Bretny Khamphavong
October 14, 2010

The nesC Language

nesC is a an extension of the C programming language used primarily for networked embedded systems. The components of such systems are referred to as “motes.” nesC was designed and written by a group of researchers from UC Berkeley, Intel and Harvard University. Authors included are Eric Brewer, David Culler, David Gay, Phil Levis, Rob von Behren and Matt Welsh.

To understand the design of nesC, it is best to first understand the challenges that face an operating system and programming language for such networked embedded systems. First, motes are highly reactive in terms of their normal operation. They react to their environment or to radio signals sent by other motes or by a controlling base station. As such, they are driven by events rather than by interaction the way many common computers are. In addition, motes must process data as well as handle event arrival concurrently and any language for these systems must be designed accordingly. Second, motes have very limited hardware resources, and this is not expected to change. In many cases the emphasis is on the physical size of the mote rather than on memory or processing power. Technology advances thus result in smaller, rather than better performing motes. Third, the software must enable highly available applications so as to reduce mote failure due to software. To achieve this goal, a language must be designed to minimize the number of run-time errors and instead detect these potential issues at compile time. Finally, there is one area where the language can be given a little more flexibility; real-time requirements are considered soft rather than hard real-time requirements. The authors note that most real-time requirements are adequately met by the holistic nature of applications written for motes as well as by limiting utilization. Other requirements are not as hard as they first seem because they rely on physical components that are inherently unreliable and therefore, may not actually have hard requirements.

To achieve these goals the authors of nesC defined a few principles which would guide the overall design of the language. As stated before, nesC is an extension of C. C already provides the low level features required to interact with hardware. Interaction with existing C code is very easy, and many developers are already familiar with the C syntax. One significant disadvantage of using just C is that it does not provide much help in writing safe code or in structuring applications; this is where the extensions that nesC provides add real value. When designing nesC, the authors assumed that any nesC compiler would have the advantage of a whole program analysis for safety and performance reasons. This is feasible due to the hardware restrictions on motes, but it does mean that all libraries must be statically compiled with applications instead of dynamically linking them at run-time. In addition, the authors made nesC a static language in terms of memory allocation. This means that memory cannot be dynamically allocated at run time. This allows the compiler further insight into the application code at compile time rather than waiting to see if the application fails at run-time.

The basic building block of nesC applications are components. Components both provide and use interfaces. If a component provides an interface, then it must implement the commands for that interface. If a component uses an interface, then it must have code to properly handle events defined within that interface. Interfaces separate definition from implementation so that it is easier to change those implementations as hardware and requirements change. For example, there is an ADC interface which can be implemented by many different sensors, whether a temperature sensor or a light sensor. Components and interfaces address the challenges around quickly changing hardware as well as the fact that motes operate in an event driven environment.

The next challenge that must be addressed centers around dealing with events and data processing at the same time, or concurrency. This must be done on the very limited hardware available. The authors define two types of code: Asynchronous Code (AC) and Synchronous Code (SC). AC is code that is reachable from at least one interrupt handler and thus the timing of its execution cannot be perfectly predicted. On the other hand, SC is code that is only reachable from tasks, and therefore can never interfere with other synchronous code, since the hardware limitations of motes prevent multiple threads of execution. Contrary to SC, any shared state that is updated from an AC is a potential race condition; and by extension if that shared state is also updated by SC, it remains a race condition. There is a concept of an “atomic section” to deal with these race conditions. By analyzing the whole application, the nesC compiler can identify data that is updated by AC and throw a compile-time error if that data is not inside an atomic section (denoted by the “atomic” keyword). nesC provides a mechanism to override this error in the case where updating a piece of data appears to be a race condition to the compiler, but the developer knows that in fact it is not by using the “norace” keyword.

In conclusion, nesC is written to support deeply embedded networked systems and has had wide success as seen in its use to implement TinyOS, TinyDB and Maté. Despite the fact it is an extension of C, the concepts and principles could be applied to any imperative language. The power and flexibility provided by interfaces make developing complex applications on these embedded systems reliable as well as high performing.

Sources Referenced:

The nesC Language: A Holistic Approach to Networked Embedded Systems
http://www.cs.berkeley.edu/~pal/pubs/nesc-pldi03.pdf

nesC: A Programming Language for Deeply Networked Systems
http://nescc.sourceforge.net/