a bit of a pain. If it wants to send the packet, it has to wait until the radio is free, but doesn't have a very easy way of figuring out when this will occur.

TinyOS 1.x had a global (not parameterized) sendDone event, which the radio would signal whenever it finished sending any packet. That way, if a component tried to send and received a FAIL, it could try to resent when it handled the sendDone event. This mostly works, except that if multiple components wire to sendDone, then the fan-out determines the priority of the send requests. E.g., if a hog of a component handles sendDone and happens to be first in the fan-out, it will always get first dibs and will monopolize the radio.

TinyOS 2.x solves this problem through the AMSender component, which is a generic configuration. AMSender is a virtualized abstraction: every instance of AMSender acts like ActiveMessageC. That is, each AMSender can handle a single outgoing packet. This means that each component that wires to an AMSender can act independently of the other components, and not worry about fan-out scheduling. The one-deep queue of ActiveMessageC is replaced by N one-deep queues, one for each of the N clients.

Each AMSenderC having its own one-deep queue is not sufficient. There's also the question of what order the senders get to send their packets. Under the covers, what the active message layer does is maintain an array of N pending packets, where N is the number of AMSenderC components. Each AMSenderC is a client of the active message sending abstraction, and so has a client ID that indexes into this array. The implementation keeps track of the last client that was able to send a packet, and makes sure that everyone else waiting gets a chance before that client does again.

Accomplishing this is a little tricker than TimerMilliC, because a request to send has a few parameters. With Timer, those parameters (period, single-shot vs. repeat) are state that the timer implementation has to keep track of in the first place. With the AMSenderC, it's a bit different: those parameters just need to be stored until the call to the underlying ActiveMessageC. The send queue could just store all of these parameters, that uses up 4 extra bytes of RAM per entry (2 for the destination, 1 for the AM type, and 1 for the length).

It turns out that the Packet and AMPacket interfaces have operations exactly for this situation. They allow a component to get and set packet fields. For example, a component can call Packet.setLength to set the length field and recover it with Packet.length. For components that just need basic send or receive abstractions , they can just use AMSend or Receive. The Packet interface, though, allows data structures such as queues to store temporary state within the packet and then recover it when it's time to actually send so it can be passed as parameters. This means that the AM send queue with n clients allocates a total of (2n +1) bytes of state, as pointers on microcontrollers are usually 2 bytes (on the intelmote2, though, they're four bytes, so it allocates 4n+1).

This means that the AMSenderC abstraction needs to do the following things:

1. Provide an AMSend interface
2. Store the AMSend.send parameters before putting a packet on the queue
3. Statically allocate a single private queue entry
4. Store a send request packet in the queue entry when it's not full
5. When it's actually time to send the packet, reconstitute the send parameters and call ActiveMessageC

What makes this abstraction more tricky is that there are two keyspaces. ActiveMessageC provides AMSend based on the AM type keyspace. The send queue, in contrast, has a client ID keyspace for keeping track of which AMSenderC is sending. Because the queue needs to be able to send any AM type, it uses a parameterized AMSend and directly wires to ActiveMessageC.AMSend. So the overall structure goes something like this:

1. Component calls AMSenderC.AMSend.send
2. This calls AMSenderP, which stores the length, AM id, and destination in the packet
3. AMSenderP is a client of AMSendQueueP and call Send.send with its client ID
4. AMSendQueueP checks that the queue entry is free and puts the packet into it.
5. Some time later, AMSendQueueP pulls the packet off the queue and calls AMSend.send on ActiveMessageC with the parameters that AMSenderP stored.
6. When ActiveMessageC signals AMSend.sendDone, AMSendQueueP signals Send.sendDone to AMSenderP, which signals AMSend.sendDone to the original calling component.

This is the code for AMSenderC:

```
generic configuration AMSenderC(am_id_t AMId) {
  provides {
    interface AMSend;
    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements as Acks;
  }
}

implementation {
  components new AMQueueEntryP(AMId) as AMQueueEntryP;
```

```
  components AMQueueP, ActiveMessageC;

  AMQueueEntryP.Send -> AMQueueP.Send[unique(UQ_AMQUEUE_SEND)];
  AMQueueEntryP.AMPacket -> ActiveMessageC;

  AMSend = AMQueueEntryP;
  Packet = ActiveMessageC;
  AMPacket = ActiveMessageC;
  Acks = ActiveMessageC;
}
```

Listing 7.23: The AMSenderC generic configuration

A send queue entry is just responsible for storing send information in a packet:

```
generic module AMQueueEntryP(am_id_t amId) {
  provides interface AMSend;
  uses{
    interface Send;
    interface AMPacket;
  }
}

implementation {

  command error_t AMSend.send(am_addr_t dest,
                              message_t* msg,
                              uint8_t len) {
    call AMPacket.setDestination(msg, dest);
    call AMPacket.setType(msg, amId);
    return call Send.send(msg, len);
  }

  command error_t AMSend.cancel(message_t* msg) {
    return call Send.cancel(msg);
  }

  event void Send.sendDone(message_t* m, error_t err) {
    signal AMSend.sendDone(m, err);
  }

  command uint8_t AMSend.maxPayloadLength() {
    return call Send.maxPayloadLength();
  }

  command void* AMSend.getPayload(message_t* m) {
    return call Send.getPayload(m);
  }

}
```

Listing 7.24: AMSendQueueEntryP

The queue itself sits on top of ActiveMessageC:

```
configuration AMQueueP {
  provides interface Send[uint8_t client];
}

implementation {
  components AMQueueImplP, ActiveMessageC;

  Send = AMQueueImplP;
  AMQueueImplP.AMSend -> ActiveMessageC;
  AMQueueImplP.AMPacket -> ActiveMessageC;
  AMQueueImplP.Packet -> ActiveMessageC;

}
```

Listing 7.25: AMQueueP

Finally, within AMSendQueueImplP, the logic to send a packet looks like this:

```
nextPacket();
if (current == QUEUE_EMPTY) {
  return;
}
else {
  message_t* msg;
  am_id_t id;
  am_addr_t addr;
  uint8_t len;
  msg = queue[current];
  id = call AMPacket.getType(msg);
  addr = call AMPacket.getDestination(msg);
  len = call Packet.getLength(msg);
  if (call AMSend.send[id](addr, msg, len) == SUCCESS) {
   ...
...
}
```

Listing 7.26: AMSendQueueImplP pseudocode

7.4.3 CC2420SpiC

Another, more complex example of using generic congfigurations is CC2420SpiC. This component provides access to the CC2420 radio over an SPI bus. When the radio stack software wants to interact with the radio, it makes calls on an instance of this component. For example, telling the CC2420 to send a packet if there is a clear channel involves writing to one of the radio's registers (TXONCCA). To write to the register, the stack sends a small series of bytes over the bus, which basically say "I'm writing to register number X with

value Y." The very fast speed of the bus means that small operations such as these can made synchronous without any significant concurrency problems.

In addition to small register reads and writes, the chip also supports accessing the receive and transmit buffers, which are 128-byte regions of memory, as well as the radio's configuration memory, which stores things such as cryptographic keys and the local address (which is used for determining whether to send an acknowledgment). These operations are split-phase. For example, before the stack writes to TXONCCA to send a packet, it must first execute a split-phase write with the CC2420Fifo interface (the receive and transmit buffers are FIFO memories).

All of the operations boil down to four interfaces:

- **CC2420Strobe**: Access to command registers. Writing a command register tells the radio to take an action, such as transmit a packet, clear its packet buffers, or transition to transmit mode. This interface has a single command, strobe, which writes to the register.
- **CC2420Register**: Access to data registers. These registers can be both read and written, and store things such as hardware configuration, addressing mode, and clear channel assessment thresholds. This interface supports reads and writes as single-phase operations.
- **CC2420Ram**: Access to configuration memory. This interface supports both reads and writes, as split-phase operations.
- **CC2420Fifo**: Access to the receive and transmit FIFO memory buffers. This interface supports both reads and writes, as split-phase operations. While one can write to the receive buffer, the CC2420 supports this only for debugging purposes.

A component that needs to interact with the CC2420 instantiates an instance of CC2420SpiC:

```
generic configuration CC2420SpiC() {

  provides interface Resource;

  provides interface CC2420Strobe as SFLUSHRX;
  provides interface CC2420Strobe as SFLUSHTX;
  provides interface CC2420Strobe as SNOP;
  provides interface CC2420Strobe as SRXON;
  provides interface CC2420Strobe as SRFOFF;
  provides interface CC2420Strobe as STXON;
  provides interface CC2420Strobe as STXONCCA;
  provides interface CC2420Strobe as SXOSCON;
  provides interface CC2420Strobe as SXOSCOFF;
```

```
  provides interface CC2420Register as FSCTRL;
  provides interface CC2420Register as IOCFG0;
  provides interface CC2420Register as IOCFG1;
  provides interface CC2420Register as MDMCTRL0;
  provides interface CC2420Register as MDMCTRL1;
  provides interface CC2420Register as TXCTRL;

  provides interface CC2420Ram as IEEEADR;
  provides interface CC2420Ram as PANID;
  provides interface CC2420Ram as SHORTADR;
  provides interface CC2420Ram as TXFIFO_RAM;

  provides interface CC2420Fifo as RXFIFO;
  provides interface CC2420Fifo as TXFIFO;
}
```

Listing 7.27: CC2420SpiC

CC2420SpiC takes the implementation of the SPI protocol (CC2420SpiP) and wires it to the platform's raw SPI implementation. The raw SPI implementation has two interfaces: SPIByte, for writing a byte as a single-phase operation, and SpiPacket, for writing a series of bytes as a split-phase operation.[1] The protocol implementation uses an interesting approach to have a simple implementation that can also find compile-time errors. While CC2420SpiC provides each register as a separate interface, CC2420SpiP provides a parameterized interface:

```
configuration CC2420SpiP {
  provides interface CC2420Fifo as Fifo[ uint8_t id ];
  provides interface CC2420Ram as Ram[ uint16_t id ];
  provides interface CC2420Register as Reg[ uint8_t id ];
  provides interface CC2420Strobe as Strobe[ uint8_t id ];
}
```

Listing 7.28: CC2420SpiP

Each CC2420 register has a unique identifier, which is a small integer. Having a separate implementation for each register operation wastes code space and the code repetition would be an easy way to introduce bugs. So CC2420SpiP has a single implementation, which takes a compile-time parameter, the register identifier. However, not all values of a uint8_t are valid registers, so allowing components to wire directly to the parameterized interface could lead to invalid wirings. Of course, CC2420SpiP could incorporate some run-time checks to make sure that register values are valid, but this wastes CPU cycles, especially when the

[1] The SPI protocol is bidirectional. To read bytes from the chip, the stack has to write onto the bus. The chip also writes onto the bus, but it is clocked by the CPU's writes. The write operation therefore takes a uint8_t as the byte to write, and a uint8_t* as a pointer to where it should store the reply.

parameters are almost always valid. So CC2420SpiC maps the a subset of the valid parameters into interface instances. It only maps a subset because there are some debugging registers the stack doesn't need to use. The implementation looks like this:

```
configuration CC2420SpiC { ...}
implementation {
  ...
  components CC2420SpiP as Spi;

  SFLUSHRX = Spi.Strobe[CC2420_SFLUSHRX];
  SFLUSHTX = Spi.Strobe[CC2420_SFLUSHTX];
  SNOP = Spi.Strobe[CC2420_SNOP];
  SRXON = Spi.Strobe[CC2420_SRXON];
  SRFOFF = Spi.Strobe[CC2420_SRFOFF];
  STXON = Spi.Strobe[CC2420_STXON];
  STXONCCA = Spi.Strobe[CC2420_STXONCCA];
  SXOSCON = Spi.Strobe[CC2420_SXOSCON];
  SXOSCOFF = Spi.Strobe[CC2420_SXOSCOFF];
}
```

Listing 7.29: CC2420SpiC mappings to CC2420SpiP

This approach gives us the best of both worlds: there is a single function for writing to a strobe register, which takes as an argument which register to write to, and the argument does not need run-time checking. However, the caller does not have to provide this identifier, and so cannot by accident specify an invalid one. Components that wire to CC2420SpiC can only wire to valid strobe registers, and rather than

```
call CC2420Strobe.strobe(CC2420_STXONCCA);
```

Listing 7.30: Strobing a register with a runtime parameter

they write

```
call TXONCCA.strobe();
```

Listing 7.31: Strobing a register through wiring

One issue that arises with CC2420SpiC is that multiple components might want to interact with the radio at the same time. For example, in the CC2420 implementation, the receive and transmit paths have (almost) completely separate logic. The radio might signal that a packet has arrived while a component is trying to send. Only one of these can access the bus (e.g., to read the RXFIFO or to write the TXFIFO) at any time. So the hardware presentation layer (HPL) of the SPI bus has a resource arbiter. Each user of the SPI bus is an arbiter client.

Combining these two abstractions together, an instance of CC2420SpiC needs to do two things:

1. Map named interface instances to parameters
2. Instantiate a client to the SPI bus

HplCC2420SpiC is the component representing a client instance to the SPI bus. It signature is this:

```
generic configuration HplCC2420SpiC() {
  provides interface Init;
  provides interface Resource;
  provides interface SPIByte;
  provides interface SPIPacket;
}
```

Listing 7.32: HplCC2420SpiC

The Init, SpiByte, and SPIPacket interfaces are rather simple: they just directly export the actual HPL bus abstraction's interfaces. Resource is an exported instance of the SPI arbiter's parameterized Resource interface. The instance parameter is generated with a call to unique(). Iinstantiating CC2420SpiC creates an instance of HplCC2420SpiC so it can arbitrate for access to the bus, but actually exports the interfaces of CC2420SpiP. Here's the implementation:

```
generic configuration CC2420SpiC {...}
implementation {
  components HplCC2420PinsC as Pins;
  components new HplCC2420SpiC();
  components CC2420SpiP as Spi;

  Init = HplCC2420SpiC;
  Resource = HplCC2420SpiC;

  SFLUSHRX = Spi.Strobe[ CC2420_SFLUSHRX ];
   ...
}
```

Listing 7.33: The implementation of CC2420SpiC

Let's look at what this means at a function level. CC2420SpiC.SNOP is the interface for the no-op strobe register. CC2420SpiC wires it like this:

```
  SNOP = Spi.Strobe[ CC2420_SNOP ];
```

Listing 7.34: Wiring a strobe register

where Spi is CC2420SpiP. This means that every component which wires to SNOP on an instance of CC2420SpiC wires to Spi.Strobe[CC242_SNOP]. This wiring ultimately terminates in the component CC2420SpiImplP:

```
async command cc2420_status_t Strobe.strobe[ uint8_t addr ]() {
  cc2420_status_t status;
  call SPIByte.write( addr, \&status );
  return status;
}
```

Listing 7.35: The strobe implementation

which writes a single byte to the bus and reads the status result. To step through each layer,

1. A component (e.g., CC2420TransmitP) calls SNOP.strobe() on an instance of CC2420SpiC
2. The nesC wiring transforms this call into CC2420SpiP.Strobe[CC2420_SNOP].strobe()
3. After the naming transformations, this actuall calls CC2420SpiImplP.Strobe[CC2420_SNOP].strobe()

Note that since the layer of indirection between 2) and 3) is purely wiring, the nesC compiler removes it. After all of the optimizations and inlining, the statement

```
call SNOP.strobe()
```

Listing 7.36: Strobing the SNOP register

effectively becomes

```
cc2420_status_t status;
call SPIByte.write(CC2420_SNOP, \&status);
```

Listing 7.37: The resulting C code of strobing the SNOP register

with possible optimizations across even the write function call boundary (it might just inline the SPI into the function, removing any need for function calls).

7.4.4 BlockStorageC

BlockStorageC is one of the most complicated example uses of generics, because it deals with four different sets of parameterized interfaces. In TinyOS 2.x, non-volatile storage is divided into volumes. A volume is a contiguous region of storage with a certain format and that can be accessed with an associated interface.

TEP 103 defines two basic formats: Logging and Block.[2] Logging is for append-only writes and streaming reads and Block is for random-access reads and writes. Logging has the advantage that its more limited interface allows for atomic operations: when a write to a Logging volume completes, it guaranteed to be written. In contrast, the Block interface has a separate commit operation.

Every volume has a unique identifier, and every BlockStorageC is associated with a single volume. However, there can be multiple BlockStorageC components that access the same volume, and not all volumes may have BlockStorageC components. A client has to be associated with a volume so that the underlying code can generate an absolute offset into the chip from a relative offset within a volume. E.g., if a 1MB chip is divided into two 512kB volumes, then address 16k on volume 1 is address 528k on the chip. This means that there are at least two keyspaces. The first keyspace is the volume ID keyspace, and the second is the client ID keyspace.

In practice, there is a third keyspace, clients to the arbiter for the HAL of the storage chip. This keyspace is shared between block clients, logging clients, and other abstractions that need exclusive access to the chip. So, all in all, BlockStorageC has to manage three different keyspaces:

1. Client key: which block storage client this is (for block storage client state)
2. Chip key: which client to the storage chip this is (for arbitration of the shared resource)
3. Volume key: which volume this client accesses (for calculating absolute offsets in the chip)

One of the most difficult parts of nesC programming is understanding how parameterized interfaces can be used, and how to manage their keyspaces. BlockStorageC is a good example, because it represents a non-trivial use of wiring to build a simple abstraction from a bunch of underlying components and how to make them interact properly.

Both the client key and chip key are generated with unique(). The client key is only among BlockStorageC components, so it uses a string UQ_BLOCK_STORAGE defined in BlockStorage.h. The chip key is shared across all components that use underlying chip. In the case of the at45db chip (used in the micaZ platform), the string UQ_AT45DB defined in At45db.h. The volume key is not generated by unique, as, among other things, it must remain consistent across compiles. It is specified in a header file that describes the volumes defined for the storage chip.

After all of that introduction, you might think that BlockStorageC is many lines of code. It isn't: it only has four wiring statements, which we'll step through one by one:

[2] There is also a third format, Configure, used for small items of configuration data, but this requires a different kind of chip than that which Logging and Block usually sit on top of.

```
generic configuration BlockStorageC(volume_id_t volid) {
  provides {
    interface BlockWrite;
    interface BlockRead;
  }
}
implementation {
  enum {
    BLOCK_ID = unique(UQ_BLOCK_STORAGE),
    RESOURCE_ID = unique(UQ_AT45DB)
  };

  components BlockStorageP, WireBlockStorageP, StorageManagerC, At45dbC;

  BlockWrite = BlockStorageP.BlockWrite[BLOCK_ID];
  BlockRead = BlockStorageP.BlockRead[BLOCK_ID];

  BlockStorageP.At45dbVolume[BLOCK_ID] -> StorageManagerC.At45dbVolume[volid];
  BlockStorageP.Resource[BLOCK_ID] -> At45dbC.Resource[RESOURCE_ID];
}
```

Listing 7.38: BlockStorageC

The first two lines,

```
BlockWrite = BlockStorageP.BlockWrite[BLOCK_ID];
BlockRead = BlockStorageP.BlockRead[BLOCK_ID];
```

Listing 7.39: Wiring BlockWrite and BlockRead as a unique client

make the BlockWrite and BlockRead interfaces clients of the service that implements them, BlockStorageP. When a component wired to a BlockStorageC calls to read or write from a block, nesC automatically includes a client ID into the call by the time it reaches the implementation.

The next line

```
BlockStorageP.At45dbVolume[BLOCK_ID] -> StorageManagerC.At45dbVolume[volid];
```

Listing 7.40: Wiring a BlockStorage client to its volume

translates between the client and volume keyspaces. When BlockStorageP makes a call on the StorageManagerC, it includes the client ID in the call as an outgoing parameter. This client ID is bound to a volume ID. nesC automatically builds a switch statement that translates between the two, so what when StorageManagertC receives the call, nesC has filled in the volume ID as the parameter. Note that there can be a many-to-one mapping between client IDs and volume IDs, as there might be multiple clients that access the same volume.

The final line,

```
BlockStorageP.Resource[BLOCK_ID] -> At45dbC.Resource[RESOURCE_ID];
```

Listing 7.41: Wiring a BlockStorage client as a client of the flash Resource

is what allows the block storage client to cooperate with other clients (blocking and logging) for access to the actual flash chip. BlockStorageP makes each of its clients a client of the flash chip resource manager.

Overall, the logic goes like this:

1. A component accesses volume V through an instance of BlockStorageC with client id C
2. The component calls BlockStorageC to read from a block
3. It becomes a call on BlockStorageP with with parameter C
4. BlockStorageP notes that there is a call pending, stores the arguments in the state allocated for C, and requests the Resource with C, which maps to resource id R
5. BlockStorageP receives the resource for client R (which maps back to C)
6. BlockStorageP requests operations on StorageManagerC with C, which maps to volume V

The two complicated parts are the mapping between key spaces. In the case of the client id and resource id, the keyspaces are used to distinguish different callers, especially for storing state. The volume keyspace, however, is a little different. It is used to calculate an offset into the storage medium. The motivation for its being a parameter of a parameterized interface is a bit different. It is more like AMSend, where the value is a constant and can be easily decoupled from the implementation. Rather than passing a volume ID into a module and forcing it to include the constant as an argument to every function call, putting it into a configuration lets nesC automatically generate code to include the constant in all calls and a dispatch table for all events.

7.5 Configurations, revisited

At first glance, configurations are rather simple: they're a way to connect components into a larger abstraction. Because nesC components are constrained to a local naming scope, a module cannot introduce any inter-component dependencies. It can introduce a dependency on a particular interface, but not on an implementation of that interface, or even name how that implementation might be found. This is very different

from C-based systems languages, whose global naming scope forces levels of indirection that can only be resolved at runtime.

A configuration can name other components and their interfaces, but in and of itself does not have any code. This means that many abstractions are broken into two parts, a module and a configuration. For example, a module that sends packets with AMSend doesn't introduce a dependency on AMSender, but a configuration that wires AMSend to an AMSender does.

Chapter 8

Design Patterns

To quote the Gang of Four, design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." As mentioned in Chapter 3, many (but certainly not all) design patterns in C-like languages deal with the problem of flexibly composing software when you have a single global namespace for functions.

This chapter presents eight nesC design patterns: three behavioural (relating to component interaction): Dispatcher, Decorator and Adapter, three structural (relating to how applications are structured): Service Instance, Placeholder and Facade and two namespace (management of identifiers such as message types): Keyset and Keymap. Each pattern's presentation follows the model of the Design Patterns book. Each one has an *Intent*, which briefly describes its purpose. A more in-depth *Motivation* follows, providing an example drawn from TinyOS. *Applicable When* provides a succinct list of conditions for use and a component diagram shows the *Structure* of how components in the pattern interact.[1] In the component diagrams, solid rectangles are modules and open rectangles are configurations. Triangles pointing into a rectangle are provided interfaces, triangles pointing out are used interfaces. Lines are wires, and component names are in bold. A folded sub-box contains source code (a floating folded box represents source code in some other, unnamed, component). The diagram is followed by a *Participants* lists explaining the role of each component. *Sample Code* shows an example nesC implementation in TinyOS 1.x, TinyOS 2.x, or both, and *Known Uses* points to some uses of the pattern in TinyOS. *Consequences* describes how the pattern achieves its goals, and notes issues to consider when using it. Finally, *Related Patterns* compares to other relevant patterns.

[1] This diagram is omitted for the Keyset pattern as it is not concerned with component interactions.

8.1 Behavioural: Dispatcher

Intent

Dynamically select between a set of operations based on an identifier. Provides a way to easily extend or modify a system by adding or changing operations.

Motivation

At a high level, sensor network applications execute operations in response to environmental input such as sensor readings or network packets. The operation's details are not important to the component that presents the input. We need to be able to easily extend and modify what inputs an application cares about, as well as the operation associated with an input.

For example, a node can receive many kinds of active messages (packets). Active messages (AM) have an 8-bit type field, to distinguish between protocols. A flooding protocol uses one AM type, while an ad-hoc routing protocol uses another. `AMStandard`, the component that signals the arrival of a packet, should not need to know what processing a protocol performs or whether an application supports a protocol. `AMStandard` just delivers packets, and higher level communication services respond to those they care about.

The traditional approach to this problem is to use function pointers or objects, which are dynamically registered as callbacks. In many cases, even though registered at run time, the set of operations is known at compile time. Thus these callbacks can be replaced by a dispatch table compiled into the executable, with two benefits. First, this allows better cross-function analysis and optimization, and secondly it conserves RAM, as no pointers or callback structures need to be stored.

Such a dispatch table could be built for the active message example by using a `switch` statement in `AMStandard`. But this is very inflexible: any change to the protocols used in an application requires a change in a system component.

A better approach in TinyOS is to use the Dispatcher pattern. A Dispatcher invokes operations using a parameterised interface, based on a data identifier. In the case of `AMStandard`, the interface is `ReceiveMsg` and the identifier is the active message type field. `AMStandard` is independent of what messages the application handles, or what processing those handlers perform. Adding a new handler requires a single wiring to `AMStandard`. If an application does not wire a receive handler for a certain type, `AMStandard` defaults to a null operation.

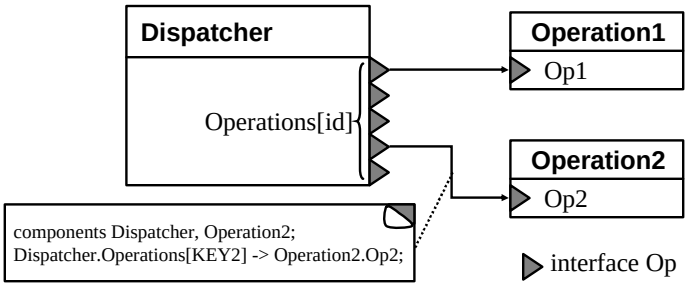
Another example of a Dispatcher is the scheduler of the Maté virtual machine. Each instruction is a separate component that provides the `MateBytecode` interface. The scheduler executes a particular

bytecode by dispatching to the instruction component using a parameterised `MateBytecode` interface. The instruction set can be easily changed by altering the wiring of the scheduler.

Applicable When

- A component needs to support an externally customisable set of operations.
- A primitive integer type can identify which operation to perform.
- The operations can all be implemented in terms of a single interface.

Structure



Participants

- **Dispatcher**: invokes its parameterized interface based on an integer type.
- **Operation**: implements the desired functionality and wires it to the dispatcher.

1.x Sample Code

`AMStandard` is the radio stack component that dispatches received messages:

```
module AMStandard {
  // Dispatcher interface for messages
  uses interface ReceiveMsg as Recv[uint8_t id];
}
implementation {
  TOS_MsgPtr received(TOS_MsgPtr packet) {
    return signal Recv.receive[packet->type](packet);
  }
  ...
}
```

Listing 8.1: AMStandard

and the `App` configuration registers `AppM` to handle two kinds of messages:

```
configuration App {}
implementation {
  components AppM, AMStandard;
  AppM.ClearIdMsg -> AMStandard.Receive[AM_CLEARIDMSG];
  AppM.SetIdMsg -> AMStandard.Receive[AM_SETIDMSG];
}
```

Listing 8.2: Wiring to AMStandard

2.x Sample Code

Dispatchers in TinyOS 2.0 are very similar to those in 1.x, except that often they are encapsulated in a generic configuration. Rather than have a component wire to a parameterized interface, it wires to a generic configuration that takes the parameter is an argument. For example, rather than wire directly to ActiveMessageC to receive packets, a component instantiates an AMReceiverC:

```
generic configuration AMReceiverC(am_id_t amId) {
  provides {
    interface Receive;
    interface Packet;
    interface AMPacket;
  }
}

implementation {
  components ActiveMessageImplP as Impl;

  Receive = Impl.Receive[amId];
  Packet = Impl;
  AMPacket = Impl;
}
```

Listing 8.3: AMReceiverC

Known Uses

The Active Messages networking layer (`AMStandard`, `AMPromiscuous` in 1.x, ActiveMessageC in 2.0) uses a dispatcher for packet reception. It also provides a parameterized packet sending interface, so services can easily match packet sends to reception handlers.

The 1.x Maté virtual machine uses a dispatcher to allow easy customization of instruction sets.

The 1.x Drip management protocol uses a Dispatcher to allow per-application configuration of management attributes.

Consequences

By leaving operation selection to nesC wirings, the dispatcher's implementation remains independent of what an application supports. However, finding the full set of supported operations can require looking

at many files. Sloppy operation identifier management can lead to dispatch problems. If two operations are wired with the same identifier, then a dispatch will call both, which may lead to resource conflicts, data corruption, or memory leaks from lost pointers. For example, the `ReceiveMsg` interface uses a buffer swap mechanism to pass buffers between the radio stack and network services, in which the higher component passes a new buffer in the return value of the event. If two services are wired to a given `ReceiveMsg` instance, only one of their pointers will be passed and the second will be lost. Wiring in this fashion is a compile-time warning in nesC, but it is still a common bug for novice TinyOS developers.

The key aspects of the dispatcher pattern are:

- It allows you to easily extend or modify the functionality an application supports: adding an operation requires a single wiring.
- It allows the elements of functionality to be independently implemented and re-used. Because each operation is implemented in a component, it can be easily included in many applications. Keeping implementations separate can also simplify testing, as the components will be smaller, simpler, and easier to pinpoint faults in. The nesC compiler will automatically inline small operations, or you can explicitly request inlining; thus this decomposition has no performance cost.
- It requires the individual operations to follow a uniform interface. The dispatcher is usually not well suited to operations that have a wide range of semantics. As all implementations have to meet the same interface, broad semantics leads to the interface being overly general, pushing error checks from compile-time to run-time. An implementor forgetting a run-time parameter check can cause a hard to diagnose system failure.

The compile-time binding of the operation simplifies program analysis and puts dispatch tables in the compiled code, saving RAM. Dispatching provides a simple way to develop programs that execute in reaction to their environment.

Related Patterns

- Service Instance: a service instance creates many instances of an implementation of an interface, while a dispatcher selects between different implementations of an interface.
- Placeholder: a placeholder allows an application to select an implementation at compile-time, while a dispatcher allows it to select an implementation at runtime.
- Keyset: the identifiers used to identify a Dispatcher's operation typically form a Global Keyset.

8.2 Structural: Service Instance

Intent

Allows multiple users to have separate instances of a particular service, where the instances can collaborate efficiently. The basic mechanism for virtualizing services.

Motivation

Sometimes many components or subsystems need to use a system abstraction, but each user wants a separate instance of that service. We don't know how many users there will be until we build a complete application. Each instance requires maintaining some state, and the service implementation needs to access all of this state to make decisions.

For example, a wide range of TinyOS components need timers, for everything from network timeouts to sensor sampling. Each timer appears independent, but they all operate on top of a single hardware clock. An efficient implementation thus requires knowing the state of all of the timers. If the implementation can easily determine which timer has to fire next, then it can schedule the underlying clock resource to fire as few interrupts as possible to meet this lowest timer's requirement. Firing fewer interrupts allows the CPU to sleep more, saving energy and increasing lifetime.

The traditional object-oriented approach to this problem is to instantiate an object representing the service and use another class to coordinate state. This approach is not applicable in TinyOS 1.x as nesC 1.1 cannot have multiple copies of components, so either requires requires sharing state across objects, which is contrary to encapsulation, or it requires state copying, which uses additional RAM.

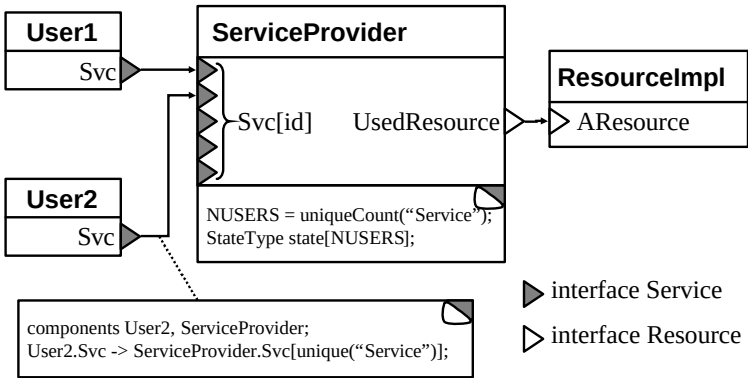
Implementing each timer in a separate module leads to duplicated code and requires inter-module coordination in order to figure out how to set the underlying hardware clock. Just setting it at a fixed rate and maintaining a counter for each Timer is inefficient: timer fidelity requires firing at a high rate, but it wastes energy to fire at 1KHz if the next timer is in four seconds.

The Service Instance pattern provides a solution to these problems. Using this pattern, each user of a service can have its own (virtual) instance, but instances share code and can access each other's state. A component following the Service Instance pattern provides its service in a parameterised interface; each user wires to a unique instance of the interface using unique. The underlying component receives the unique identity of each client in each command, and can use it to index into a state array. The component can determine at compile-time how many instances exist using the uniqueCount function and dimension the state array accordingly.

Applicable When

- A component needs to provide multiple instances of a service, but does not know how many until compile time.
- Each service instance appears to its user to be independent of the others.
- The service implementation needs to be able to easily access the state of every instance.

Structure



Participants

- **ServiceProvider**: allocates state for each instance of the service and coordinates underlying resources based on all of the instances.
- **ResourceImpl**: an underlying system resource that ServiceProvider multiplexes/demultiplexes service instances on.

Sample 1.x Code

`TimerC` wires `TimerM`, which contains the actual timer logic, to an underlying hardware clock and exports its `Timer` interfaces:

```
configuration TimerC {
  provides interface Timer[uint8_t id];
}
implementation {
  components TimerM, ClockC;

  Timer = TimerM.Timer;
  TimerM.Clock -> ClockC.Clock;
}
```

Listing 8.4: TimerM builds on top of ClockC

and `TimerM` uses uniqueCount to determine how many timers to allocate and accesses them using unique IDs:

```
module TimerM {
  provides interface Timer[uint8_t clientId];
  uses interface Clock;
}
implementation {
  // per-client state
  timer_t timers[uniqueCount("Timer")];

  command result_t Timer.start[uint8_t clientId](...) {
    if (timers[clientId].busy)
      ...
  }
}
```

Listing 8.5: TimerM uses uniqueCount to count the timers

Clients wanting a timer wire using unique:

```
  C.Timer -> TimerC.Timer[unique("Timer")];
```

Listing 8.6: Wiring a timer with unique

Known Uses

The TinyOS 1.x and 2.x timer systems, as detailed above, uses a service instance pattern to manage various application timers.

The viral code propagation subsystem of the Maté virtual machine uses a service instance to manage version metadata for code capsules. As the virtual machine is customizable, the number of needed capsules isn't known until the virtual machine is actually compiled.

In a similar vein, the epidemic dissemination protocol Drip uses the service instance pattern to maintain epidemic state for each disseminated value.

Sample 2.x Code

HilTimerMilliC uses a VirtualizeTimerC to present a Service Instance of millisecond precision timers. VirtualizeTimerC takes a single underlying timer and virtualizes it to n timers for others to use:

```
generic module VirtualizeTimerC( typedef precision_tag, int max_timers ) {
  provides interface Timer<precision_tag> as Timer[ uint8_t num ];
  uses interface Timer<precision_tag> as TimerFrom;
}
```

Listing 8.7: VirtualizeTimerC

It takes two parameters, a type for the Timer precision tag, and the number of timers (the number of service instances). There are therefore four files involved:

1. Timer.h, a header file that defines the unique key as UQ_TIMER_MILLI
2. VirtualizeTimerC, which provides n virtualized timers
3. HilTimerMilliC, which instantiates a VirtualizeTimerC with a call to uniqueCount passing UQ_TIMER_MILLI
4. TimerMilliC, which wires to HilTimerMilliC's parameterized interface with a call to unique passing UQ_TIMER_MILLI

Consequences

The key aspects of the Service Instance pattern are:

- It allows many components to request independent instances of a common system service: adding an instance requires a single wiring.
- It controls state allocation, so the amount of RAM used is scaled to exactly the number of instances needed, conserving memory while preventing run-time failures due to many requests exhausting resources.
- It allows a single component to coordinate all of the instances, which enables efficient resource management and coordination.

Because the pattern scales to a variable number of instances, the cost of its operations may scale linearly with the number of users. For example, if setting the underlying clock interrupt rate depends on the timer with the shortest remaining duration, an implementation might determine this by scanning all of the timers, an O(n) operation.

If many users require an instance of a service, but each of those instances are used rarely, then allocating state for each one can be wasteful. The other option is to allocate a smaller amount of state and dynamically allocate it to users as need be. This can conserve RAM, but requires more RAM per real instance (client IDs need to be maintained), imposes a CPU overhead (allocation and deallocation), can fail at run-time (if there are too many simultaneous users), and assumes a reclamation strategy (misuse of which would lead to leaks). This long list of challenges makes the Service Instance an attractive – and more and more commonly used – way to efficiently support application requirements. There are situations, however, when a component internally re-uses a single service instance for several purposes: for example, the Maté code

propagation component `MVirus` uses a single timer instance for several different timers which never operate concurrently.

Related Patterns

- Dispatcher: a service instance creates many instances of an implementation of an interface, while a dispatcher selects between different implementations of an interface.
- Keyset: a Service Instance's instance identifiers form a Local Keyset.

8.3 Namespace: Keysets

Intent

Provide namespaces for referring to protocols, structures, or other entities in a program.

Motivation

A typical sensor network program needs namespaces for the various entities it manages, such as protocols, data types, or structure instances. Limited resources mean names are usually stored as small integer keys.

For data types representing internal program structures, each instance must have a unique name, but as they are only relevant to a single mote, the names can be chosen freely. These *local* namespaces are usually dense, for efficiency. The Service Instance pattern (Section 8.2) uses a local namespace to identify instances. In contrast, communication requires a shared, *global* namespace: two motes/applications must agree on an element's name. As a mote may only use a few elements, global namespaces are typically sparse. The Dispatcher pattern (Section 8.1) uses a global namespace to select operations.

The Keyset patterns provide solutions to these problems. Using these patterns, programs can refer to elements using identifiers optimised for their particular use. Components using the Keyset patterns often take advantage of a parameterised interface, in which the parameter is an element in a Keyset. Local Keysets are designed for referring to local data structures (e.g., arrays) and are generated with `unique`; Global Keysets are designed for communication and use global constants.

The bytecodes of the Maté virtual machine form a Global Keyset. The Maté scheduler uses these in conjunction with a Dispatcher to execute individual instructions, each of which is implemented in a separate component. Maté also uses a Local Keyset to identify locks corresponding to resources used by Maté programs. These lock identifiers are allocated with `unique` as the Maté virtual machine can be compiled with varying sets of resources.

The file descriptors of the Matchbox flash filesystem form a Local Keyset.

Applicable When

- A program must keep track of a set of elements or data types.
- The set is known and fixed at compile-time.

Sample 1.x Code

The Maté bytecodes are defined as global constants:

```
typedef enum {
  OP_HALT = 0x0,
  OP_MADD =     0x1,
  OP_MBA3 =     0x2,
  OP_MBF3 =     0xa,
  ...
} MateInstruction;
```

Listing 8.8: Mate VM bytecodes

and used by the Maté scheduler to execute individual instructions:

```
module MateEngineM {
  uses interface MateBytecode[uint8_t bytecode];
  ...
}
implementation {
  void computeInstruction(MateContext* context) {
    MateOpcode instr = getOpcode(context);
    context->pc += call MateBytecode.byteLength[instr]();
    call MateBytecode.execute[instr](context);
  }
  ...
}
```

Listing 8.9: Mate VM execution loop

The Maté lock subsystem identifies locks by small integers:

```
module MLocks {
  provides interface MateLocks as Locks;
}
implementation {
  MateLock locks[MATE_LOCK_COUNT];

  command void Locks.lock(MateContext* uint8_t lockNum) {
    locks[lockNum].holder = context;
    context->heldSet[lockNum / 8] |= 1 << (lockNum % 8);
```

```
  }
  ...
```

Listing 8.10: Mate lock subsystem

Locks are allocated in components providing shared resources:

```
module OPgetsetvar1M { ... } // a shared variable
implementation {
  typedef enum {
    MATE_LOCK_1_0 = unique("MateLock"),
    MATE_LOCK_1_1 = unique("MateLock"),
  } LockNames;
  ...

module OPbpush1M { ... } // a shared buffer
implementation {
  typedef enum {
    MATE_BUF_LOCK_1_0 = unique("MateLock"),
    MATE_BUF_LOCK_1_1 = unique("MateLock"),
  } BufLockNames;
  ...
```

Listing 8.11: Mate lock allocation

and `uniqueCount` is used to find the total number of locks:

```
enum {
  MATE_LOCK_COUNT = uniqueCount("MateLock")
};
```

Listing 8.12: Using uniqueCount to count the Mate locks

Sample 2.x Code

Keysets in 2.x operate exactly as they do in 1.x, with the exception that sometimes they are encapsulated in generic configurations. Active message IDs, for example, are global constants, while queue entries in the AM send queue are local indentifiers.

Known Uses

Many components use Local Keysets: they are a fundamental part of the Service Instance pattern. See for example the timer service, `TimerC`, or the Matchbox flash file system.

Maté uses a Local Keyset to keep track of Maté shared resource locks (see above).

Active Messages (`AMStandard`) uses a Global Keyset for Active Message types.

The Drip management protocol uses a Global Keyset for referring to configurable variables.

The TinyDB sensor-network-as-database application uses a Global Keyset for its attributes; in this case,

however, the keyset is composed of strings, which are then mapped to a Local Keyset using a table.

Consequences

Keysets allow a component to refer to data items or types through a parameterised interface. In a Local Keyset, `unique` ensures that every element has a unique identifier. Global Keysets can also have unique identifiers, but this requires external namespace management.

As Local Keysets are generated with `unique`, mapping names to keys (e.g., for debugging purposes) is not obvious. The nesC constant generator, `ncg`, can be used to extract this information.

Keysets are rarely used in isolation; they support other patterns such as Dispatcher and Service Instance.

Related Patterns

- Keymap: two Keysets are often related, e.g., one Service Instance may be built on top of another, requiring a mapping between two Keysets. The Keymap pattern provides an efficient way of implementing such maps.
- Service Instance: the identifiers used to identify individual services form a Local Keyset.
- Dispatcher: the identifiers used by a dispatcher are typically taken from a Global Keyset.

8.4 Namespace: Keymap

Intent

Map keys from one keyset to another. Allows you to translate global, shared names to local, optimized names, or to efficiently subset another keyset.

Motivation

Mapping between namespaces is often useful: it allows motes to use a global, sparse namespace for easy cross-application communication and an internal, compact namespace for efficiency.

The Drip management protocol uses the Keyset and Keymap patterns to allow a user to configure parameters at run-time. A component registers a parameter with the `DripC` component with a Global Keyset, so it can be named in an application-independent manner. The user modifies a parameter by sending a key-value pair using an epidemic protocol, which distributes the change to every mote. `DripC` maintains state for each configurable parameter with the Service Instance pattern, using a Local Keyset. A Keymap maps the global key to the local key.

Keymaps are also useful for mapping between two local keysets, when some service, based on the Service Instance pattern, accesses a subset of the resources provided by another service, also based on the

Service Instance pattern.

For instance, in TinyOS 2.0, there are three storage abstractions with different interfaces: blocks, logs and configuration data. Each of these is identified by keys from its own local keyset. However, all three are built upon a common volume abstraction (provided by the StorageManagerC component) used to partition a mote's flash chip into independent areas. The volumes used by an application are identified by a local keyset. Thus it is necessary to map a block's identifier to its corresponding volume identifier.

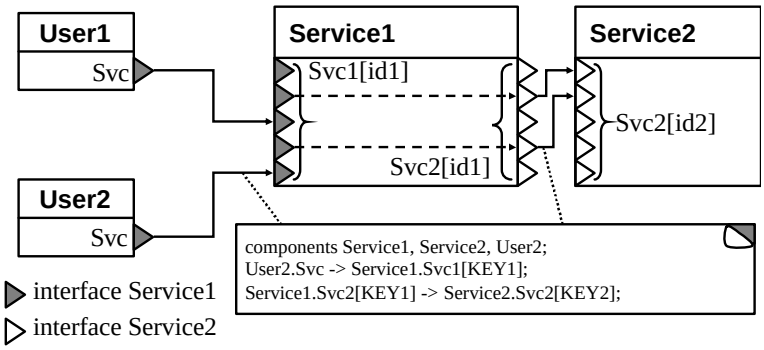
Maps could be implemented using a table and some lookup code. However, this has several problems. If we want to store this table in ROM, then it must be initialised in one place. However, this conflicts with the desire to specify keys in separate components (either with `unique` or with constants). If the table is stored in RAM, then keys can be specified in separate components, but RAM is in very short supply on motes. Finally, keys of Global Keysets are sparse, so the resulting tables would be large and waste space.

Instead, we can use nesC's wiring to build Keymaps. By mapping a parameterised interface indexed with one key to another parameterised interface indexed by a second key, we can have the nesC compiler generate the map at compile-time. Additionally, as the map exists as an automatically generated switch statement, it uses no RAM.

Applicable When

- An application uses global identifiers for communication (or other purposes) and wishes to map them to local identifiers for efficiency.
- Two services are implemented following the Service Instance pattern, and the first service needs an instance of the second service for each of its own instances.
- The identifiers are integer constants.
- The map is known at compile-time.

Structure



Participants

- **Service1**: service accessed via key 1, dependent on Service2.
- **Service2**: service accessed via key 2.

Sample 1.x Code

The `DripC` component provides a parameterised interface for components to register configurable values with a Global Keyset:

```
enum { DRIP_GLOBAL = 0x20};
App.Drip -> DripC.Drip[DRIP_GLOBAL];
```

Listing 8.13: A Drip global ID

`DripC` uses another component to manage its internal state, `DripStateM`. `DripStateM` uses a Local Keyset for the configurable values (an example of the Service Instance pattern, in Section 8.2), and a Keymap maps between the two:

```
enum { DRIP_LOCAL = unique("DripState")};
DripC.DripState[DRIP_GLOBAL] -> DripStateM.DripState[DRIP_LOCAL];
```

Listing 8.14: A Drip local ID

In this example, a user can generate a new value for `App`'s parameter, and distribute it based on the `DRIP_GLOBAL` key. `DripC` uses the global key to refer to the value, but `DripStateM` can use a local key to refer to the state it maintains for that value. The wiring compiles down to a simple switch statement that calls `DripStateM` with the proper local key.

The `BlockStorageC` component follows the Service Instance pattern to provide access to blocks, and the `StorageManagerC` uses the same pattern to provide access to volumes:

```
configuration BlockStorageC {
  provides interface Block[int blockId];
} ...
configuration StorageManagerC {
  provides interface Volume[int volumeId];
} ...
```

Listing 8.15: BlockStorageC uses the Service Instance Pattern

To use a block, you need to allocate unique block and volume identifiers and wire `BlockStorageC` to `StorageManagerC` to form the Keymap:

```
enum {
  MY_BLOCK = unique("Block"),
  MY_VOLUME = unique("Volume")
};

configuration MyBlock {
  provides interface Block;
}
implementation {
  components BlockStorageC, StorageManagerC;

  Block = BlockStorageC.Block[MY_BLOCK];
  BlockStorageC.Volume[MY_BLOCK] -> StorageManagerC.Volume[MY_VOLUME];
}
```

Listing 8.16: Allocating a block storage instance

Sample 2.x Code

Keymaps operate in TinyOS 2.0 just as they do in 1.x.

Known Uses

The Drip parameter configuration component – described above – uses a Keymap.

The TinyOS 2.0 storage system – also described above – uses a Keymap to map the different storage abstractions to the common volume abstraction.

Consequences

A Keymap uses nesC wiring to allow components to transparently map between different keysets. As with Keysets, the Keymap must be fixed at compile-time.

A Keymap translates into a `switch` at compile-time. It thus doesn't use any RAM; its speed depends on the behaviour of the C compiler used to compile nesC's output.

Keymaps only support mapping between integers. If you need, e.g., to map from strings to a Local Keyset, you will need to build your own map.

Related Patterns

- Keyset: A Keymap establishes a map from one keyset to another.

8.5 Structural: Placeholder

Intent

Easily change which implementation of a service an entire application uses. Prevent inadvertent inclusion of multiple, incompatible implementations.

Motivation

Many TinyOS systems and abstractions have several implementations. For example, there are many ad-hoc tree routing protocols (Route, MintRoute, ReliableRoute), but they all expose the same interface, `Send`. The standardized interface allows applications to use any of the implementations without code changes. Simpler abstractions can also have multiple implementations. For example, the `LedsC` component actually turns the LEDs on and off, while the `NoLedsC` component, which provides the same interface, has null operations. During testing, `LedsC` is useful for debugging, but in deployment it is a significant energy cost and usually replaced with `NoLedsC`.

Sometimes, the decision of which implementation to use needs to be uniform across an application. For example, if a network health monitoring subsystem (`HealthC`) wires to MintRoute, while an application uses ReliableRoute, two routing trees will be built, wasting resources. As every configuration that wires to a service names it, changing the choice of implementation in a large application could require changing many files. Some of these files, such as `HealthC`, are part of the system; an application writer should not have to modify them.

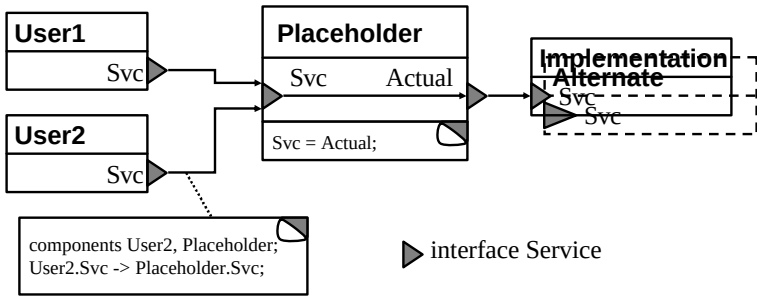
One option is for every implementation to use the same component name, and put them in separate directories. Manipulating the nesC search order allows an application to select which version to use. This approach doesn't scale well: each implementation of each component needs a separate directory. Streamlining this structure by bundling several implementations (e.g., the "safe" versions and the "optimized" ones) in a single directory requires all-or-nothing inclusion. This approach also precludes the possibility of including two implementations, even if they can interoperate.

The Placeholder pattern offers a solution. A placeholder configuration represents the desired service through a level of naming indirection. All components that need to use the service wire to the placeholder. The placeholder itself is just "a pass through" of the service's interfaces. A second configuration (typically provided by the application) wires the placeholder to the selected implementation. This selection can then be changed centrally by editing a single file. As the level of indirection is solely in terms of names – there is no additional code generated – it imposes no CPU overhead.

Applicable When

- A component or service has multiple, mutually exclusive implementations.
- Many subsystems and parts of your application need to use this component/service.
- You need to easily switch between the implementations.

Structure



Participants

- **Placeholder**: the component that all other components wire to. It encapsulates the implementation and exports its interfaces with pass-through wiring. It has the same signature as the Implementation component.

- **Implementation**: the specific version of the component.

Sample 1.x Code

Several parts of an application use ad-hoc collection routing to collect and aggregate sensor readings. However, the application design is independent of a particular routing implementation, so that improvements or new algorithms can be easily incorporated. The routing subsystem is represented by a Placeholder, which provides a unified name for the underlying implementation and just exports its interfaces:

```
configuration CollectionRouter {
  provides interface StdControl as SC;
  uses     interface StdControl as ActualSC;
  provides interface SendMsg as Send;
  uses     interface SendMsg as ActualSend;
}
implementation {
  SC = ActualSC;     // Just "forward" the
  Send = ActualSend; // interfaces
}
```

Listing 8.17: CollectionRouter

Component using collection routing wire to CollectionRouter:

```
  SensingM.Send -> CollectionRouter.Send;
```

Listing 8.18: Wiring to CollectionRouter

and the application must globally select its routing component by wiring the "Actual" interfaces of the Placeholder to the desired component:

```
configuration AppMain { }
implementation {
  components CollectionRouter, EWMARouter;

  CollectionRouter.ActualSC -> EWMARouter.SC;
  CollectionRouter.ActualSend -> EWMARouter.Send;
  ...
}
```

Listing 8.19: Wiring CollectionRouter to an implementation

Sample 2.x Code

The Telos platform uses a Placeholder to map ActiveMessageC to its radio stack, CC2420ActiveMessageC:

```
configuration ActiveMessageC {
  provides {
    interface Init;
    interface SplitControl;

    interface AMSend[uint8_t id];
    interface Receive[uint8_t id];
    interface Receive as Snoop[uint8_t id];

    interface Packet;
    interface AMPacket;
    interface PacketAcknowledgements;
  }
}
implementation {
  components CC2420ActiveMessageC as AM;

  Init         = AM;
  SplitControl = AM;

  AMSend       = AM;
  Receive      = AM.Receive;
  Snoop        = AM.Snoop;
  Packet       = AM;
  AMPacket     = AM;
  PacketAcknowledgements = AM;
}
```

Listing 8.20: Telos ActiveMessageC

Known Uses

The Maté virtual machine uses Placeholders for all its major abstractions (stacks, type checking, locks,

etc). The motlle language replaces the Maté stack handler because it uses a different value representation – the replacement stack handler converts between the motlle and Maté value representations.

HIL abstractions in TinyOS 2.0 (e.g., ActiveMessageC) are often built with placeholders. This allows a platform to map the global name to a particular implementation that it supports. In the case of ActiveMessageC, for example, this can be important if a platform has two radios and one should be used as the default.

Consequences

The key aspects of the Placeholder pattern are:

- Establishes a global name that users of a common service can wire to.
- Allows you to specify the implementation of the service on an application-wide basis.
- Does not require every component to use the Placeholder's implementation.

By adding a level of naming indirection, a Placeholder provides a single point at which you can choose an implementation. Placeholders create a global namespace for implementation-independent users of common system services. As using the Placeholder pattern generally requires every component to wire to the Placeholder instead of a concrete instance, incorporating a Placeholder into an existing application can require modifying many components. However, the nesC compiler optimises away the added level of wiring indirection, so a Placeholder imposes no run-time overhead. The Placeholder supports flexible composition and simplifies use of alternative service implementations.

Related Patterns

- Dispatcher: a placeholder allows an application to select an implementation at compile-time, while a dispatcher allows it to select an implementation at runtime.
- Facade: a placeholder allows easy selection of the implementation of a group of interfaces, while a facade allows easy use of a group of interfaces. An application may well connect a placeholder to a facade.

8.6 Structural: Facade

Intent

Provides a unified access point to a set of inter-related services and interfaces. Simplifies use, inclusion, and composition of the subservices.

Motivation

Complex system components, such as a filesystem or networking abstraction, are often implemented across many components. Higher-level operations may be based on lower-level ones, and a user needs access to both. Complex functionality may be spread across several components. Although implemented separately, these pieces of functionality are part of a cohesive whole that we want to present as a logical unit.

For example, the Matchbox filing system provides interfaces for reading and writing files, as well as for metadata operations such as deleting and renaming. Separate modules implement each of the interfaces, depending on common underlying services such as reading blocks.

One option would be to put all of the operations in a single, shared interface. This raises two problems. First, the nesC wiring rules mean that a component that wants to use *any* command in the interface has to handle *all* of its events. In the case of a file system, all the operations are split-phase; having to handle a half dozen events (`readDone`, `writeDone`, `openDone`, etc.) merely to be able to delete a file is hardly usable. Second, the implementation cannot be easily decomposed into separate components without introducing internal interfaces, as the top-level component will need to call out into the subcomponents. Implementing the entire subsystem as a single huge component is not easy to maintain.

Another option is to export each interface in a separate component (e.g., MatchboxRead, MatchboxWrite, MatchboxRename, etc.). This increases wiring complexity, making the abstraction more difficult to use. For a simple open, read, and write sequence, the application would have to wire to three different components. Additionally, each interface would need a separate configuration to wire it to the subsystems it depends on, increasing clutter in the component namespace. The implementer needs to be careful with these configurations, to prevent inadvertent double-wirings.

The Facade pattern provides a better solution to this problem. The Facade pattern provides a uniform access point to interfaces provided by many components. A Facade is a nesC configuration that defines a coherent abstraction boundary by exporting the interfaces of several underlying components. Additionally, the Facade can wire the underlying components, simplifying dependency resolution.

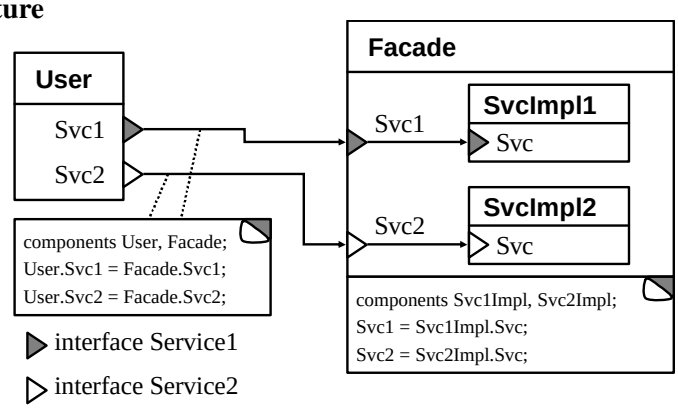
A nesC Facade has strong resemblances to the object oriented pattern of the same name. The distinction lies in nesC's static model. An object-oriented Facade instantiates its subcomponents at run-time, storing pointers and resolving operations through another level of call indirection. In contrast, as a nesC Facade is defined through naming (pass through wiring) at compile time, there is no run time cost.

Applicable When

- An abstraction, or series of related abstractions, is implemented across several separate components.

- It is preferable to present the abstraction in whole rather than in parts.

Structure



Participants

- **Facade**: the uniform presentation of a set of related services.
- **SvcImpl**: the separate implementations of each service composing the Facade.

Sample 1.x Code

The Matchbox filing system uses a Facade to present a uniform filesystem abstraction. File operations are all implemented in different components, but the top-level Matchbox configuration provides them in a single place. Each of these components depends on a wide range of underlying abstractions, such as a block interface to non-volatile storage; `Matchbox` wires them appropriately, resolving all of the dependencies.

```
configuration Matchbox {
  provides {
    interface FileRead[uint8_t fd];
    interface FileWrite[uint8_t fd];
    interface FileDir;
    interface FileRename;
    interface FileDelete;
  }
}
implementation {
  // File operation implementations
  components Read, Write, Dir, Rename, Delete;

  FileRead = Read.FileRead;
  FileWrite = Write.FileWrite;
  FileDir = Dir.FileDir;
  FileRename = Rename.FileRename;
  FileDelete = Delete.FileDelete;
  // Wiring of operations to sub-services omitted
}
```

Listing 8.21: The Matchbox facade

Sample 2.x Code

The CC2420 stack is broken up into three call paths: control, transmission, and reception. The top-level CC2420CsmaC component presents these three paths together as a single abstraction using a Facade:

```
configuration CC2420CsmaC {

  provides interface Init;
  provides interface SplitControl;

  provides interface Send;
  provides interface Receive;
  provides interface PacketAcknowledgements as Acks;

  uses interface AMPacket;

}

implementation {

  components CC2420CsmaP as CsmaP;

  Init = CsmaP;
  SplitControl = CsmaP;
  Send = CsmaP;
  Acks = CsmaP;
  AMPacket = CsmaP;

  components CC2420ControlC;
  Init = CC2420ControlC;
  AMPacket = CC2420ControlC;
  CsmaP.Resource -> CC2420ControlC;
  CsmaP.CC2420Config -> CC2420ControlC;

  components CC2420TransmitC;
  Init = CC2420TransmitC;
  CsmaP.SubControl -> CC2420TransmitC;
  CsmaP.CC2420Transmit -> CC2420TransmitC;
  CsmaP.CsmaBackoff -> CC2420TransmitC;

  components CC2420ReceiveC;
  Init = CC2420ReceiveC;
  Receive = CC2420ReceiveC;
  CsmaP.SubControl -> CC2420ReceiveC;

  components RandomC;
  CsmaP.Random -> RandomC;

  components LedsC as Leds;
  CsmaP.Leds -> Leds;

}
```

Listing 8.22: The CC2420CsmaC uses a Facade

Known Uses

Stable, commonly used abstract boundaries such as radio stacks (CC2420CsmaC, CC1000CsmaRadioC) and storage (BlockStorageC) often use a Facade. This allows them to implement complex abstractions in smaller, distinct parts, which simplifies code.

Consequences

The key aspects of the Facade pattern are:

- Provides an abstraction boundary as a set of interfaces. A user can easily see the set of operations the abstraction support, and only needs to include a single component to use the whole service.
- Presents the interfaces separately. A user can wire to only the needed parts of the abstraction, but be certain everything underneath is composed correctly.

A Facade is not always without cost. Because the Facade names all of its sub-parts, they will all be included in the application. While the nesC compiler attempts to remove unreachable code, this analysis is necessarily conservative and may end up keeping much useless code. In particular, unused interrupt handlers are never removed, so all the code reachable from them will be included every time the Facade is used. If you expect applications to only use a very narrow part of an abstraction, then a Facade can be wasteful.

Related Patterns

- Placeholder: a placeholder allows easy selection of the implementation of a group of interfaces, while a facade allows easy use of a group of interfaces. An application may well connect a placeholder to a facade.

8.7 Behavioural: Decorator

Intent

Enhance or modify a component's capabilities without modifying its implementation. Be able to apply these changes to any component that provides the interface.

Motivation

We often need to add extra functionality to an existing component, or to modify the way it works without changing its interfaces. For instance, the standard `ByteEEPROM` component provides a `LogData` interface

to log data to a region of flash memory. In some circumstances, we would like to introduce a RAM write buffer on top of the interface. This would reduce the number of actual writes to the EEPROM, conserving energy (writes to EEPROM are expensive) and the lifetime of the medium.

Adding a buffer to the `ByteEEPROM` component forces all logging applications to allocate the buffer. As some application may not able to spare the RAM, this is undesirable. Providing two versions, buffered and unbuffered, replicates code, reducing reuse and increasing the possibility of incomplete bug fixes. It is possible that several implementers of the interface – any component that provides `LogData` – may benefit from the added functionality. Having multiple copies of the buffering version, spread across several services, further replicates code.

There are two traditional object-oriented approaches to this problem: inheritance, which defines the relationship at compile time through a class hierarchy, and decorators, which define the relationship at run time through encapsulation. As nesC is not an object-oriented language, and has no notion of inheritance, the former option is not possible. Similarly, run-time encapsulation is not readily supported by nesC's static component composition model and imposes overhead in terms of pointers and call forwarding. However, we can use nesC's component composition and wiring to provide a compile time version of the Decorator.

A Decorator component is typically a module that provides and uses the same interface type, such as `LogData`. The provided interface adds functionality on top of the used interface. For example, the `BufferedLog` component sits on top of a `LogData` provider. It implements its additional functionality by aggregating several `BufferedLog` writes into a single `LogData` write.

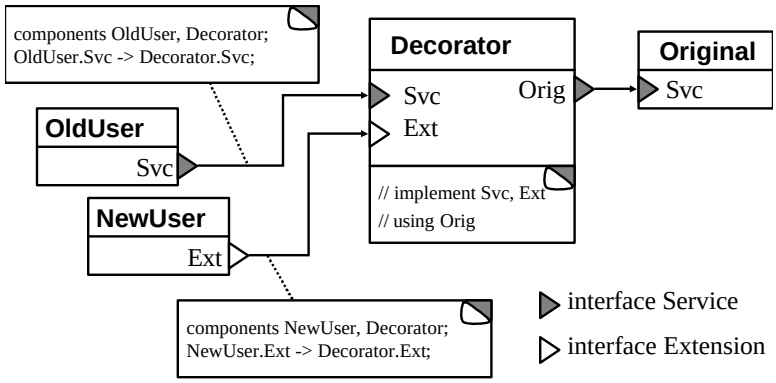
Using a Decorator can have further benefits. In addition to augmenting existing interfaces, they can introduce new ones that provide alternative abstractions. For example, `BufferedLog` provides a synchronous (not split phase) `FastLog` interface; a call to `FastLog` writes directly into the buffer.

Finally, separating added functionality into a Decorator allows it to apply to any implementation. For example, a packet send queue Decorator can be interposed on top of any networking abstraction that provides the `Send` interface; this allows flexible interpositioning of queues and queueing policies in a networking system.

Applicable When

- You wish to extend the functionality of an existing component without changing its implementation, or
- You wish to provide several variants of a component without having to implement each possible combination separately.

Structure



Participants

- **Original**: the original service.
- **Decorator**: the extra functionality added to the service.

Sample 1.x Code

The standard `LogData` interface includes split-phase `erase`, `append` and `sync` operations. `BufferedLog` adds buffering to the `LogData` operations, and, additionally, supports a `FastLogData` interface with a non-split-phase `append` operation (for small writes only):

```
module BufferedLog {
  provides interface LogData as Log;
  provides interface FastLogData as FastLog;
  uses interface LogData as UnbufferedLog;
}
implementation {
  uint8_t buffer1[BUFSIZE], buffer2[BUFSIZE];
  uint8_t *buffer;
  command result_t FastLog.append(data, n) {
    if (bufferFull()) {
      call UnbufferedLog.append(buffer, offset);
      // ... switch to other buffer ...
    }
    // ... append to buffer ...
  }
```

Listing 8.23: The BufferedLog decorator

The `SendQueue` Decorator introduces a send queue on top of a split-phase `Send` interface:

```
module SendQueue {
  provides interface Send;
  uses interface Send as SubSend;
```

```
}
implementation {
  TOS_MsgPtr queue[QUEUE_SIZE];
  uint8_t head, tail;
  command result_t Send.send(TOS_MsgPtr msg) {
    if (!queueFull()) enqueue(msg);
    if (!subSendBusy()) startSendRequest();
  }
```

Listing 8.24: The SendQueue decorator

Known Uses

`BufferedLog` improves split-phase logging interface by buffering small writes.

`CRCFilter` decorates a `ReceiveMsg` interface by filtering packets that did not pass a CRC check: packets that pass are signalled up, those that don't are not.

`QueuedSend` decorates a `SendMsg` interface by enqueuing multiple requests, which it serlialises onto an underlying `SendMsg`, providing in-order transmission.

Consequences

Applying a Decorator allows you to extend or modify a component's behaviour though a separate component: the original implementation can remain unchanged. Additionally, the Decorator can be applied to any component that provides the interface.

In most cases, a decorated component should not be used directly, as the Decorator is already handling its events. The Placeholder pattern (Section 8.5) can be used to help ensure this.

Additional interfaces are likely to use the underlying component, creating dependencies between the original and extra interfaces of a Decorator. For instance, in `BufferedLog`, `FastLog` uses `UnbufferedLog`, so concurrent requests to `FastLog` and `Log` are likely to conflict: only one can access the `UnbufferedLog` at once.

Decorators are a lightweight but flexible way to extend component functionality. Interpositioning is a common technique in building networking stacks, and Decorators enable this style of composition.

Related Patterns

- Adapter: An Adapter presents the existing functionality of a component with a different interface, rather than adding additional functionality and preserving the current interface.

8.8 Behavioural: Adapter

Intent

Convert the interface of a component into another interface, without modifying the original implementation. Allow two components with different interfaces to interoperate.

Motivation

Sometimes, a piece of functionality offered by a component with one interface needs to be accessed by another component via a different interface. For instance, the TinyDB application – which provides a database-like abstraction over a sensor network – accesses the "database attributes" of the sensor network via the `AttrRegister` interface. These attributes represent, amongst other things, the sensors attached to the sensor network's motes. However, in TinyOS, sensors are accessed via the `ADC` interface.

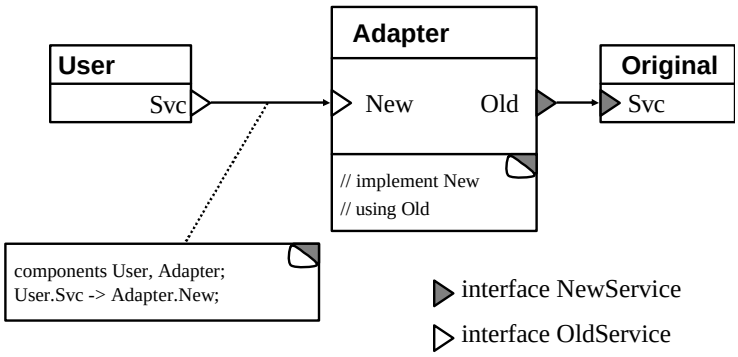
Modifying each sensor to provide an `AttrRegister` as well, or instead of, its current interfaces is not desirable, as `AttrRegister` provides functionality which is not desirable for all applications (named access to sensors) and does not provide necessary functionality (access to sensors from interrupt handlers). Instead, TinyDB uses Adapter components which implement the `AttrRegister` interface based on the functionality of the the `ADC` interface provided by sensors.

An Adapter is a component which provides an interface of type A, e.g., `AttrRegister`, and uses an interface of type B, , e.g., `ADC`, and implements the operations of A in terms of those of B. An Adapter may also need to implement functionality not provided by the B interface, e.g., `AttrRegister` needs to provide a name for the attribute. More generally, an Adapter may provide several interfaces $A_1, \ldots, A_n$ and implement them in terms of several used interfaces $B_1, \ldots, B_m$.

Applicable When

- You wish to provide the functionality of an existing component with a different interface.

Structure



Participants

- **Original**: the original service.

- **Adapter**: implements the new interface in terms of the functionality offered by the old.

Sample Code

The `AttrPhotoM` component adapts the standard `Photo` (light) sensor for use with TinyDB. It gives the attribute a name, implements getting the attribute using the underlying `ADC` interface and refuses requests to set the attribute:

```
module AttrPhotoM {
  provides interface StdControl;
  provides interface AttrRegister;
  uses interface ADC;
}
implementation {
  void *buffer;

  command void StdControl.init() {
    // register the attribute with a friendly name and size
    signal AttrRegister.registerAttr("light", 2);
  }

  command result_t AttrRegister.getAttr(void *result) {
    buffer = result;
    return call ADC.getData();
  }

  event void ADC.dataReady(uint16_t data) {
    *(uint16_t*)buffer = data;
    signal AttrRegister.getAttrDone(result);
  }

  command result_t AttrRegister.setAttr(void *attrVal) {
    // cannot "set" a sensor
    return FAIL;
  }
}
```

Listing 8.25: Adapting the ADC interface to AttrRegister

Known Uses

Many TinyDB attributes are implemented using Adapters.

In TinyOS 2.0, hardware resources such as A/D converters are presented by a hardware abstraction layer (HAL) which offers high-level, but hardware-specific interfaces and a hardware independent layer (HIL) which offers high-level, platform-independent interfaces. The HIL layer is typically an Adapter over the HAL layer. For example, see the `AdcP` A/D converter component for the ATmega128.

TinyOS 2.0 offers two timing interfaces: `Alarm`, which signals timer events immediately (within an interrupt handler), and `Timer` which signals its events in a task (with some delay). The `AlarmToTimerC`

component is an Adapter for converting between these interfaces.

Consequences

An Adapter allows a component to be reused in circumstances other than initially planned for, without changing the original implementation.

In many cases, a component used with an Adapter cannot be used independently in the same application, as the Adapter will already be handling its events. As with the Decorator, the Placeholder pattern (Section 8.5) can help ensure this.

An Adapter can be used to adapt many different implementations of its used interfaces if it doesn't embody assumptions or behaviour specific to a particular adapted component. However, the singleton nature of nesC components means that a particular adapter can only be used once in a given application.

Adding an additional layer to convert between interfaces may increase the application's resource consumption (ROM, RAM and execution time).

Related Patterns

- Decorator: A Decorator adds functionality to an existing component while preserving its original interface. An Adapter presents existing (and possibly additional) functionality via a different interface.

Chapter 9

Advanced Topics

The prior chapers have dealt with the basics of nesC programming: components and composition. Besides generic components, version 1.2 of nesC introduces several new language features which deal with limitations and problems encountered in TinyOS 1.x. This chapter describes two major features, attributes and external types. The former is a general mechanism for extending the language and as of TinyOS 2.0 has only a few uses, including wiring checks. The latter is a way to specify platform-independent data formats, thereby supporting cross-platform message formats that do not require C preprocessor macros or utility functions such as the ntoh and hton of sockets fame.

9.1 Attributes

Attributes are a way to associate metadata with program statements. nesC attributes are based on the approach taken with Java 5 attributes. All of the details are a bit beyond the scope of this document, but it's worthwhile to present the most common attributes and how they are used.

First, there are two kinds of attributes: gcc and nesC. gcc attributes look like this:

```
__attribute__((...))
```

Listing 9.1: A gcc attribute

An example of a gcc attribute is packed, which tells gcc to ignore normal platform struct alignment requirements and instead pack a structure tightly. This attribute is used somewhat in 1.x:

```
typedef struct RPEstEntry {
    uint16_t id;
    uint8_t receiveEst;
```

```
} __attribute__ ((packed)) RPEstEntry;
```

Listing 9.2: The dreaded "packed" attribute in the 1.x MintRoute library

The packed attribute was originally thought of as a good way to make data structures platform independent. Specifically, to make it so code running on an atmega128 and code running on an x86 could agree on a data format. The problem is that not all architectures can deal with packed structures properly (they require that you be able to load unaligned words). For example, the MSP430 family (used in Telos motes) can't handle packed structures. In short, using packed structures leads to code that can only run on certain platforms. The next section of this chapter discusses the way to address cross-platform data formats in TinyOS 2.0. Most of the gcc attributes have nesC equivalents; their use is slowly being phased out.

Programming Hint 14: Never, ever use the "packed" attribute.

nesC attributes look more like Java attributes. An attribute declaration is a struct declaration where the name is preceded by @. Attributes can therefore have fields, which can be initialized. The nesC reference manual goes into greater depth on this topic: for the most part, all currently used nesC attributes do not have fields and so this detail isn't too important. To annotate code with an attribute, you instantiate it. For example

```
struct @atleastonce { };

configuration LedsC {
  provides interface Init @atleastonce();
  provides interface Leds;
}
```

Listing 9.3: nesC attributes

This example shows the declaration of the atleastonce attribute, which has no fields. The configuration LedsC annotates its Init interface with the attribute. By default, the @atleastonce attribute doesn't do anything. But the nesC compiler has tools to output information about an application, including attributes. Part of the default TinyOS 2.x compilation process includes running a script that checks wiring constraints, of which atleastonce is one (the others are atmostonce, exactlyonce).

The tool checks that an interface annotated with atleastonce is wired to at least once. In the case of something like Init, the utility of this check is pretty clear: you can check at compile time that your component is being initialized.

Attributes provide a way to expend the nesC language without introducing new keywords, which could break older code. The current common attributes are:

- **@spontaneous**: this function might be called from outside the nesC program, and so should not be pruned by nesC's dead code elimination. This attribute is needed for interrupt handlers and whenever you want to link binaries (e.g., with TOSSIM).
- **@C**: this function should be considered a C, rather than a nesC, function. Specifically, if a component defines a function with this attribute, then it is not make private to the component. This attribute is needed for when C code needs to call nesC code (e.g., in TOSSIM).
- **@hwevent**: this function will be called as a result of a hardware event. Implies spontaneous.
- **@atomic_hwevent**: this function will be called as a result of a hardware event, and will execute in an atomic section. Implies spontaneous. The distinction between this attribute and @hwevent is needed so nesC can know whether additional atomic sections are needed.
- **@atmostonce**: this interface must be wired to zero or one times.
- **@atleastonce**: this interface must be wired one or more times.
- **@exactlyonce**: this interface must be wired to once, no more, no less.
- **@integer**: this type parameter to a generic component must be an integer. This attribute allows generic components to use arithmetic on the type, which is important for things like scaling timers.
- **@combine**: this attribute is used to specify a combine function for a type when the type is declared. It takes a parameter, a string of the name of the combine function.

9.2 Platform Independent Types

The previous section alluded to a problem encountered in TinyOS programs that the packed attribute tried to solve and failed: platform independent data formats. The basic problem is that TinyOS generally uses structs to define message formats. For example, the standard header of a CC2420 packet (from CC2420.h)[1] looks something like this:

```
typedef struct cc2420_header_t {
  uint8_t length;
  uint16_t fcf;
  uint8_t dsn;
```

[1] Standard in that 802.15.4 has several options, such as 0-byte, 2-byte, or 8-byte addressing, and so this is just the format TinyOS uses by default.

```
  uint16_t destpan;
  uint16_t dest;
  uint16_t src;
  uint8_t type;
} cc2420_header_t;
```

Listing 9.4: CC2420 packet header

That is, it has a one byte length field, a two-byte frame control field, a one byte sequence number, a two byte group, a two byte destination, a two byte source, and one byte type fields. Defining this as a structure allows you to easily access the fields, allocate storage, etc. The problem, though, is that the layout and encoding of this structure depends on the chip you're compiling for. For example, the CC2420 expects all of these fields to be little endian. If your microcontroller is big endian, then you won't be able to easily access the bits of the frame control field. On an atmega128, the structure fields will be aligned on one-byte boundaries, so the layout will work fine. On an MSP430, however, two-byte values have to be aligned on two-byte boundaries: you can't load an unaligned word. So the MSP430 compiler will introduce a byte of padding after the length field, making the structure incompatible with the CC2420 and other platforms. There are a couple of other issues that arise, but the eventual point is the same: TinyOS programs need to be able to specify platform-independent data formats that can be easily accessed and used.

To accomplish this goal, nesC 1.2 introduces platform independent types. Platform independent simple types (integers) are either big-endian or little-endian, independently of the underlying chip hardware. Generally, an external type is the same as a normal type except that it has `nx_` preceding it:

```
nx_uint16_t val;  // A big endian 16-bit value
nxle_uint32_t otherVal; // A little endian 32-bit value
```

Listing 9.5: Examples of external simple types

In addition to simple types, there are also platform independent structs, declared with nx_struct. Every field of a platform independent struct must be a platform independent type, and it aligns all fields on byte boundaries. For example, this is how TinyOS 2.0 declares the CC2420 header:

```
typedef nx_struct cc2420_header_t {
  nxle_uint8_t length;
  nxle_uint16_t fcf;
  nxle_uint8_t dsn;
  nxle_uint16_t destpan;
  nxle_uint16_t dest;
  nxle_uint16_t src;
  nxle_uint8_t type;
} cc2420_header_t;
```

Listing 9.6: The CC2420 header

Any hardware architecture that compiles this structure uses the same memory layout and the same endianness for all of the fields. This enables platform code without resorting to macros to pack and unpack stuctures, or utility functions such as UNIX socket htonl and ntohs.

Programming Hint 15: Always use platform independent types when defining message structures.

Under the covers, nesC translates network types into byte arrays, which it packs and unpacks with each access. For most nesC codes, this has a negligible runtime cost. For example, this code

```
nx_uint16_t x = 5;
uint16_t y = x;
```

Listing 9.7: Translating between local and platform independent types

will rearrange the bytes of x into a native chip layout for y, which takes a few cycles. This means that if you need to perform significant computation on arrays of multibyte values (e.g., encryption), then you should copy them to a native format before doing so, then move them back to a platform indepenent format when done. A single access costs a few cycles, but thousands of accesses costs a few thousand cycles.

Programming Hint 16: If you have to perform significant computation on a platform independent type or access it many (hundreds or more) times, then temporarily copying it to a native type can be a good idea.