EVENT-TRIGGERED DESIGN OF NETWORKED EMBEDDED AUTOMATION SYSTEMS

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Chidi H. Anozie

December, 2010
EVENT-TRIGGERED DESIGN OF NETWORKED EMBEDDED
AUTOMATION SYSTEMS

Chidi H. Anozie

Thesis

Approved:

Advisor
Dr. Shivakumar Sastry

Committee Member
Dr. Alex De Abreu Garcia

Committee Member
Dr. Hamid Bahrami

Accepted:

Department Chair
Dr. Alex De Abreu Garcia

Dean of the College
Dr. George K. Haritos

Dean of the Graduate School
Dr. George R. Newkome

Date

ii
ABSTRACT

Systems that interact with the physical world can be designed in either an event-triggered (ET) paradigm or a time-triggered (TT) paradigm. Most real-time systems today are designed in the TT-paradigm because it is easier to achieve predictable performance. However, when some critical aspect of the system changes, designs based on the TT-paradigm must be completely revised. Modern application require systems to be flexible and reliable. Since the event-triggered systems are fundamentally designed as a collection of loosely coupled local entities, it was interesting to investigate the predictability of ET-systems.

This investigation focused on a ET-design for operating a MicroFactory, which is a physical simulation of a discrete assembly system. The design was implemented and evaluated using a collection of networked embedded microcontrollers. Common tasks in a discrete assembly environment were captured as a collection of primitives. These primitives were redesigned and implemented in the ET-paradigm. Experimental results collected reflect the performance at the level of a single node and at the level of the system. Results were studied in comparison with similar ones obtained using a design that was based on the TT-paradigm. In the future, principles and techniques for reconfigurability can be incorporated in the proposed framework.
ACKNOWLEDGEMENTS

I would first like to thank God for giving me the ability to work on this research. Thanks to my adviser, Dr. Shivakumar Sastry for his constant support and exceptional guidance that made this thesis a reality. Thanks to all my committee members, Dr. Alex De Abreu Garcia and Dr. Hamid Bahrami for their comments and very important feedback on this thesis. I would also like to acknowledge my colleagues in the Complex Engineered Systems Laboratory, Branden Archer, John McGonnell, Kranthi Mamidisetty, Mukesh Chippa, Chandana Cheerala, and Sanchita Subedi for their support. A big thank you especially to Branden, for helping with the simple things that just made this research process evolve like clockwork.

Finally, I dedicate this thesis to my parents, who have sustained me with their undying love and support!
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CHAPTER</strong></td>
<td></td>
</tr>
<tr>
<td>I. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Contributions</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Overview of this Thesis</td>
<td>5</td>
</tr>
<tr>
<td>II. BACKGROUND</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Motes</td>
<td>6</td>
</tr>
<tr>
<td>2.2 TinyOS</td>
<td>7</td>
</tr>
<tr>
<td>2.3 Automation Systems</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Design Paradigms</td>
<td>13</td>
</tr>
<tr>
<td>III. MICROFACTORY</td>
<td>18</td>
</tr>
<tr>
<td>3.1 Modular Units</td>
<td>18</td>
</tr>
<tr>
<td>3.2 Primitives for Networked Embedded Automation</td>
<td>20</td>
</tr>
<tr>
<td>3.3 Interface to Mica2DOT Motes</td>
<td>27</td>
</tr>
<tr>
<td>IV. COMPONENT-ORIENTED DESIGN</td>
<td>28</td>
</tr>
<tr>
<td>4.1 TinySchedulerC Component</td>
<td>28</td>
</tr>
</tbody>
</table>
4.2 MainC Component ........................................... 29
4.3 Timer Components and Interfaces ....................... 30
4.4 Communication Components and Interfaces .......... 31
4.5 The Main Module Component ............................ 32
4.6 Tasks .................................................... 32
4.7 Design Template ........................................ 35
4.8 Implementation ......................................... 39

V. EXPERIMENTAL RESULTS .................................. 45
5.1 Experimental Approach ................................. 45
5.2 Local Response .......................................... 46
5.3 Remote Response ....................................... 48
5.4 Request Response ...................................... 49
5.5 Publish Subscribe ...................................... 50
5.6 Multihop Primitives ................................... 52

VI. CONCLUSION ............................................... 54

BIBLIOGRAPHY ................................................ 56
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Scan architecture of Programmable Automation Controllers [1].</td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>The Input Bin (a) pushes parts on to a conveyor to initiate assembly operations. The Workcell (b) pulls a part off the conveyor. The turntable turns the parts successively through three workstations and returns the part to the conveyor.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>The Paint Station (a) simulates typical paint operations on a part. The Output Bin pulls parts off the conveyor at the end of the assembly line.</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>An example of a local response primitive.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>An example of a remote response primitive [1].</td>
<td>22</td>
</tr>
<tr>
<td>3.5</td>
<td>An example of a request/response primitive [1].</td>
<td>23</td>
</tr>
<tr>
<td>3.6</td>
<td>An example of a publish/subscribe primitive [1].</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>An example of a multihop remote response primitive [1].</td>
<td>25</td>
</tr>
<tr>
<td>3.8</td>
<td>An example of a multihop request/response primitive [1].</td>
<td>26</td>
</tr>
<tr>
<td>4.1</td>
<td>TinySchedulerC component relationships.</td>
<td>29</td>
</tr>
<tr>
<td>4.2</td>
<td>MainC component Relationships.</td>
<td>30</td>
</tr>
<tr>
<td>4.3</td>
<td>TimerMilliC component relationships.</td>
<td>31</td>
</tr>
</tbody>
</table>
4.4 Main module component relationships. ........................................ 33
4.5 5 possible module unit states for any unit that constitutes the MicroFactory. 36
4.6 An illustration of the structure of the first part of the template: shellP component. .......................... 37
4.7 An illustration of the second part of the template: shellAppC component. 38
4.8 A layout of the units selected for the investigation. .......................... 39
4.9 State machine for the input bin unit. ........................................ 40
4.10 State machine for the painter unit. ......................................... 41
4.11 State machine for the output bin unit. ...................................... 42
4.12 Stage for the unit-to-unit communication protocol implemented in the design. ........................................ 43
5.1 Data gathered from the local response primitive experiment with no background communication .......................... 46
5.2 Data gathered from the local response primitive experiment with background communication incorporated ........................................ 47
5.3 Data gathered from the remote response primitive experiment ........ 48
5.4 Data gathered from the request/response primitive experiment .......... 50
5.5 Data gathered from the publish/subscribe primitive experiment ...... 51
5.6 Data gathered from the multihop remote response primitive experiment 52
5.7 Data gathered from the multihop request/response primitive experiment 53
CHAPTER I
INTRODUCTION

The design paradigm that is used in any system fundamentally limits the system-level performance that can be achieved. This choice of a design paradigm is particularly important in systems where multiple nodes are closely integrated with the dynamics of underlying physical phenomena. *Networked Embedded Automation* systems are a class of systems in which a large number of *motes*, each with its own microcontroller and wireless transceiver, are deeply embedded into modular automation units that can perform pre-defined tasks [1, 2, 3]. The microcontrollers read values of local sensors, compute values for local actuators, and set values for local actuators. These microcontrollers interact over wireless links to coordinate their local operations in order to achieve system-level objectives. This investigation focused on using an event-triggered (ET) design paradigm for realizing the local and system-level behaviors in networked embedded automation systems.

Automation systems are critical infrastructures that drive our economies and societies. The current-state-of-practice is based on using one or more industrial controllers, called Programmable Automation Controllers, that are organized as a hierarchy with multiple layers of supervisory controllers [4]. These systems are integrated with information systems, supply chain, inventory control, and such other enterprise-
level systems. Such designs are brittle and do not benefit from recent advances in distributed systems and fault management. Nevertheless, despite the limitations of the hierarchical design paradigm, existing automation systems are reliable, safe and predictable. Consequently, it is important to assure designers that any new technologies for automation will not compromise the reliability, predictability and safety of these systems. Experience with automation systems over the past two decades has shown that modular and networked systems enable finer-grained control, improved diagnostics and maintainability, and reduced system life-cycle costs [4].

The simplicity of node-level design that enables easy scale-up of systems is one of the most attractive features of networked embedded automation [5]. This enables designers to match the structure of the computing and communicating platform with the structure and the geometry of the controlled applications. Such an approach makes it possible to co-locate system functionality close to events of interest in the application. Further, this approach enables system designers to incrementally grow the size and capacity of the automation system as demand increases and quickly reconfigure the system in response to planned or unplanned disruptions. The wireless connectivity between the motes that regulate local units is particularly important for rapid reconfiguration of the system topology, especially in response to unplanned disruptions such as a component, device, or sub-system failure. Because the system-level functionality is now distributed over multiple motes, there are interesting design challenges that must be addressed to achieve the coordinated system-level objectives for the automation systems.
There are two fundamentally different design paradigms that have been extensively studied in the literature. These are the event-triggered (ET) paradigm and the time-triggered (TT) paradigm [6]. Both approaches assume that the physical world, i.e., the application environment, is event-triggered. This means that the automation system has little or no control on dictating when events of interest occur in the application. To achieve local or system-level objectives, the automation system needs to respond to such events in a timely manner. In the event-triggered paradigm, the nodes of the system, i.e., the motes that regulate local behaviors, are programmed to respond to the non-temporal events that occur in the application as soon as possible. Typically, when an event occurs, an interrupt is issued to a local microcontroller; an interrupt-handler in the microcontroller responds by posting a corresponding task to a local task-queue; and tasks in the task-queue are executed in order depending on the number of threads of control that the microcontroller can support. A schedule is dynamically unfolded at runtime for the event-triggered paradigm. This allows for communication between nodes to be initiated by these non-temporal events, as a result, better flexibility and simplicity of implementation is able to be realized [7, 8].

In contrast, in the time-triggered paradigm, all node-level activities such as sensing, actuation, and communication, are performed according to a temporal schedule that is established at design-time. Typically, such a design assumes that a collection of periodic, aperiodic, and sporadic tasks must be executed to achieve the local and system-level objectives of the automation system. The significant differences between these two approaches are described in more detail in Chapter 2.
This investigation focused on the event-triggered paradigm for designing networked embedded automation systems. The design was realized using the component-oriented architecture in the TinyOS framework [9]. In this approach, all computation and communication tasks are viewed as a collection of components that interact with each other via two kinds of messages — commands and events. Each component also encapsulates a collection of tasks that can be posted to a task-queue for execution. By considering a collection of system-level primitives that are commonly used in distributed automation systems [3], TinyOS components were designed to regulate the behavior of modular units in a MicroFactory. The MicroFactory is described in more detail in Chapter 3. The approach was evaluated by implementing the designed components on a collection of Mica2DOT motes and comparing the performance achieved with that achieved using a time-triggered design for the same collection of system-level primitives.

1.1 Contributions

The contributions of this thesis are:

1. TinyOS Components for discrete assembly operations,

2. A component-based event-triggered template and method for designing networked embedded automation systems in the discrete assembly domain, and

3. Empirical results for commonly used primitives based on experiments with a physical testbed.
1.2 Overview of this Thesis

Chapter 2 presents a brief overview of TinyOS and the design paradigms. Chapter 3 describes the MicroFactory module units and how the motes were interfaced with these units in the MicroFactory. Chapter 4 presents the design template and the components designed for this investigation. Chapter 5 presents results of experiments and finally Chapter 6 presents the conclusions of this investigation.
CHAPTER II

BACKGROUND

This chapter presents details of the mote platform used in this investigation, the TinyOS environment, automation systems, and design paradigms for networked embedded systems.

2.1 Motes

A mote is a tiny, resource-constrained, microcontroller that is integrated with a low power wireless transceiver. It is typically used to monitor a variety of sensors, including those for light, position, sound, temperature, and humidity. Motes gather the readings from these sensor devices, store intermediate values, aggregate values, and forward them to other motes or monitoring stations in the network for further processing. In networked embedded automation systems, motes also compute values for their local actuators.

Motes have been used in biological, ecological and industrial applications, to name a few. Several mote platforms, such as the Mica2 [10, 11], MicaZ [10, 12], Mica2DOT [13, 14], Telos [10], TinyNode [15], and Firefly [4, 16], are reported in the sensor networks literature. The first five of these typically execute a reconfigurable, component-based, event-triggered operating system and application framework called
TinyOS [10]. The Telos and TinyNode platforms both use a Texas Instruments MSP430 microprocessor with 10K RAM, 48K ROM and 8K RAM, 92K ROM respectively. The Firefly uses an Atmel ATMega1281 microprocessor with 8K RAM and 128K ROM, and has an available mini-SD card slot. The experiments in this investigation were carried out on the Mica2DOT motes which are equipped with an 18-pin expansion connector for all major I/O signals, and use the Atmel ATMega128L microprocessor with 4KB of RAM and 128KB of ROM. Both the Mica2 and Mica2DOT motes use the CC1000 radio which operates on the 315, 433, 868, and 915 MHz bands, and uses no packet buffer for transmitting or receiving [10, 11, 14]. The CC1000 radio has a maximum data rate of 38.4Kbaud Manchester encoded and transmits with a current draw of about 25mA to 27mA when transmitting at maximum power.

The MicaZ, Firefly, and Telos all use the Chipcon CC2420 IEEE 802.15.4 standard-compliant radio transceiver with a 128 byte packet buffer and maximum data rate of 250Kbps [16]. The radio used on the TinyNode is the Semtech SX1211 which operates in the 863-870 MHz, 902-928 MHz and 950-960 MHz ranges, has a packet buffer of 64 bytes, and has a maximum data rate of 200Kbps [15].

2.2 TinyOS

TinyOS is a tiny microthreaded component-based operating system platform that was designed as a foundation for research in sensor networks [9]. It was designed for the implementation of event-centric, concurrent, low-power applications. Applications are constructed using fine grain software components that can be selectively replaced
or have new components interposed within the stack that holds them, in order to allow for new functionality.

*Components*, some of which abstract hardware by encapsulating their functionality in software, are the basis for which TinyOS programs are built upon, and *interfaces* are used to connect these modular components with one another. Within the structure of these component-interface relationships are commands and event-handlers that define the framework for various intended applications. Sensing, actuating, routing, packet communication, and data storage, are some of the standard abstractions for which the operating systems provides interfaces and components. There are generic components also made available that can be used multiple times within a single application.

In addition, the TinyOS provides its users with the ability to define desired components and interfaces that suite their specific applications, for situations where these components or interfaces were not already made available. With only one stack, TinyOS is non-blocking, enabling it to maintain high concurrency. Non-preemptive FIFO scheduled abstractions that can be posted by components, for which a scheduler will assign to the stack to run at a later time, are called *tasks*. Tasks are used to support large complex computations, but do not exclude simple ones such as I/O operations, and starting or stopping timers.

The design of TinyOS was driven by four broad requirements [9]:

1. Limited resources: Due to the limited physical resources that motes possess, Moore’s Law had to be applied during the design process for TinyOS.
2. Reactive Concurrency: Nodes sample their environment for changes or events. These events mostly require real-time responses, therefore, an approach that reduces potential bugs while respecting resource availability and timing constraints was needed.

3. Flexibility: There needed to be support for a variety of hardware and applications while remaining application specific.

4. Low Power: The operating system needed to be able to address extremely low power operation as well as provide a great deal of flexibility in duty cycle strategies and in power management.

The simplest application for the TinyOS platform uses 400 bytes as opposed to the C runtime predecessors which require just over 1KB.

For these reasons, TinyOS was a natural choice for this investigation.

2.2.1 Reliable Communication

To reliably communicate between nodes in the TinyOS environment, it is important to resolve media access control. To cope with the unreliable communication environment in the low power regime, there is a need to add-in additional software support.

Media access control is an important activity that must be implemented to enable multiple transceivers to share a common network medium. In wired networks such as Ethernet, collision-detection based media access control is viable. In this approach a sending node reads back the medium while transmitting a message. If the node senses something different than what it is sending, it assumes that a collision
has occurred and initiates a recovery protocol. Unlike wired networks, collisions in wireless networks occur at a receiver node and not a sending node, hence collision-detection is not a viable strategy.

The two common approaches used for wireless networks in the literature [17] are based on collision-avoidance and time-division multiple access. Collision-avoidance strategy is typically combined with a carrier sensing protocol such as the well-known Carrier Sense Multiple Access (CSMA) protocol. When it is integrated with a collision-avoidance strategy, it is known as CSMA/CA. In this approach, a node that transmits a message implements a collision avoidance strategy in which it delays the time at which it actually sends the message by a small but random duration of time called the backoff time. In the CSMA/CA protocol implemented in TinyOS, the backoff time depends on whether or not there is congestion in the network. Before transmitting any message, the node first listens to check if it can receive any valid message from the network for a fixed duration of time. If such a message is received, the node assumes that there must be more communication underway and hence it backs off from sending for a longer duration of time. If no such message is received, then the backoff time is shorter. These times are typically determined by invoking the \texttt{rand} function in a programming language like C.

The other approach for media access control is commonly described as time-division multiple access (TDMA). In this approach, time is viewed as a sequence of discrete slots. Each node is assigned to transmit or receive a message of fixed size, or remain idle in each slot. These assignments are carried out by offline analysis and
must be revisited whenever the topology of the system is changed. Because the focus of this investigation was on an event-triggered approach, the CSMA/CA approach was used.

To achieve this reliability, a simple strategy based on acknowledgments and retries was implemented. It is important to note that because wireless communication is in its nascent stages, especially in the factory automation domain, the message loss rates are rather high [18, 19, 12, 20]. For example, in a lab-based experiment 12% loss rates were observed. Modern techniques for modulation such as CDMA and OFDM are typically not used in such systems because of the higher costs and power requirements. For these reasons, it was necessary to implement a retry-based mechanism for achieving reliable communications.

Any node that transmitted a unicast message waited to receive an acknowledgement from the intended receiver node. The maximum wait time for and acknowledge message in TinyOS is fixed at 18 milliseconds based on experimental studies. If the sending node did not receive an acknowledgment, the original message was presumed to be lost and was sent again. After five retries, the sending node backs off for a long duration of about three to five seconds and then restarts the communication process. Assuming a probability of failure of 0.12, it is easy to see that the probability that five successive retries fail is decreased to 0.000025. Thus, despite persistent loss of messages, the communication between nodes is fairly reliable when retries are used.
2.3 Automation Systems

Automation systems are distributed real-time systems that regulate the coordinated operations of humans and machines in a safe, predictable, and reliable manner. These systems use thousands of sensors to detect phenomena in the physical world and regulate operations via actuators. The logic for computing values of actuators executes on specialized industrial controllers [5].

To achieve predictable operations, most programmable automation controllers use the periodic scan architecture, depicted in Figure 2.1. At the beginning of the scan, the environment is sensed and copied into an input image table to ensure that the values do not change during evaluation. Next, output values are computed and held in the output image table until the end of the evaluation. After the evaluation is complete the output image table is copied to local actuators. These actions repeat periodically. Before each scan, values of the output image table are restored to their default state.

Networked embedded automation offers significant opportunities such as dynamically configurable system topologies, readily deployable wireless sensor nodes, and improved capabilities for safety, monitoring and diagnostics. For example, a collection of wireless sensors used to augment an existing automation system can detect deviations from the expected conditions as soon as possible, and perhaps prevent unnecessary operations on a defective part or deflect the defective parts to a repair or recycle stream of flow. In applications such as drilling rigs, early monitoring of
operational conditions can significantly reduce the cost of operating and maintaining the system.

2.4 Design Paradigms

There is significant anecdotal evidence that the real-world is event-triggered and predictable systems are time-triggered [6]. Fundamentally, time-triggered and event-triggered architectures are two different paradigms for designing systems that interact with the physical world. All sensing, actuation, computation, and communication for a time-triggered system occur periodically at predetermined instances of time. Typically, one designs a schedule and the implementation faithfully realizes this schedule. In contrast, in event-triggered systems activities occur in a reactive manner in re-
sponse to stimulus from the environment and because of causal dependencies between components of the system [6, 21, 22].

A real-time system is partitioned into two different but equally important parts. One is the controlling system, and the other is the object being controlled by that system. Real-time deadline requirements need to be satisfied by the controlling system, i.e., the system responsiveness to stimuli observed from the controlled object has to be such that significant changes in the state of the object can be promptly detected. The proper implementation of this responsiveness allows for temporal accuracy to be maintained. As a consequence, if these significant changes are not detected, real-time system control is unattainable [6].

Comparison between ET-systems and TT-systems is usually based on metrics of predictability, resource utilization, testability, extensibility and assumption coverage in [6]. In TT-systems, processing activities and node communication are periodically aroused at predetermined instances in time that should match the real-time dynamics of the object being controlled. The fundamental concern of such a system is with the rapid propagation of the current state of all the subsets of the object to all the nodes that constitute that system. For the ET-systems, communication and processing activities are activated whenever significant state changes are observed [6]. Collisions occur as a result of these dynamically scheduled communication instances of ET-systems. Consequently, either message retransmissions are required or various arbitration techniques are employed. Either of these situations result in the delayed transmission of at least one of the messages being sent over the
network. This is characteristic of all event-triggered protocols [23].

Events are separated into two categories, predictable events that are anticipated, and chance events that occur at random. The flow control for events is implemented differently between TT-systems and ET-systems. TT-systems are more predictable than their event-triggered counterparts. Due to their periodic sampling of the state of an observed object, TT-systems handle chance events based on the sample rate at which they are observed. This allows for a constant flow control in the system. Conversely, ET-systems use a buffer mechanism to handle events. Multiple significant events happen at the same time causing an (event showers). Due to the processing capacity limitations of the system, sometimes these events cannot be processed simultaneously. The buffer mechanism properly manages such situations.

The temporal accuracy of the system is dependent on the wait time an event experiences while in the event buffer. Scheduling strategies are also different for the two paradigms. Activities are scheduled dynamically in ET-systems due the nature of such systems, however, in TT-systems activities are preplanned before system deployment. These activities are then polled, and the responsiveness of the system is determined by the granularity of the periods of these polls. Although runtime resources are required for dynamic task schedule execution in ET-systems, only tasks that have been activated need to be scheduled. Better average resource utilization under lower load conditions is realized in ET-systems because actual execution times of their dynamically scheduled determine the availability of these system resources to other tasks to be scheduled. Tasks in TT-systems are statically scheduled offline.
before deployment, and resources are assigned to tasks in such a manner to allow for
the maximum duration any particular task can run for to be set as the time allocated
for that task within the schedule, i.e. peak load demands determine the structure of
the schedule. It is easy to see how in conditions where system load is lower, resource
utilization in TT-systems is not as efficient as in their event-triggered counterpart.

In contrast to TT-systems that allow for more systematic testing, due to the
vastly larger number of execution scenarios that can occur, testing in ET-systems
is more cumbersome and less systematic. ET-systems also perform error detection
in their communication protocol at the sending node of a message. This is based
on a timeout set for waiting for an acknowledgment from the intended receiving
node of that message. For situations where several events occur concurrently, a
congestion control mechanism needs to be implemented for such systems. In contrast,
TT-systems implement fault-tolerance at the receiving node because of the relative
awareness these nodes have of the periods messages are meant to arrive due to the
static schedule that governs the operations of these nodes.

ET-systems are simpler to design and implement. These systems also utilize
the available resources more fully than their time-triggered counterparts. While there
is a significant body of knowledge for analyzing time-triggered behaviors at the level of
a single node, little progress has been made in the realm of distributed asynchronous
systems. However, TT-systems are more predictable than ET-systems. In addition,
it is not known if TT-systems can scale to large number of nodes. Thus, there is con-
siderable interest to understand how elements of event-triggered and time-triggered
architectures can be combined to realize better execution performance for a broad spectrum of applications.[6]
CHAPTER III
MICROFACTORY

The MicroFactory at the Complex Engineered Systems Lab is a physical simulation of an assembly operation line. There are modular units that represent typical operations that are performed in manufacturing plants, such as the ones used for automotive assembly. The sensors, and actuators, are discrete in the sense that these devices present, and receive, digital values at their interface; thus a microcontroller can interact with these devices, i.e., sense and actuate, by reading and writing digital values to one of its ports. This chapter describes the MicroFactory used for this investigation in more detail.

3.1 Modular Units

The modular units are electrically independent of one another. Each unit performs a pre-defined task; a collection of such units can be arranged to represent typical workflows in an assembly line. Some of the modular units are shown in Figure 3.1 and Figure 3.2.

The input bin injects parts into the assembly line. There are two part bins each with its own pusher-arm. The position of the parts on the bins, position of the
Figure 3.1: The Input Bin (a) pushes parts on to a conveyor to initiate assembly operations. The Workcell (b) pulls a part off the conveyor. The turntable turns the parts successively through three workstations and returns the part to the conveyor.

pushers, and position of parts on the conveyor is detected by proximity sensors that are mounted on the units. The output bin pulls parts from the assembly line at the end of the process. The paint station comprises a conveyor, a paint arm, limit switches and a part position sensor is used for simulating paint operations. The workcell unit can process multiple parts simultaneously at its different stations - namely, a horizontal mill, a vertical mill, an air chip station, and a load-unload station. This unit also has a conveyor for moving parts along, a turntable for rotating parts to the various stations, and an arm to place or remove a part from the turntable. Various IR-sensors and limit switches are used to sense parts and positions of the arms, and the table.

The units of such a MicroFactory and the conveyors that move parts between units can be regulated using automation controllers based on the scan architecture
Figure 3.2: The Paint Station (a) simulates typical paint operations on a part. The Output Bin pulls parts off the conveyor at the end of the assembly line.

discussed in the preceding chapter. In this investigation, a networked collection of motes were used to regulate the operations of the MicroFactory.

3.2 Primitives for Networked Embedded Automation

Because of the diverse nature of automation tasks, it is not feasible to evaluate the performance of the networked collection of motes on every possible arrangement of the units of the MicroFactory. To work around this practical limitation, an approach that is consistent with the state of the practice was used. In this approach, the common tasks typically encountered in an assembly line are organized as a collection of primitives [1, 3]. Each primitive represents an operation that is either performed locally on some mote or performed in coordination with activities on another mote. The only means for coordinating the operations is via communication.
3.2.1 Local Response

Figure 3.3 presents an example of the *Local Response* primitive. This is the time required for the mote to sense local values, execute the logic necessary to compute values for local actuators, and present the computed values to the actuators. As shown in Figure 3.3, at time $t_0$ the unit is idle, at time $t_1$ a part is placed within the bin which is observed by a local sensor. This then causes the discrete actuator that controls the arm to energize and move the arm, pushing the part onto the conveyor. At time $t_2$ the part is fully on the conveyor, completing the response to the sensed stimulus.

![Figure 3.3: An example of a local response primitive.](image-url)
3.2.2 Remote Response

This primitive captures the scenario where a mote must energize a local actuator based on a non-local sensor value that is received over a single-hop communication link, i.e., from another mote from which it can receive messages. The non-local value is sent from the other motes using a \textit{push} model, i.e., the receiving mote does not initiate a request for the data. This can be seen in Figure 3.4 where a message is sent from one node to another when a local state changes. In the figure, at time $t_0$ a sensor is triggered on node $n_1$, which then sends a message to node $n_2$; $n_2$ then turns on an actuator using its local logic, in response to the message.

Figure 3.4: An example of a remote response primitive [1].

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{remote_response.png}
\caption{An example of a remote response primitive [1].}
\end{figure}
3.2.3 Request/Response

The request/response primitive represents situations where a node uses local logic to energize a local discrete actuator resulting from the response it received from a neighboring node one hop away about information it requested about non-local values from that neighbor. Figure 3.5 depicts this primitive; node $n_1$ requests information from node $n_2$; $n_2$ responds by sending a wireless reply. When $n_1$ receives this response, it energizes a local discrete actuator.

![Diagram of request/response primitive]

Figure 3.5: An example of a request/response primitive [1].

3.2.4 Publish/Subscribe

The publish/subscribe primitive builds on the remote response primitive. A dedicated node, called the publisher broadcasts messages to several receiver nodes (subscribers)
either periodically or based on some local event at the publisher node. This primitive is depicted in Figure 3.6. A publisher node \( n_1 \) detects a local sensor change, sends a message to two subscriber nodes \( n_2 \) and \( n_3 \); these subscribers respond by energizing local discrete actuators.

![Diagram of publish/subscribe primitive](image)

Figure 3.6: An example of a publish/subscribe primitive [1].
3.2.5 Multihop Remote Response

Figure 3.7 depicts the multihop remote response primitive. This primitive is similar to the remote response primitive, except that the communication message is relayed by one or more intermediate nodes before it reaches the target node.

![Diagram of multihop remote response primitive]

Figure 3.7: An example of a multihop remote response primitive [1].
3.2.6 Multihop Request/Response

Figure 3.8 shows an example of a multihop request/response primitive. This primitive is used to represent the multihop version of the previously discussed request/response primitive. Here, a node uses local logic to energize a discrete actuator based on the response received from a different node via multiple hops.

Figure 3.8: An example of a multihop request/response primitive [1].
3.3 Interface to Mica2DOT Motes

The IO pins on the Mica2DOT motes were configured to accept and send digital signals from and to the MicroFactory. The sensors on the MicroFactory present 5 VDC signals. The actuators require 12 VDC and the DC motors that move various parts on the MicroFactory require between 300 mA and 800 mA depending on the number of parts being moved on the conveyor. The digital ports on the motes were configured to provide 3.3V and no more than 30 mA could be drawn from each pin. For these reasons, interface boards that translated the signal levels and provided optical isolation between the motes and the MicroFactory were used. These boards were already available in the Complex Engineered Systems Lab.

The number of IO pins available on each mote constrained the choice of how many motes were required for each unit. There are 14 expansion pins on each Mica2DOT mote and 12 of these can be configured as either input or output pins. The other 2 pins can only be used as input pins. Each interface board supported 8 pins that could be used either for input or for output. Thus, a separate pair of interface boards were used for the input bin, the output bin, and the paint station. The Workcell unit was more complex than the other units and required a pair of boards for each of its stations; consequently, three Mica2DOT motes were needed for this unit.

\[1\] When a unit requires multiple motes because the number of IO pins in each mote is limited, one of these motes was designated as the communication mote. This communication mote represented the local collection of motes as a single unit at the system level.
CHAPTER IV
COMPONENT-ORIENTED DESIGN

This chapter describes the TinyOS components and interfaces, the design process, and the template used in this investigation. The components in TinyOS are interleaved and form a complex web of component-interface relationships. A list of all the components in TinyOS can be obtained from the TinyOS documentation [9]. The important components necessary for understanding the design are discussed in this chapter.

The main components necessary are TinyScheduler, MainC, Timer, and the components that enable communication between the motes. The descriptions and relationships between these components are excerpted from [9]. The main module components that abstract the functionality of each module unit of the MicroFactory were developed as a part of this investigation.

4.1 TinySchedulerC Component

The TinySchedulerC component is responsible for initializing and running all the tasks in a unit. This component provides two interfaces, namely Scheduler and TaskBasic, via which other components can interact with it. The Scheduler interface provides functions for initializing tasks, running tasks and looping task queues,
while the `TaskBasic` interface is the simplest TinyOS task class that are run in a FIFO manner. The `TaskBasic` interface is used to post a task to the `Scheduler` and captures the event that is signaled when a task starts to run. Figure 4.1 shows the component relationships of the TinyOS `TinySchedulerC` component.

![component relationships diagram]

Figure 4.1: `TinySchedulerC` component relationships.

4.2 MainC Component

The system interface for the TinyOS boot sequence is the `MainC` component. The boot sequence is the precise sequence of instructions that are executed when the mote is powered on. This sequence is pre-designed to ensure that the mote is ready to execute a designated program immediately upon completing the boot sequence.
The Scheduler and the underlying hardware were interfaced with this component. Figure 4.2 depicts the component-interface relationships for the MainC in TinyOS.

4.3 Timer Components and Interfaces

The Timer components used for this application are abstractions of the generic TimerMilliC component provided by TinyOS. This is a standard timer that provides interrupts that are one millisecond apart in TinyOS. This timer can be extended by specialization for use in any application, as long as there is adequate RAM to support
the extension. Timers are necessary to keep track of the duration of certain tasks such as (a) wait time before a message is resent if it was not acknowledged and (b) duration for which a conveyor belt must be energized to ensure part transfer. Figure 4.3 shows the relationships in TinyOS that involve TimerMilliC.

```
Timer

Timer<T Milli>

TimerMilliP
```

Figure 4.3: TimerMilliC component relationships.

4.4 Communication Components and Interfaces

The components used to support communication in this investigation include: ActiveMessageC, AMSenderC, AMSnoopingReceiverC, CC1000CsmaRadioC and RandomC. The interfaces configured to work with these components were:

1. **Packet**: For defining the structure of a data packet to be sent.

2. **AMSend**: Interface used to send a predefined data packet.

3. **Receive**: Interface responsible for the reception of a data packet.
4. **PacketAcknowledgement**: Interface responsible for requesting and providing acknowledgements for sent and received packets respectively.

5. **Random**: Interface used to generate the random backoff time used for resending messages that were not acknowledged by an intended receiver.

The TinyOS documentation provides additional details on how these components must be configured to work collectively.

4.5 The Main Module Component

This component is a user-defined component that encapsulates all the TinyOS components necessary for a module unit. Each MicroFactory module unit has a unique main module component. Figure 4.4 shows how this component is connected with other components that TinyOS provides.

4.6 Tasks

Tasks in TinyOS are basically functions where the main computation that results in sensing and actuation actions are performed. There are several tasks designed and implemented to achieve the behavior for each unit of the MicroFactory. For instance, the input bin unit that has two pusher arms, two bins and a conveyor. A simple task on this unit is the `ram1Task()`. This task contains logic that is used to move one of the arms on the unit back and forth to push a part onto the conveyor as needed.
Figure 4.4: Main module component relationships.

The following structure of the code for this task, written in a readable form, is an example.

```
ram1Task() {
  if(arm is home, arm status is ready to push, no part is blocking arm
    & a part is present in bin) {
    set arm status to pushing
    move arm out
  }
  if(arm is fully pushed) {
    bring arm back home
    set arm status to returning
  }
```
if(arm is fully returned) {
  stop arm in place
  set arm status to {ready to push}
  task is finished
}
if(task is not finished) {
  repost task
} else if (task is finished) {
  post next task
}

The tasks used in this investigation were classified as either an initialization task, a processing task, or a communication task. Some tasks, such as marking a message to be sent or sending a marked message, have simple logic and need to be executed only once. Other tasks that are used to sense or actuate events post themselves to the Scheduler until some external conditions are satisfied. Nominally, when a task completes its operation, it posts the next task in line to start its specific operation. This posting of successive task continues until the final task is posted. For example, the input bin cycles through its task set in the following order:

1. *Initialization task set:* These tasks ensure that all arms on the unit are in the home position and ready for parts to be put in the bins. During initialization, special communication messages are sent to configure the addresses of the neighboring nodes.

2. *Processing task set:* These tasks guide the performance of physical actions on the parts placed in the unit.
3. **Communication task set**: These tasks are necessary for unit-to-unit communication for transferring parts.

Tasks that comprise the processing task set are posted sequentially until the final processing task is done. Upon completion of the last task in this set, the communication task set is executed. The processing task set is what gives a module unit its specialized definition. For example, the paint operation, or milling operation as performed for the painter and workcell units, respectively, are processing tasks. However, each unit follows the same communication protocol and therefore their communication task sets are identical. The only difference between one mote and another is in the neighbors with which they communicate. The implementation of this neighbor-neighbor communication is addressed in the following design template and implementation sections of this chapter.

4.7 Design Template

A template was designed to simplify the design and implementation. This template was tailored to suit an individual module unit and can be used in a similar manner for future implementations. The key idea in designing the template was that the processing tasks are specific to each unit. A uniform notion of state was defined for any unit in the system. A state is an unambiguous condition that could be recognized in the program of a unit based on the values of internal variables or sensors. Five module unit states were defined and are depicted in Figure 4.5. In each state, a unit
was allowed, by design, only to perform a pre-defined set of actions. These design decisions were implemented in code.

Figure 4.5: 5 possible module unit states for any unit that constitutes the MicroFactory.

The output bin does not implement the *ready to transfer part* and the *transferred part* states. This is because it was assumed that the output bin is always ready to accept a part and never has to transfer a part to an adjacent unit. The only information required by an arbitrary unit $A$ from another unit $B$ is to know when $B$ is in the *ready for new part* state when $A$ is in the *ready to transfer part* state. This interaction was the only one found to be necessary for the implementation of proper modularization of the MicroFactory.

The design template provides the framework for the main module component, the communication components, possible module unit states, timer components, and
tasks that are necessary for node-to-node communication. The template also config-
ures IO pins available on the Mica2DOT motes and connects these pins to the appro-
riate TinyOS components. This template is depicted in Figure 4.6 and Figure 4.7
and these two components, namely \texttt{shellP} and \texttt{shellAppC}, collectively comprise the
template.

Figure 4.6: An illustration of the structure of the first part of the template: \texttt{shellP}
component.
To use this template one must:

1. identify what IO pins need to be used for a unit,

2. implement the code for the tasks that represent the processing tasks specific to that unit,

3. anchor these tasks with the possible module unit states already set up by the template,
4. and invoke these tasks and other communication tasks at appropriate times based on the desired functionality.

4.8 Implementation

Figure 4.8 shows an example layout of the units selected from the MicroFactory used for this investigation.

A separate mote was associated with each unit. The mote was programmed using the design template discussed in the preceding section. Every mote had a unique identifier (id) and each mote also had a list of id’s for its neighboring motes. In this investigation, a central topology-control mote broadcasted this information to all the motes. Several decentralized protocols in the literature for topology control and maintenance can also be incorporated in future experiments on this testbed [24].

After a unit, i.e., the mote associated with the unit, was initialized, it sent a message to the topology-control mote requesting its neighborhood information. The
topology-control mote then sends messages back to all the units with the requested information.

Since parts can only transfer between units that are physically adjacent, a unit only needs to know its immediate neighbors to enable part transfers. This way, units can be interchanged since any arbitrary unit $A$ really only cares if there is another arbitrary unit $B$ ready to accept a part when $A$ is done processing it, and not the functionality of the unit $B$.

![State machine for the input bin unit](image-url)

Figure 4.9: State machine for the input bin unit.
Figure 4.10: State machine for the painter unit.

Figure 4.9 shows the finite state machine for the input bin, Figure 4.10 and Figure 4.11 show the state machines for the painter and the output bin, respectively. The CSMA/CA protocol with acknowledgment that was discussed in Chapter 2 was used for this investigation. Figure 4.12 depicts this protocol. Every message sent must be acknowledged by the receiver. If such an acknowledgment is not received, the sender assumes that the message is assumed to be lost and is resent.
making 5 such retry attempts, the sender waits for a 3-second interval and restarted the communication. This continued until the message was acknowledged.

When a unit is ready to send a part to its upstream unit, it sends a message to that unit to request permission. If the message is acknowledged, the sending unit goes into a slow polling state waiting for the upstream unit to send a message saying it is now in the ready for new part state. Upon reception of this ready for new part message, the sending unit transfers the part by energizing its conveyor actuator.
and sends a message saying the part has been transferred. If however the upstream unit was not ready to receive a new part, the sending unit periodically polled and re-requested permission to transfer the part.

Figure 4.12: Stage for the unit-to-unit communication protocol implemented in the design.
The design template and the communication described in this section provided an efficient framework to develop the TinyOS components needed to evaluate the performance of the event-triggered design and implementation of the primitives for networked embedded automation.
CHAPTER V
EXPERIMENTAL RESULTS

The experiments reported in this chapter were designed to quantify latency and jitter achieved when using a event-triggered design for the primitives of networked embedded automation.

5.1 Experimental Approach

The primitives described in Chapter 3 were used as the basis for all the experiments. Sensor events were triggered by a Pulse Width Modulated (PWM) signal that was generated by a microcontroller not involved in the experiment. Latency was measured as the time difference between a sensor event and a corresponding actuation event. The time stamps were obtained using a digital oscilloscope that was connected directly to the sensor event and the actuation event. For each value of latency desired, 120 samples were collected and the average of these values are reported here. Jitter reported is the standard deviation of the latency over all the 120 samples.

To introduce uncertainty in the operating environment of the motes, background communication was introduced by programming other motes, which were not involved in the experiment, to communicate with each other periodically. These motes also used CSMA/CA with random backoff. To increase the disturbance, more
motes were made to communicate in the background. The delays caused because of this background communication in the messages between motes that regulated the units of the MicroFactory are included in the data.

5.2 Local Response

Figure 5.1 presents the results of the local response primitive. Notice that as the number of sensor events increase, the latency and the jitter do not vary significantly. This is because of the large difference in speeds between the electronic processing in the motes and the physical transfer of parts that trigger sensor events. For this reason, experiments with other primitives were limited to a single sensor event.

![Figure 5.1: Data gathered from the local response primitive experiment with no background communication](image)

Figure 5.1: Data gathered from the local response primitive experiment with no background communication
Figure 5.2 shows that when the background communication increases, the maximum latency increases. Since the wireless medium is an inherently broadcast medium, the receiving mote must first receive a packet and then determine that it was not the intended target based on the id in the received packet. However, because of the large difference in speeds explained above, the impact was not significant.

![Figure 5.2: Data gathered from the local response primitive experiment with background communication incorporated](image-url)

Figure 5.2: Data gathered from the local response primitive experiment with background communication incorporated
Figure 5.3: Data gathered from the remote response primitive experiment

5.3 Remote Response

Figure 5.3 presents the results obtained from performing the remote response primitive. Two motes $n_1$ and $n_2$ have sensor $S$ and actuator $A$, respectively. $S$ is triggered by an external input; $n_1$ then sends a message to $n_2$ which responds by energizing $A$. The cause of jitter in the system was likely due to the variations in the time a mote is able to respond to messages received over a wireless link. To provoke these variations, background motes were gradually added to the system, increasingly from 0 to N, where $N = 7$. This way, the interference caused for $n_1$ and $n_2$ was significant. The latencies reported reflect the average of the time $(t_{A\text{-energize}} - t_{S\text{-triggered}})$ for all the samples. The jitter observed varied between 10ms to 70ms. The reason for
this is because of the randomness of the backoff values generated by a mote before it retries to send a message that was not acknowledged by its intended receiver. These random backoff waiting intervals are characteristic of the MAC protocol implemented for Mica2DOT motes in TinyOS.

If more background motes are interfering with \( n_1 \) and \( n_2 \)’s communication, the possibility for lost packets due to channel contention increases accordingly. However, this is not an indication that there is a linear relationship between jitter and the number of communicating background motes. Furthermore, the impact of these randomly generated backoff times for message retry is likely causing the gap in range of the values obtained for max latencies (when higher random backoff times are generated), and minimum or average latencies (when smaller times are generated), as observed in Figure 5.3.

5.4 Request Response

The experiments for the request/response primitive were performed in a similar fashion as with that for the remote response primitive. For consistency, the sensor was also triggered at the same rate as with previous experiments. The difference however, is that when \( S \) is triggered \( n_1 \) sends a message to \( n_2 \) requesting permission to energize \( A \). \( n_2 \) sends a reply back to \( n_1 \) granting the request, and upon \( n_1 \)’s reception of \( n_2 \)’s response, the actuator \( A \) is energized. The cause for the jitter experienced is likely the same as with that for the remote response primitive experiment. As can also be seen from Figure 5.4, the latencies observed are in accordance with two way
communication when considering the remote response primitive scenario as its one way communication counterpart.

5.5 Publish Subscribe

Figure 5.5 shows the data for the experiment with the publish/subscribe primitive. The experiment was performed in the following manner: \( n_k \) motes were used for the different variations, where \( k \) increased gradually from 2 to a maximum of 8 for each successive variation of the experiment. A sensor \( S \) is externally triggered periodically on \( n_1 \), who broadcasts a message to the responding motes for every trigger of the sensor \( S \). \( n_1 \) then waits for a response from the responding motes, for which actuator
A on $n_1$ is energized upon $n_1$’s reception of all responses from the remaining $k - 1$ motes.

Unlike the results from the experiments addressed so far, there is a more prevalent positive relationship between the latencies experienced by the system and the number of motes simultaneously communicating. That is, the more motes in the system that receive the broadcast message at the same instance and attempt to respond, the more packet collisions that occur. As a result, more backoff times are experienced, and $n_1$ has to wait for longer to receive all the replies from responding motes.
Figure 5.6 shows the performance of the multihop remote response primitive. This experiment involved a sender node $n_1$ and a receiver node $n_k$. At time $t_1$, node $n_1$ sends sensor information over multiple hops to $n_k$. Upon $n_k$'s reception of the message at time $t_k$, it energizes a local actuator. The latency measured for this experiment was the time interval $t_k - t_1$.

Figure 5.7 presents that of the multihop request/response primitive. The multihop request response experiment involved a node $n_1$ initiating a request for sensor information at time $t_1$ from a node $n_k$ over multiple hops. $n_k$ responds back over the multiple hops to $n_1$. Upon $n_1$'s reception at time $t_{2k}$, it energizes a local
actuator. The latency measured for this experiment was the interval $t_{2k} - t_1$.

As expected with both experiments, there is a noticeable linear rise in the latencies observed for each successive increase in the number of hops in the system. Also not surprising, the latencies observed in Figure 5.7 relatively scale up from those in Figure 5.6. Comparing both figures, the trends in Figure 5.7 are typical for two-way multihop communication, considering Figure 5.6 to be the trends observed for one-way multihop communication in the system. However, the jitter experienced for both experiments does not follow the same linear pattern the latencies do. The maximum jitter experienced from all the experiments conducted herein was observed around 125ms for the multihop request/response primitive, which is quite appealing.
CHAPTER VI

CONCLUSION

This thesis presented an event-driven approach for designing networked embedded automation systems based on the TinyOS framework. In this approach, a collection of motes were used to regulate the behaviors of a MicroFactory, which is a physical simulation of a typical assembly line. Any implementation that is based on TinyOS involves components, interfaces, tasks, and events. A design template was presented to simplify the development of these entities. The tasks that were required to execute on the motes were classified as initialization tasks, processing tasks, and communication tasks. It was shown that the design template offered a common framework that could be tailored to suit the needs of individual modular units of the MicroFactory by modifying the processing task set.

Because the modular units of the MicroFactory could be arranged in hundreds of different ways, it was infeasible to evaluate the performance of the approach for all possible arrangements. Instead, the investigation used an approach that was guided by typical scenarios in which motes must coordinate their operations to achieve system-level behaviors. Six primitives, namely Local Response, Remote Response, Request Response, Publish/Subscribe, Multihop Request Response, and Multihop Request Response were presented. TinyOS components were developed for
each of these primitives using the design template and evaluated using latency and jitter as the two primary metrics. These results showed that while the latency and jitter showed some variability, the performance of the event-triggered system was compelling under the conditions of the experiments. Experiments also demonstrated that the MicroFactory units could be successfully regulated using these TinyOS components.

The results in this thesis offer initial encouragement for further investigation of the event-triggered approach. In the future, a more comprehensive evaluation of the event-triggered design that quantifies system-level reconfigurability and predictability must be carried out before this approach is recommended for adoption in existing and future automation systems.


