TinyGALS: A Programming Model for Event-Driven Embedded Systems

Elaine Cheong, Judy Liebman
Department of EECS
University of California, Berkeley
Berkeley, CA 94720
{celaine, judithl}@eecs.berkeley.edu

Jie Liu, Feng Zhao
Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304
{jieliu, zhao}@parc.com

ABSTRACT

Networked embedded systems such as wireless sensor networks are usually designed to be event-driven so that they are reactive and power efficient. Programming embedded systems with multiple reactive tasks is difficult due to the complex nature of managing the concurrency of execution threads and consistency of shared states. This paper describes a globally asynchronous and locally synchronous model (TinyGALS) for programming event-driven embedded systems. Software components are composed locally through synchronous method calls to form modules, and asynchronous message passing is used between modules to separate the flow of control. In addition, a guarded yet synchronous model (TinyGUYS) is designed to allow thread-safe sharing of global state by multiple modules without explicitly passing messages. This programming model is structured such that all asynchronous message passing code and module triggering mechanisms can be automatically generated from a high-level specification. We have implemented the programming model and code generation facilities on a wireless sensor network platform known as the Berkeley motes. As an example, we have redesigned a multi-hop *ad hoc* communication protocol using the TinyGALS model.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features — *Concurrent programming structures*

General Terms

Languages

Keywords

Embedded systems, programming models, globally synchronous and locally asynchronous, code generation, sensor network

1. INTRODUCTION

Emerging embedded systems, such as those in sensor networks [10], [18], [26], intelligent highway systems [24], smart office spaces [1], peer-to-peer collaborative cell phones and PDAs, are usually networked and event-driven. In these systems, the communication network is typically formed in an *ad hoc* manner, and the embedded computers must respond to many (typically unstructured) stimuli, including physical events, user commands, and messages from other systems. As a consequence, software in these systems is typically concurrent and event-driven, involving multiple tasks with extensive synchronization between the tasks.

As the complexity of networked embedded systems grows, the cost of developing software for these systems increases dramatically. Typical embedded software development technologies, inherited from writing device drivers and optimizing assembly code for fast response and a small footprint, do not scale with the application complexity. In fact, it was not until recently that "high-level" languages such as C replaced assembly languages as the dominant embedded software programming languages. Nevertheless, most of these high-level languages are designed for sequential programs running on an operating system; thus they fail to handle concurrency intrinsically. Embedded software designers directly face issues such as maintaining consistent state across multiple tasks, handling interrupts, avoiding deadlock, scheduling concurrent threads to achieve timely responses, and managing resources to preserve energy [11], [15].

Modern software engineering practices advocate the use of software components such as standard libraries, objects, and software services to reduce redundant development and improve productivity. However, when developing a component in an event-driven system, it is not foreseeable whether the component should lock any resources. For example, if a software component is developed with resource synchronization in mind, then it may be overspecified in an application that does not require synchronization, which will result in a large footprint and slow execution. On the other hand, a software component that does not include synchronization code may not be thread-safe, and it may exhibit unexpected behavior when composed with other components.

In this paper, we describe a globally asynchronous and locally synchronous (GALS) approach for programming event-driven embedded systems. The approach maintains the sequentiality of basic components, provides language constructs to allow designers to specify concurrency explicitly at the system level, and generates an application-specific operating system (or more precisely, an execution framework) to provide a thread-safe execution environment for the components.

Terms such as "synchronous," "asynchronous," and "globally asynchronous and locally synchronous (GALS)" mean different things to different communities, thus causing confusion. They may sometimes refer to whether there exists a common clock [20] or a common tick [5] in a model. Our notions of synchrony and asynchrony, which are consistent with the usage of them in distributed

This work is supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-00-C-0139 through the SensIT Program.

SAC 2003, Melbourne, Florida, USA

programming paradigms [19], refer to whether the software flow of control is immediately transferred to a component.

Our programming model, called *TinyGALS*, uses two levels of hierarchy to build an application. At the system level, a set of *modules* communicates asynchronously through message passing. Within each module, *components* communicate via synchronous method calls, as in most imperative languages. Thus, the programming model is globally asynchronous and locally synchronous in terms of the method call mechanisms. Message passing at the global level may be inefficient if every update of data triggers a reaction. In our model, a set of guarded yet synchronous variables (called *TinyGUYS*) is provided at the system level for asynchronous modules to exchange global information "lazily". These variables are thread-safe, yet components can quickly read their values. In this programming model, the designers have precise control over the concurrency in the system, and they can develop software components without the burden of thinking in multiple threads. Together with our code generation facilities, this model can greatly improve software productivity and encourage component reuse.

The rest of the paper is organized as follows. Section 2 discusses related work, emphasizing relevant embedded software models. Section 3 describes the TinyGALS language constructs and semantics. Section 4 discusses a code generation technique based on the two-level execution hierarchy and a system-level scheduler. An example of developing a multi-hop routing protocol using the TinyGALS model is given in Section 5.

2. RELATED WORK

Although concurrency has long been a key research theme for systems software such as operating systems and distributed computing middleware [2], [21], [22], formal treatment of them in embedded systems has largely been ignored by mainstream computer science researchers until recently [16]. Embedded software is typically application specific with tight constraints on available processing power and memory space, which makes traditional approaches such as the use of layers of abstraction and middleware less applicable. A more promising approach is to use formal concurrency models to construct and analyze software designs, and to use software synthesis technologies to generate application-specific scheduling and execution frameworks that give designers detailed control of timing, concurrency, and memory usage.

One example of this more formal approach is the synchronous/reactive design methodology [13]. Languages such as Esterel [6] and Signal [12] allow users to specify a system using the notion of global ticks and concurrent zero delay reactions. These models are specific enough that the concurrency in the system can be compiled away, and the system behaves like a state machine at run time. Actor-oriented designs, such as those seen in the Ptolemy II framework [9], integrate a rich set of sequential and concurrent models, use a system-level type system to check their composability [17], and compile away as much concurrency as possible to obtain run-time efficiency [4], [25], [7]. In these approaches, the basic components in the system typically have an intrinsic notion of concurrency. However, designing thread-safe components can be a big challenge for complex engineering applications, not to mention that many of the existing components and libraries only work in sequential execution environments.

Not all concurrency can be compiled away in all applications. When the rate of input events does not match the embedded computers' processing speed and real-time requirements, multiple threads of execution are truly needed. In these situations, it is not uncommon to use asynchronous models to coordinate sequential reactions. A key question for the coordination model is how can asynchronous tasks share messages and global state. In the port-based object (PBO) model as implemented in Chimera [23], data are stored in a global space. A concurrent task, called a PBO, is free to access the data space at any time. Data in the space are persistent, and PBOs are only triggered by time with no explicit message passing among them. In the POLIS co-design approach [3], software modules are generated as asynchronous tasks connected by message passing through a buffer of size one. There is a global scheduler that schedules tasks in a round-robin fashion. There is no global data space in which to share data, and information is only exchanged in the communication messages. Our approach differs in that we allow designers to directly control concurrent execution and the buffer sizes among asynchronous modules. At the same time, we use a thread-safe global data space to store lazy messages that do not trigger immediate reactions. Components in our model are sequential, so that they are both easy to develop and backward compatible with most legacy software.

Our model is also influenced by the TinyOS project at U.C. Berkeley [8]. In fact, the first implementation of our model uses TinyOS components, and targets the same hardware platform — the Berkeley motes. Unlike our model, concurrent tasks in TinyOS are not part of the component interface and are completely hidden from the users. Lack of explicit management of concurrency forces component developers to manage concurrency by themselves, which may make TinyOS components extremely hard to develop.

3. TinyGALS PROGRAMMING MODEL

TinyGALS is based on a model of computation that contains globally asynchronous and locally synchronous method calls. We use this architecture to separate the rates of control in a system — the reactive part and the computational part — via asynchrony.

3.1 Language Constructs and Semantics

A TinyGALS program contains a single system composed of modules, which are in turn composed of components. We describe each of these constructs (components, modules, and system) and their semantics in turn.

Components are the most basic elements of a TinyGALS program. A TinyGALS component C has a set of internal variables V_C, a set of external variables X_C, and a set of methods that operate on these variables I_C. An external variables can not be an internal variable of another component. The set of method call interfaces I_C is further divided into two disjoint sets: $ACCEPTS_C$ and $USES_C$. The methods in $ACCEPTS_C$ can be called by other components, while the methods in $USES_C$ are those needed by C and may possibly belong to other components. Thus, a component is like an object in most object-oriented programming languages, but with explicit definitions of the external variables and the methods it needs. Syntactically, a component is defined in two parts — an interface definition and an implementation[1]. Figure 1 shows a pseudodefinition of a component `DownSample`. The interface of the component has two *ACCEPTS* methods and one *USES* method. Let `_active` be an internal boolean variable, then for every other `fire()` method called, the component will call the `fireOut()` method with the same integer argument. The component only cares

1. The syntax is similar to that in TinyOS.

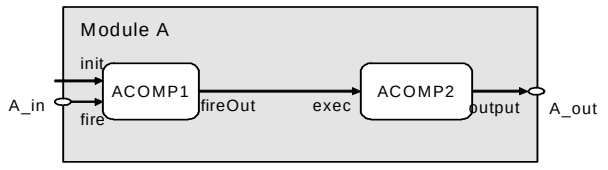
```
COMPONENT DownSample
ACCEPTS {
  void init(void);
  void fire(int in);
};
USES {
  void fireOut(int out);
};
```

```
//_active is an
//internal variable.
void init() {
  _active = true;
}
void fire(int in) {
  if (_active) {
    CALL_COMMAND
      (fireOut)(in);
    _active = false;
  } else {
    _active = true;
  }
}
```

Figure 1. The definition of a component `DownSample`**.**

about the type signature of `fireOut()`, but it does not care about to which component the method belongs. The `CALL_COMMAND` keyword indicates that the `fireOut()` method will be called synchronously with the integer as its argument.

Modules encompass one or more TinyGALS components. A module M is a 6-tuple: $M = (COMPONENTS_M, INIT_M, INPORTS_M, OUTPORTS_M, PARAMETERS_M, LINKS_M)$, where $COMPONENTS_M$ is the set of the components that form the module; $INIT_M$ is a list of methods that belong to the components in $COMPONENTS_M$; $INPORTS_M$ and $OUTPORTS_M$ are sets that specify the inputs and outputs of the module, respectively; $PARAMETERS_M$ is a set of variables external to the components; and $LINKS_M$ specifies the relations among the method call interfaces of the components and the inputs and outputs of the module. Links are synchronous method calls. When a component calls an external method through `CALL_COMMAND`, the flow of control in the module is immediately transferred to the callee component or an output port. Figure 2 shows a graphical representation and the module definition code for a module `A` that contains two components `ACOMP1` and `ACOMP2`. Component `ACOMP1` accepts `init()` and `fire()`, and uses `fireOut()`. Component `ACOMP2` accepts `exec()` and uses `output()`. The link section at the bottom of the definition of module `A` declares that whenever `A_in` is triggered (which will be explained later), the `fire()` method of `ACOMP1` will be called; whenever `ACOMP1` calls `fireOut()`, the method `exec()` of `ACOMP2` will be called; and whenever `ACOMP2` calls `output()`, an event will be produced from `A_out`. All of these are synchronous method calls that will be

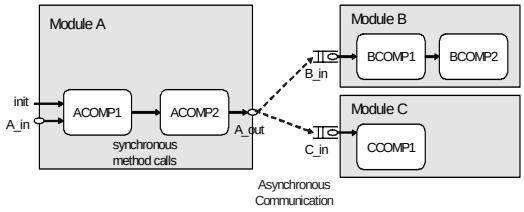



```
include components{
  ACOMP1;
  ACOMP2;
};
init{
  ACOMP1:init
};
ports in{
  A_in;
};
```

```
ports out{
  A_out;
};

ACOMP1:fireOut\
ACOMP2:exec
ACOMP2:output A_out
A_in ACOMP1:fire
```

Figure 2. A module `A` **contains two components that communicate via synchronous method calls.**




```
// Definition of S
include modules{
  A;
  B;
  C;
};
A_out -> B_in 10
A_out -> C_in 10

START@ A_in
```

Figure 3. A TinyGALS system with three modules.

executed immediately, and components inside of modules execute to completion. Some components may handle interrupts. In a module, these components appear as source components that do not connect to any input ports.

At the top level of a TinyGALS program, modules are connected to form a complete system. A system S is a 5-tuple: $S = (MODULES_S, GLOBALS_S, VAR\_MAPS_S, CONNECTIONS_S, START_S)$, where $MODULES_S$ is a list of modules in the system; $GLOBALS_S$ is a set of global variables; VAR_MAPS_S is a set of mappings, each of which maps a global variable to a parameter of a module in $MODULES_S$; $CONNECTIONS_S$ is a list of the connections between module output ports and input ports; $START_S$ is the name of an input port of exactly one module in the system that should be used as the starting point of execution of the system. Although links within a module can only have one caller and one callee, the connections between modules can have multiple input ports and multiple output ports.

Communication between ports occurs asynchronously via FIFO queues. When a component within a module calls a method that is linked to an output port, the arguments of the call will be converted into *tokens*. For each input port connected to the output port, a copy of the token is placed in its FIFO queue. Later, a scheduler in the TinyGALS runtime system will remove a token from the FIFO queue and call the method that is linked to the input port with the contents of the token as its arguments. The queue separates the flow of control between modules; the call to the output port will return immediately, and the component within the module can proceed. The TinyGALS semantics do not define exactly when the input port is triggered. A simple implementation could process the tokens in the order that they are generated. More sophisticated scheduling algorithms can be added, such as one that would take care of timing and energy concerns. Communication between modules is also possible without the transfer of data. In this case, an empty message transferred between ports acts as a pure trigger for activation of the receiving module.

Figure 3 shows a TinyGALS system with three modules `A`, `B`, and `C`. The output port of `A` is connected to the input ports `B_in` of module `B` and `C_in` of module `C`. The definition declares that the connection between `A_out` and `B_in` has a FIFO queue of size 10, and the same for the connection between `A_out` and `C_in`. The single-output-multiple-input connection acts as a fork, i.e. every token produced by `A_out` will be duplicated and trigger

both B_in and C_in. On the other hand, a multiple-output-single-input connection has a merge semantics, such that tokens from multiple sources are merged into a single stream in the order that the tokens are produced. The last line in the definition says that the system will start by triggering port A_in exactly once.

3.2 TinyGUYS

The TinyGALS programming model has the advantages that modules become decoupled through message passing and are easy to develop independently. However, each message passed will trigger the scheduler and activate a receiving module, which may quickly become inefficient if there is global state which must be updated frequently. TinyGUYS (Guarded Yet Synchronous) variables is a mechanism for sharing global state, allowing quick access but with protected modification of the data.

In the TinyGUYS mechanism, global variables are guarded. Modules may read the global variables synchronously (without delay). However, writes to the variables are asynchronous in the sense that all writes are buffered. The buffer is of size one, so the last module that writes to a variable wins. TinyGUYS variables are updated by the scheduler only when it is safe (e.g., after one module finishes and before the scheduler triggers the next module).

```
// System definition
include modules{
  A;
  B;
};
globals{
  statevar;
};

statevar <=> A_param
statevar <=> B_param
...
```

```
//Module definition: A
include components{
  A1;
};
parameters{
  int A_param;
};

...
```

```
// Component implementation:
// A1
...
void fire() {
  ...
  int a;
  a=PARAM_GET(A_param);
  a++;
  PARAM_PUT(A_param)(a);
  ...
}
```

Figure 4. Defining and accessing TinyGUYS variables.

TinyGUYS have global names that are mapped to the parameters of each module and are further mapped to the external variables of the components that use these variables, as shown in Figure 4. The external variables are accessed within a component by using special keywords: PARAM_GET and PARAM_PUT.

4. CODE GENERATION

Given the highly structured architecture of the TinyGALS model, scheduling and event handling code can be automatically generated to free software developers from writing error-prone concurrency control code. We have created a set of code generation tools for the Berkeley motes that, given the definitions for the components, modules, and system, automatically generate all of the necessary code for (1) component links and module connections, (2) system initialization and start of execution, (3) communication between modules, and (4) global variable reads and writes.

Throughout this section, we will use the example system illustrated in Figure 5. This system is composed of two modules: A and B. Module A contains two components, ACOMP1 and ACOMP2. Component ACOMP2 accesses a parameter and has a definition similar to the definition of component A1 given in Figure 4. Module B contains a single component named BCOMP1, which has a function called fire() that is connected to the input port B_in of module B. Figure 5 also summarizes all functions and data structures generated by the code generator.

4.1 Links and Connections

The code generator generates a set of aliases which create the mappings for the links between components, as well as the mappings for the connections between modules. In the example, an alias is generated for the link between ACOMP1 and ACOMP2, and another for the connection between modules A and B.

4.2 System Initialization and Start of Execution

The code generator creates a system-level initialization function called app_init(), which contains calls to the *INIT* methods of each module in the system. The module order listed in the system definition determines the order in which the *INIT* methods are called. The app_init() function is one of the first functions called by the TinyGALS runtime scheduler before executing the application. In Figure 5, only module A contains an initialization list. Therefore, the generated app_init() function only contains a call to ACOMP1:init.

The code generator also creates an application start function, app_start(). This function triggers the input port of the module defined as the system starting point. In the example, app_start() contains a trigger for the input port A_in of module A.

4.3 Communication

The code generator automatically generates a set of scheduler data structures and functions for each connection between modules.

For each input port of a module, the code generator generates a queue of length *n*, where *n* is specified in the system definition. The width of the queue depends on the number of arguments of the method that is connected to the port. If there are no arguments, then as an optimization, no queue is generated for the port (but space is still reserved for events in the scheduler event queue). A pointer and a counter are also generated for each input port to keep track of the location and number of tokens in the queue. For the example in Figure 5, the definition of port B_in results in the gen-

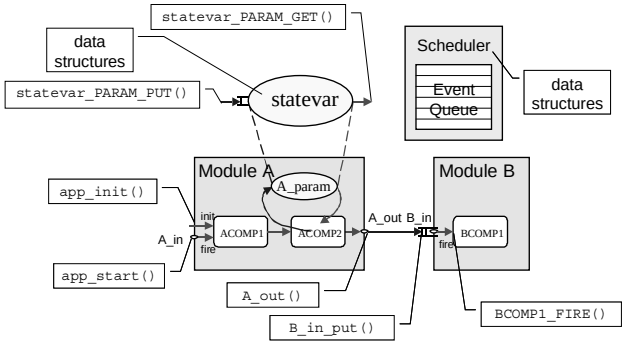


Figure 5. A code generation example.

eration of a port queue called `B_in_0[]` (assuming port `B_in` is linked to a function with a single argument), as well as the generation of `B_in_head` and `B_in_count`.

For each output port of a module, the code generator generates a function that has the same name as the output port. This outport function is called whenever a method of a component wishes to write to an output port. The type signature of the outport function matches that of the method that connects to the port. For each input port connected to the outport, a `put()` function is generated which handles the actual copying of data to the inport queue. The outport function calls the inport `put()` function for each connected inport. The `put()` function adds the port identifier to the scheduler event queue so that the scheduler will activate the module at a later time. In the example, for the outport `A_out` of module `A`, a function `A_out()` is generated, which in turn calls the generated function `B_in_put()` to insert data into the queue.

If the queue is full when attempting to insert data into the queue, one of several strategies can be taken. We currently take the simple approach of dropping input events when the queue is full. However, an alternate method is to generate a callback function which attempts to requeue the event at a later time. Yet another approach would be to place a higher priority on more recent events by deleting the oldest event in the queue to make room for the new event.

For each connection between a component method and a module input port, a function is generated with a name formed from the name of the input port and the name of the component method. When the scheduler activates a module via an input port, it first calls this generated function to remove data from the input port queue and then passes it to the component method. In the example, module `B` contains an input port `B_in`, which is connected to function `fire()` of component `BCOMP1`. The code generator creates a function called `BCOMP1_FIRE()`, which removes data queued in `B_in_0[]`, modifies `B_in_head` and `B_in_count`, and calls `BCOMP1:fire`.

4.4 Global Variables

The code generator generates a pair of data structures and a pair of access functions for each global variable declared in the system definition. The pair of data structures consists of a data storage location of the type specified in the module definition that uses the global variable, along with a buffer for the storage location. The pair of access functions consists of a `PARAM_GET()` function that returns the value of the global variable, and a `PARAM_PUT()` function that stores a new value for the variable in the variable's buffer. A generated flag indicates whether the scheduler needs to update the variables by copying data from the buffer.

In the example, a global variable named `params.statevar` will be generated, along with a buffer named `params_buffer.statevar`. The code generator will also create functions `statevar_PARAM_GET()` and `statevar_PARAM_PUT()`.

4.5 Memory Usage

Table 1 and Table 2 show the sizes of the generated functions and variables for the system shown in Figure 5. Note that the system contains one port. Here, we assume that `A_out` writes variables of type `short` and returns a confirmation value of type `char` (type signature: `char A_out(short)`). Additionally, the queue connected to inport B_in is of size 50 (i.e., it holds 50 elements of type `short`).

Table 1: Sizes of generated functions

Function Name	Bytes of code (decimal)
`app_init()`	58
`app_start()`	6
`A_out()`	12
`B_in_put()`	160
`BCOMP1_FIRE()`	98
`A_param_PARAM_GET()`	10
`A_param_PARAM_PUT()`	16

Table 2: Sizes of generated variables

Variable Name	Bytes (decimal)
`eventqueue_head`	2
`params`	2
`entrypoints`	2
`eventqueue_count`	2
`eventqueue`	100
`ports`	104
`params_buffer_flag`	1
`params_buffer`	2

Thus, memory usage of a TinyGALS application is determined mainly by the user-specified queue sizes and total number of ports in the system. The TinyGALS run-time scheduler is very lightweight, since the event queues are generated as application specific data structures. The scheduler itself takes 112 bytes of memory, comparable with the 86-byte original TinyOS scheduler.

5. EXAMPLE

The TinyGALS programming model and code generation tools have been implemented for the Berkeley motes [14] and are compatible with existing TinyOS components. In this section, we will give an example of the redesign of a multi-hop *ad hoc* communication protocol known as the beacon-less protocol or BLESS.

The specific generation of motes, called *MICA motes*, that we used in this example have a 4MHz ATMEGA103L microcontroller, 128K of programming memory, 4K data memory, 5 types of sensors, and a RF communication component. BLESS is a single base multi-hop protocol, where a single base station collects information from a distributed wireless sensor network. BLESS dynamically builds a routing tree rooted at the base station. A sensor node finds its *parent* in the routing tree by listening to the network traffic.

The base station periodically sends a beacon message. Those nodes that directly receive this message will label themselves one hop from the base, as in Figure 6(a). When a one hop node sends a message, several nodes that were not able to hear the base, and therefore have no node to which to send their information messages, will be able to overhear the one hop node's message, as shown in Figure 6(b). Any such overhearing nodes will label themselves two hop nodes and will remember to send their next information message to the one hop node they just overheard. The one

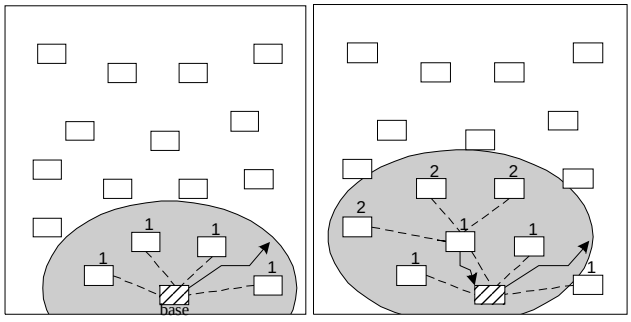
hop node, in turn, will forward any packets destined for itself to the base. After a two hop node has sent an information message the process will repeat and three hop nodes will be designated, and so on. A timeout clock is used to re-initialize the parent list to empty, which allows nodes to tolerate unreliable links as well as nodes that join and leave the network dynamically.

From a single node point of view, the BLESS protocol responds to three types of events −− information messages sent by the local application, network messages overheard or to be forwarded, and the timeout event to reset the parent variables. The three threads of reaction are intertwined. For example, while a node is busy forwarding a message from its child node, its own application may try to send a message and may modify the message buffer. It is also possible that, while a node sends an information message, an overheard message may arrive, which will change the node's parent and in turn the hop number of this node.

The BLESS protocol is implemented in TinyOS v0.6.1[2] as a single monolithic component with eight method call interfaces that handle all three threads of reaction internally. All methods are non-reentrant. To avoid concurrent modification of message buffers, the component makes extensive use of a *pending* lock. Six of the eight methods contain code that adjusts behavior accordingly after setting and/or checking the pending lock. Even so, due to the complexity of managing message queues, the queuing of events is left to application designers who use this component.

Using the TinyGALS model, we redesigned and implemented the BLESS protocol on the motes as three separate modules: BLESS_Start, BLESS_Receive, and BLESS_Send[3], each of which only deals with a single thread of reaction. In addition, there are a number of global variables that use the TinyGUYS mechanism and are accessed from within at least two of the modules. This structure is displayed in Figure 7.

The BLESS_Start module contains a single component that prepares a message initiated by the host mote's own application for sending. The input port of BLESS_Start therefore is connected to an application module and the output port is connected to the sending component within BLESS_Send. The BLESS_Start module



(a) All nodes that can hear the base station label themselves as level 1.

(b) When a level-1 node sends a message, unlabelled nodes that overhear this message will label themselves as level 2.

Figure 6. Illustration of the BLESS protocol.

2. Available at http://webs.cs.berkeley.edu/tos/.
3. Resetting the parent is done within a separated module triggered directly by the clock.

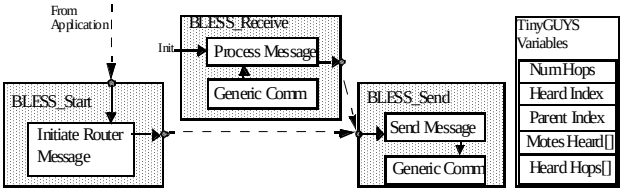


Figure 7. Structure of BLESS implementation under TinyGALS. Each shaded rectangle represents a module. The guarded global variables are shown on the right.

reads global variables to determine the current parent and setup the outgoing message.

The BLESS_Receive module is comprised of two components, one for processing received router messages, Process Message, and the other for supporting the reception of messages from the radio, Generic Comm. Note that Generic Comm is an off-the-shelf TinyOS component, which implements the network stack. The output port of BLESS_Receive is connected to the sending component within BLESS_Send to pass messages that must be forwarded. The BLESS_Receive module accesses the global variables both in order to update them and to use them when forwarding a message.

The BLESS_Send module is comprised of two components, one for reformatting messages, Send Message, and the second for sending messages over the radio, Generic Comm. No global variables are used.

The redesign of the BLESS protocol takes about 3K bytes of memory, which is about 1.5x the code size of the original implementation. But, it has several advantages. First, modularity is built into the design. In the above example, the BLESS_Send module is conveniently used by both of the other two modules to send the specialized router message. Second, global variables are guarded. This is a crucial point due to the earlier discussion of potential problems with shared state and interrupts. Therefore, this second benefit simplifies the code and debugging process by eliminating the need for explicit locks. The third advantage is the easy management of message queues. In the TinyGALS architecture of the above example, when a message is to be sent through Send Message, a request is queued at the input port to the BLESS_Send module. When processing time is available, BLESS_Send will run after dequeuing the messages from its input port. This same call in the existing TinyOS implementation would result in the message being sent immediately, or, if busy, not at all, due to its usage of a single pending lock. The third advantage has consequences not only for code structure and feasibility but also for the overall communication reliability of the network.

6. CONCLUSION

This paper describes a TinyGALS programming model for event-driven multitasking embedded systems. The model allows software designers to use high-level constructs such as ports and message queues to separate the flow of control between modules that contain components composed of synchronous method calls. Guarded yet synchronous variables (TinyGUYS) provide a means of exchanging global data between asynchronous modules without triggering reactions. The high-level constructs are amenable to automatic code generation that releases designers from writing error-prone task synchronization code. We implemented this model for the Berkeley mote sensor network platform. Finally, this

paper describes a multi-hop communication protocol redesigned using the TinyGALS model as an example.

7. REFERENCES

[1] B.A. Akyol, M. Fredette, A.W. Jackson, R. Krishnan, D. Mankins, C. Partridge, N. Shectman, and G.D. Troxel, "Smart Office Spaces," *USENIX Workshop on Embedded Systems*, Cambridge, MA, USA, Mar 29-31, 1999.

[2] G.R. Andrews, *Concurrent Programming: Principles and Practice*, Addison-Wesley, 1991.

[3] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich, and K. Suzuki, "Synthesis of Software Programs for Embedded Control Applications," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, Vol.18, No. 6, 1999, pp. 834-849.

[4] S.S. Bhattacharyya, P.K. Murthy, and E.A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer, 1996.

[5] A. Benveniste, B. Caillaud and P. Le Guernic. "From synchrony to asynchrony," in J.C.M. Baeten and S. Mauw, editors, *10th International Conference on Concurrency Theory* (CONCUR'99), LNCS 1664, Springer Verlag, 1999, pp. 162-177.

[6] G. Berry and G. Gonthier, "The Esterel Synchronous Programming Language: Design, Semantics, Implementation," *Science of Computer Programming*, Vol. 19, No. 2, pp. 87-152, 1992.

[7] M. Cornero, F. Thoen, G. Goossens, and F. Curatelli, "Software synthesis for real-time information processing systems," in P. Marwedel and G. Goossens, Eds., *Code Generation for Embedded Processors*, pp. 260--279. Kluwer Academic Publishers, 1995.

[8] D.E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo, "A Network-Centric Approach to Embedded Software for Tiny Devices," *First Workshop on Embedded Software* (EMSOFT2001), Lake Tahoe, CA, USA, Oct. 8-10, 2001, pp. 114-130.

[9] J. Eker, J.W. Janneck, E.A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity—the Ptolemy Approach," to appear in *Proceedings of the IEEE*.

[10] D. Estrin, L. Girod, G. Pottie, and M. Srivastava, "Instrumenting the world with wireless sensor networks," in *Proc. of ICASSP'2001*, Salt Lake City, UT, May 2001, pp. 2675-2678.

[11] J.G. Ganssle, *The Art of Designing Embedded Systems*, Newnes, 1999.

[12] P. Le Guernic, T. Gautier, M. Le Borgne, and C. Le Maire. "Programming real-time applications with Signal," *Proc. of the IEEE*, Vol. 79, No. 9, pp. 1321-1336, 1991.

[13] N. Halbwachs, *Synchronous Programming of Reactive Systems*, Kluwer Academic Publishers, 1993.

[14] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D.E. Culler, and K. Pister, "System architecture directions for network sensors," in *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS-IX), Cambridge, MA, Nov. 12-15, 2000.

[15] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Kluwer Academic Publishers, 1997.

[16] E.A. Lee, "What's Ahead for Embedded Software?" *IEEE Computer*, September 2000, pp. 18-26.

[17] E.A. Lee and Y. Xiong, "System-Level Types for Component-Based Design," *First Workshop on Embedded Software* (EMSOFT2001), Lake Tahoe, CA, USA, Oct. 8-10, 2001, pp.237-253.

[18] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler, "Wireless Sensor Networks for Habitat Monitoring," *2002 ACM International Workshop on Wireless Sensor Networks and Applications* (WSNA'02), Atlanta, GA, Sept. 2002.

[19] T.J. Mowbray, W.A. Ruh, R.M. Soley, *Inside Corba: Distributed Object Standards and Applications*, Addison-Wesley, 1997.

[20] J. Muttersbach, T. Villiger, W. Fichtner, "Practical Design of Globally-Asynchronous Locally-Synchronous Systems", in *Proc. of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, ASYNC'2000, Eilat, Israel, pp. 52-59, April 2-6, 2000.

[21] D.C. Schmidt, H. Rohnert, M. Stal, and D. Schultz, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, Vol. 2, 2nd Ed., Wiley, John & Sons, 2000.

[22] A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts* (6th Ed.), John Wiley & Sons, 2001.

[23] D.B. Stewart, R.A. Volpe, and P.K. Khosla, "Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects," *IEEE Trans. on Software Engineering*, Vol.23, No.12, Dec. 1997, pp. 759-776.

[24] Pravin Varaiya, "Smart Cars on Smart Roads: Problems of Control," *IEEE Trans. on Automatic Control*, AC-38(2), 1993, pp. 195-207.

[25] L. Wernli, *Design and implementation of a code generator for the CAL actor language*, Technical Memo UCB/ERL M02/5, University of California, Berkeley, CA 94720, March 2002.

[26] F. Zhao, J. Shin, and J. Reich, "Information-Driven Dynamic Sensor Collaboration," *IEEE Signal Processing Magazine*, Vol. 19, No. 2, March 2002, pp. 61-85.