2008

A Software Architecture for Adaptive Modular Sensing Systems

Andrew C. Lyle
Western University

A Software Architecture for Adaptive Modular Sensing Systems

(Thesis format: Monograph)

by

Andrew C. Lyle

Graduate Program in Engineering Science
Department of Mechanical and Materials Engineering

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Engineering Science

The School of Graduate and Postdoctoral Studies
The University of Western Ontario
London, Ontario, Canada

Abstract

In this thesis, a novel software architecture and knowledge representation scheme is described that facilitates the combination and reconfiguration of modular sensor and actuator components, termed *transducer interface modules* (TIMs), to produce flexible modular sensor systems. Each TIM provides a core sensing or actuation functionality. A composite sensor is able to automatically determine its overall geometry and assume an appropriate collective identity, and if reconfigured, may then assume a different identity to match its new geometry. In current practice, a fixed combination of sensors and actuators is typically utilized, and is tailored to a specific application. Such systems cannot be cheaply or quickly reconfigured to handle a change in process requirements. Domains that may benefit from easily reconfigurable modular sensing systems include flexible inspection, mobile robotics, surveillance, and even space exploration.

The software architecture is distributed, and is comprised of six layers where the implementation of each layer is encapsulated from the layer above, to which it provides service. The use of a distributed and layered architecture promotes scalability, mitigates against a single point of failure, and enables each layer to be easily implemented, modified, and debugged independently of the others. The modularization of the software architecture is further facilitated through the utilization of a pre-emptive real-time operating system, which enables the concurrent execution of the various software components specific to the architecture that implement the services provided within most of its layers.

Among the layers comprising the software architecture is a *virtual machine layer*, which implements a lightweight, architecture-specific version of Sun Microsystems' Java Virtual Machine that runs on top of the real-time operating system. The integration of a virtual machine enables the platform-independent *template algorithms* utilized at the *composition layer* to be written once and executed on any TIM irrespective of its underlying hardware architecture. These template algorithms are unique to this software architecture and provide intelligence to a set of heterogeneous TIMs, enabling them to collaborate and behave as a single entity termed a *logical module*.

The evaluation of the software architecture consists of performing multiple runs of two tests in which select sensors and actuators are associated with TIMs that are then allowed to interact in order to form a logical entity. The first test evaluates the behaviour of a logical module in which the constituent TIMs interact entirely through wireless communication. The second test evaluates the behaviour of a logical module in which the constituent TIMs are physically connected in various orientations, and interact through both wireless communication as well as through their physically connected faces.

In both tests, correct behaviour was exhibited. However, the performance and scalability of the architecture was somewhat restricted by the limited processing and memory resources present in the current implementation of the TIMs. The design of the software architecture facilitates easy portability between embedded platforms and scales with increasing hardware capability. Therefore, utilization of future TIM hardware variations possessing increased processing and memory resources will reduce the latencies introduced throughout the architecture and lead to tangible improvements in its performance.

Keywords: distributed software architecture, adaptive sensing system, modular sensing system, sensor module, actuator module, transducer module, logical module, position and orientation determination, virtual machine.

Acknowledgements

Many individuals have played a guiding and supporting role in the completion of this thesis, whom I would like to thank. Firstly, I would like to thank my supervisor, Dr. Michael D. Naish, for his expert guidance and patience, and for providing me with the opportunity to pursue graduate studies at The University of Western Ontario. This research programme has been an immense learning experience for me, and from him I have learned a great deal about the engineering process that I will be able to apply in my future endeavours.

I would like to thank my fellow Sensing and Mechatronics Systems (SaMS) Laboratory researchers Anita Jain, who coordinated with me on this project, for all her hard work and Christopher Ward for his insightful input. I am also grateful to the engineers in the Electrical and Computer Engineering Electronics Shop for their assistance in sourcing components, as well as their useful technical advice.

I am very grateful to Marie Wyatt, who played a key role in ensuring that my stay in London was a comfortable one. I am also grateful to my friends and family who maintained an active interest in my progress, especially to my parents for their steadfast support and financial assistance, and for always encouraging me to pursue my ambitions. I am also thankful to God for blessing me throughout my programme.

The financial support of the Natural Sciences and Engineering Research Council of Canada (NSERC) is also gratefully acknowledged.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Sensors and Actuators in Industry

Sensors and actuators have seen widespread utilization in many of today's industrial processes. These devices respectively convert physical phenomena to and from electrical signals for the purpose of measurement, tracking, and/or control by way of digital devices such as microcontrollers, programmable logic controllers (PLCs), and mainstream computers. In current practice, fixed combinations of sensors and actuators are typically employed, with each combination often deployed in a static orientation and tailored to fulfil a specific application.

In order to enhance accuracy and reliability in many such applications, multiple sensors are often directly or indirectly combined into composite entities. For example, a single camera is only capable of reporting a grid of values representing the intensity of incident light on its array of sensing elements. However, two or more cameras operating in tandem could, through sensor fusion, effectively form a sensor capable of depth perception. Sensors that detect different, but related, types of physical phenomena may also be combined to produce a new device that produces measurements that are more

accurate than either of its constituent sensors are capable of providing. An example of this would be the combination of a thermocouple and an infrared camera for the purpose of increasing the accuracy of sensed temperature.

The sizes of the transistors used in the implementation of microprocessors and other integrated circuits through *very large-scale integration* (VLSI) are becoming ever smaller, consistent with *Moore's Law* [1], due to advancements in semiconductor fabrication techniques. The sizes of sensors and actuators are also being reduced at an equally rapid rate due to advancements in *microelectromechanical systems* (MEMS) and *nanoelectromechanical systems* (NEMS) fabrication techniques. As a result of these technological advancements, it has become quite practical to combine sensing, actuation, processing logic, as well as transceivers that provide wired and wireless networking capability into a single monolithic device termed a *smart transducer.* With the ability to transmit information and locally execute algorithms independently, without depending upon a larger, static, and more powerful mainstream computer system, the potential for smart transducers to collaborate amongst themselves without any external influence in order to achieve a specific goal becomes worthy of consideration.

1.2 The Need to Combine Sensors and Actuators

Sensing systems designed to be operated in a static orientation and under controlled operating conditions cannot be cheaply or quickly reconfigured to handle a change in process requirements, such as in assembly lines where the product being assembled changes completely or is now required to be processed in previously unconsidered orientations. Instead of merely considering each existing sensor as a strictly self-contained device that is to be utilized in an exclusive scenario, or in tandem with others sensors, each sensor may be enhanced through physical combination with one or more actuators in addition to other

sensors, resulting in an *active* sensing device.

Combining a sensor with an actuator greatly enhances the ability of the sensor, which is now augmented with mobility and gains the ability to adapt to changing process requirements, such as monitoring non-stationary objects of interest. For example, a camera could be mounted on a rotational stage to form a panoramic camera with a field of view of 360 degrees, enabling it to track objects that move anywhere within a particular plane. The relocation of processing logic directly onto the hardware comprising a smart transducer allows such a composite sensing device to be completely self-contained, and scalable to even larger combinations of modules. The ability to combine diverse modular sensor and actuator components to produce flexible modular sensor systems facilitates rapid reconfiguration to suit any requirement, and is a technique that will prove useful in many modern applications. Examples of applicable domains include flexible inspection, mobile robotics, surveillance, and even space exploration.

1.3 Survey of Related Work

1.3.1 Logical Sensor Architectures

Modular sensing systems are often composed of a number of sensors and possibly actuators of diverse types. Enabling intercommunication and collaboration among these transducers, especially in a manner such that the sensing system is easily reconfigurable, is often problematic due to the various interfaces through which communication with inherently different types transducers must take place. For example, the interface through which readings are obtained from an analog transducer is often quite different from that through which readings are obtained from a digital transducer. Therefore, facilitating interoperability between the devices often requires solutions that are specific to the interfaces through which they communicate. Reconfiguring such solutions in large systems

in which numerous sensors and actuators are present can become unwieldy to the system user when transducers with newer interfaces are to be introduced and utilized within the systems.

One approach that aims to simplify the specification and assembly of multi-sensor systems, aspects of which are utilized in the design of the software architecture described in thesis, is the *Logical Sensor Specification* (LSS) [2, 3]. The LSS introduces the useful abstraction of a *logical sensor*, shown in Figure 1.1. The specification of a logical sensor facilitates the abstraction of the data produced by many different types of sensors into a uniform representation that is adhered to by all the sensors, and thus the internal hardware implementation of a sensor and the details of its data acquisition interface are completely hidden from the system user. As a result, dynamically reconfigurable modular sensing systems may be easily assembled through the composition of the logical sensor representations of its comprising sensors.

Since a logical sensor exposes a standardized interface, another strength of the logical sensor abstraction is that a logical sensor need not necessarily be associated with a physical entity. A logical sensor may be a software program that satisfies this abstraction interface, or may be a physical sensor that is augmented with processing algorithms implemented in software. Hierarchies of logical sensors may even be assembled in which multiple logical sensors are combined to form a composite logical sensor that appears to the system user as a single entity. Logical sensors higher in the hierarchy communicate with encompassed logical sensors lower in the hierarchy through the transmission of commands that are interpreted by a *control command interpreter*, and acquire data from the logical sensors through any of a set of programs each designed to obtain data in a unique fashion from a set of inputs. The ability of a logical module to selectively utilize different methods of acquiring data, through the use of its *selector*, is useful in the event of failure in the lower levels of the hierarchy. An example of a logical sensor hierarchy in

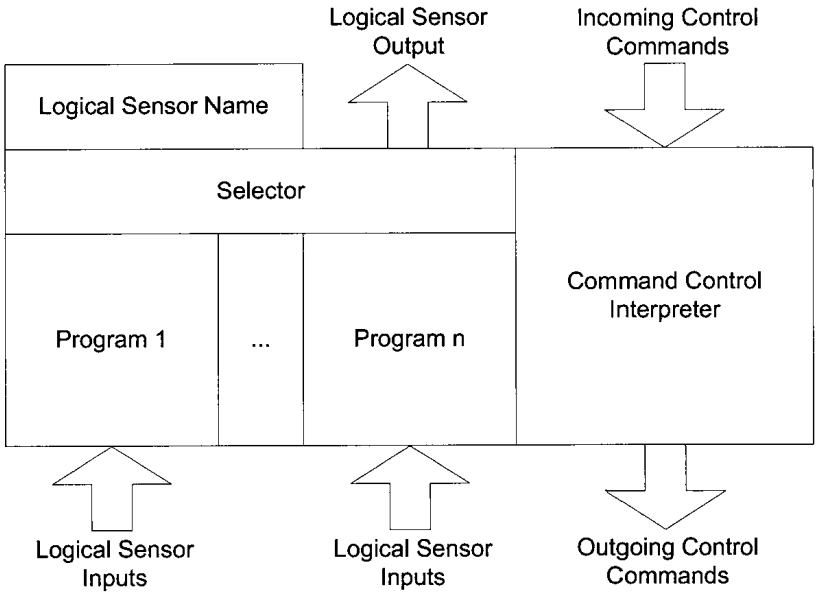


Figure 1.1: Logical sensor block diagram [2].

which two two-dimensional cameras and a stereo processing algorithm, or alternatively an active range camera, are used to implement a three-dimensional measurement system is shown in Figure 1.2.

Another existing architecture that provides similar benefits to that of a logical sensor hierarchy is that of *logical neighbourhoods* [4, 5]. In this architecture, sensor and actuator nodes are abstracted into uniform *virtual nodes* that may in turn be further abstracted into a composite collection termed a *logical neighbourhood.* Logical neighbourhoods appear as a single virtual node entity that may be further composed into larger neighbourhoods. Virtual nodes higher in the hierarchy transmit commands and data to nodes lower in the hierarchy through a wireless interface. The definition of virtual nodes and logical neighbourhoods is facilitated through template definitions written in the *SPIDEY* declarative language [4, 5].

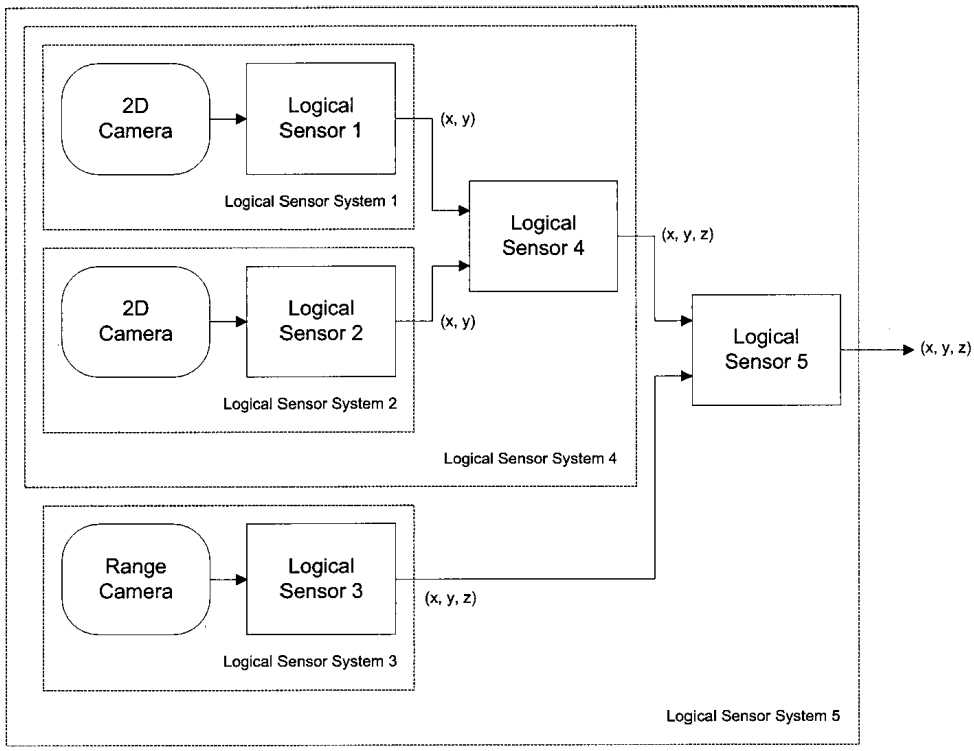


Figure 1.2: Sample logical sensor hierarchy [3].

1.3.2 The IEEE 1451 Standards

The knowledge representation scheme utilized with the software architecture to represent the functionality and capabilities of modular sensing and actuation components utilizes aspects of the NIST *IEEE 1451* [6, 7] family of standards for smart transducers. These standards describe a set of network-independent communication interfaces that simplify the connection of sensors or actuators to microprocessors, instrumentation systems, and networks, enabling them to be utilized in a "plug-and-play" manner. The core feature of these standards is the *Transducer Electronic Data Sheets* (TEDS) defined for each transducer type, which is a region of memory that stores information about the functionality and capabilities of the transducers, such as calibration information and measurement range, in an easily accessible and network-independent form. Where embedded mem-

ory is not available to facilitate local storage of the TEDS, a remote *virtual TEDS* may instead be used. A standard TEDS layout is shown in Figure 1.3.

The first member of the IEEE 1451 family of standards is *IEEE 1451.0*. This standard defines a common set of commands and TEDS on which the other members of the IEEE 1451 family are built. Through the use of these commands, sensors and actuators may be accessed in a standard fashion independent of the communications medium to which they are connected. This also simplifies the addition of further IEEE 1451 standards to the family as the need arises. The second standard, *IEEE 1451.1*, defines a standard object-oriented, extensible class hierarchy that describes the behaviour of, and provides a software interface to, networked smart transducers. This network-based software interface to the transducers is facilitated by a processing node present within each smart transducer known as the *Network Capable Application Processor* (NCAP). The NCAP contains the logic and hardware necessary to enable the smart transducer module to be interfaced with the communications medium. A customized software framework may be easily implemented through the utilization and composition of the classes present within the IEEE 1451.1 hierarchy.

Each of the remaining members of the IEEE 1451 family define a unique TEDS specification and an interface that provides a link between the NCAP and a particular class of transducers, enabling those transducers to be accessed through any communications medium to which the NCAP may be interfaced. These standards are depicted in Figure 1.4 and are outlined below:

- **IEEE 1451.2** — The *IEEE 1451.2* standard defines a point-to-point digital interface, termed the *Transducer-Independent Interface* (TII), between the NCAP and a *Smart Transducer Interface Module* (STIM). A STIM contains and provides a standard digital interface to the various analog and digital transducers present within a particular networked smart transducer module.

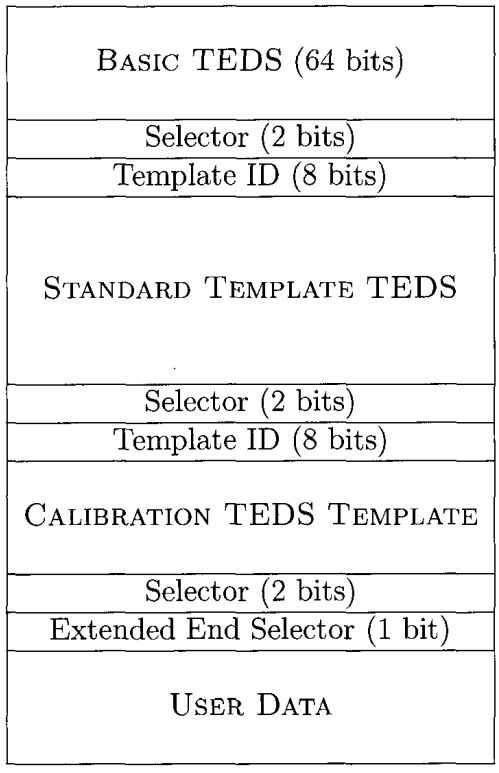


Figure 1.3: Sample TEDS layout [8].

- **IEEE 1451.3** — The *IEEE 1451.3* standard defines a distributed multi-drop digital interface, facilitated through a *Transducer Bus Controller* (TBC), between the NCAP and any number of *Transducer Bus Interface Modules* (TBIMs). Each TBIM provides a standard digital interface to one or more of the transducers present within the multi-drop network.

- **IEEE 1451.4** — The *IEEE 1451.4* standard defines a digital *mixed-mode interface* (MMI) between the NCAP and any number of *mixed-mode transducers* (MMX). Mixed-mode transducers are able to transfer data in both analog and digital forms.

- **IEEE 1451.5** — The *IEEE 1451.5* standard defines a digital wireless interface between the NCAP and any number of transducer modules utilizing various standard wireless transmission protocols. Supported protocols include *WiFi* (IEEE 802.11),

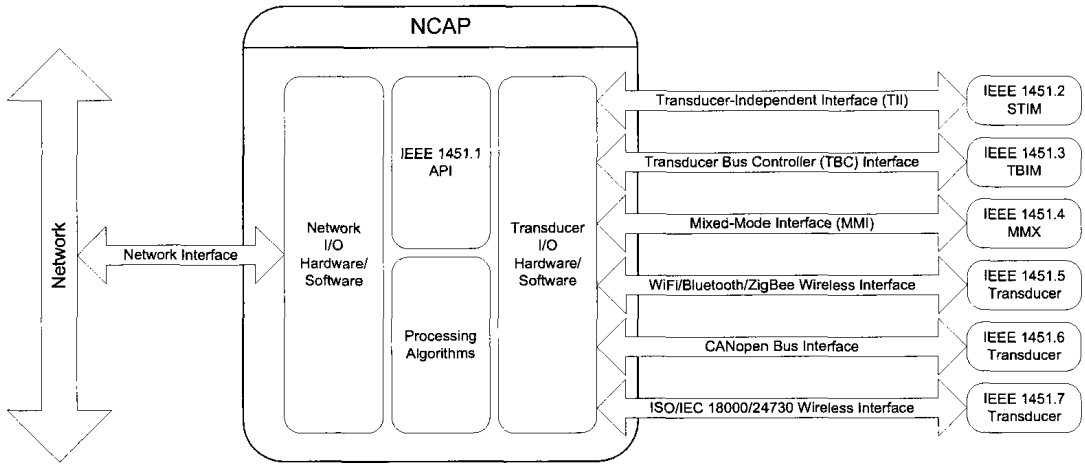


Figure 1.4: Application of the IEEE 1451 standards.

Bluetooth (IEEE 802.15.1), and *ZigBee* (IEEE 802.15.4).

- **IEEE 1451.6** — The *IEEE 1451.6* standard defines a digital interface between the NCAP and any number of transducer modules utilizing the *CANopen* controller area network bus protocol. Both intrinsically safe and non-intrinsically safe operating modes are supported.

- **IEEE 1451.7** — The *IEEE 1451.7* [9] standard defines a digital interface between the NCAP and any number of transducer modules that support communication with ISO/IEC 24753-compliant *radio-frequency identification* (RFID) tags. RFID tags are normally applied to objects of interest for tracking and identification purposes.

1.3.3 Existing Modular Sensing Systems

A number of implementations of reconfigurable modular sensing systems exist in which smart sensor and actuator components may be combined or otherwise collaborate, and are described in the following subsections. Further relevant literature pertinent to the individual components of the software architecture described in this thesis may be found

in subsequent chapters.

Mica

A popular implementation of a reconfigurable modular sensing system is the UC Berkeley *Mica* platform [10]. Each Mica node, known as a *mote*, measures 1.25 × 2.25 inches and runs the TinyOS real-time operating system [11] using a 4 MHz ATmega103L or ATmega128 microcontroller. Wireless communication capability of up to 115 kbps (kilobits per second) is facilitated through the use of an RF Monolithics TR1000 transceiver. Although the motes are capable of collaboration through the use of a peer-to-peer multi-hop wireless networking protocol, no actuation capabilities are supported, and therefore the motes are limited to operating in non-active sensing applications.

Smart-Its

A similar system to the Mica project is the *Smart-Its* [12] project. Smart-Its are self-contained nodes, as small as 17 × 25 × 15 mm, designed to be stuck onto everyday objects. The objects are thus enhanced with sensing and computational capabilities. Each Smart-Its node is aware of its attached sensors and is capable of relaying this information to other nodes in its environment on demand through the use of a query-based *Perception Application Programming Interface* (PAPI). Using the PAPI, each node can gather readings from other nodes in its environment in addition to its own sensor reading through a wireless interface. These values may then be processed locally and transmitted to higher-end devices such as personal computers and personal digital assistants (PDAs). However, like the Mica motes, the active operation and automatic reprogramming capability of Smart-Its nodes is limited.

eBlocks

More closely related to the system described in this thesis are the *eBlocks* [13, 14] embedded system building blocks. An eBlock is an electronic module, incorporating a Microchip PIC microcontroller to provide local intelligence, that allows a small-scale sensor-based system to be created by connecting various eBlocks together. Unlike general purpose sensor-network nodes such as the Mica motes, and like the system described in this thesis, each eBlock performs a specific, well-defined function. Simple sensor networks may be constructed even by users who are not technically adept.

There are four classes of eBlock: *sensor blocks*, which include sensors such as light and motion detectors and output either a digital "yes" or "no"; *logic/state blocks*, which combine the yes and no outputs from the sensor blocks and generate further outputs using combinational or sequential logic; *communicate blocks*, which transform a wired interface into a wireless link; and *output blocks*, which include an actuator such as an LED, buzzer, or relay, and also possess a general-purpose interface that may be used to control other electronic devices or communicate with a more powerful processing device such as a personal computer. Although reconfigurable, connected blocks are unable to determine their overall geometry or quickly and automatically assume a collective identity to suit new configuration requirements. The possible applications of the system are also limited due to the usage of simple combinational and sequential logic functions to produce composite readings and actions.

I-BLOCKS

Another relevant project focusing on the development of modular sensing systems is the *I-BLOCKS* project [15], in which LEGO DUPLO bricks are populated with a PIC16F876 microcontroller as well as select sensors and actuators. These building blocks, like the eBlocks, allow the creation of a modular sensing system without the need to learn and

use a traditional programming language. When physically connected, the blocks are able to communicate with each other through a half-duplex connection, and may also employ wireless communication if desired. The blocks have also been demonstrated to be capable of achieving a degree of positional awareness through the use of infrared positioning techniques based on sensor fusion of the readings produced by multiple infrared sensors. However, like the eBlocks, connected blocks are unable to determine their overall geometry or automatically assume a collective identity based on their orientations. The blocks are also not designed to be easily reprogrammed to suit changing application requirements.

MASS

MASS (Modular Architecture for Sensing Systems) [16] is a modular sensing system architecture that is optimized for low power consumption and is based on modular intelligent nodes. Each node itself consists of physically separable and hot-pluggable modules, each containing a processing controller that facilitates access to resources specific to the module. Node modules communicate through a shared 80-pin bus that also provides structural integrity.

There are four types of MASS modules that may be combined as necessary to produce a node to suit a specific application: *General Purpose Processor* modules (GPPs), which contain a powerful microprocessor or digital signal processor (DSP) used for heavy local data processing or sensor fusion; *sensor* modules, which contain a specific type of sensor and a low power controller that performs rudimentary local sensor data analysis; *Wireless Network Connector* (WNC) modules, which provide wireless connectivity that facilitates inter-node communication; and *power* modules, which provide power to an entire node. Upon connection, modules within a particular node detect each other's resources and the node assumes an appropriate behaviour profile based on the resources discovered.

Similar to the software architecture described in this thesis, the MASS software ar-

chitecture is a layered architecture based on the *Open Systems Interconnection* (OSI) reference model [17], and contains a message-based API (Application Programming Interface) for inter-node communication. Exchange of IEEE 1451-compliant datasheets as described in Section 1.3.2 is also supported. However, MASS provides no capability for active nodes nor the assumption of behaviour profiles based on node positions and orientations.

BUG

BUG [18] is a powerful modular sensing system platform consisting of a collection of electronic modules that are designed to be snapped together to produce a variety of composite components. Available BUG modules include: *BUGbase*, which is a fully programmable embedded computer based on the ARM1136JF-S microprocessor possessing 128 MB (megabytes) of on-board memory, several high-speed communication interfaces including Ethernet, and four slots to which other BUG modules are attached; *BUGview*, which contains a 2.46 inch touch-screen LCD with a resolution of 320 × 240 pixels; *BUGmotion*, which consists of an infrared motion detector with a range of 2 meters and an accelerometer with a software-selectable 2.5 g to 10 g sensitivity; *BUGlocate*, which contains a (GPS) receiver based on a SiRF chipset; and *BUGcam2MP*, which contains a 2-megapixel camera capable of capturing video. Although extremely flexible, the BUG platform does not currently provide functionality to facilitate active sensing. In addition, the orientations in which BUG modules may be attached to the BUGbase are still somewhat limited, and cannot be determined, thus restricting the ability of a composite BUG system to form a new collective identity based on the orientation of its constituent components.

Posey

Posey [19] is a hub-and-strut construction kit enhanced with computational ability. Within a Posey assembly, hubs and struts are optocoupled into flexible ball-and-socket joints with three degrees of freedom, where each ball possesses an array of 11 infrared LEDs and each socket possesses an array of four phototransistors. Each hub and strut contains an embedded ATmega168 microcontroller that captures data from the optocoupled connections, which is used to determine the geometric configuration of any particular joint without requiring explicit alignment of the joint. The microcontroller then relays this information through an XBee ZigBee (IEEE 802.15.4) wireless transceiver to a remote personal computer for further processing. Although Posey supports the acquisition of position and orientation information from the local processing unit located within each ball-and-socket joint, the units themselves do not locally collaborate to form a composite entity. Rather, the system depends upon a more powerful mainstream computer system to provide the necessary intelligence to compose the data provided from the joints.

1.4 Research Objective

The aim of this work is to develop a software architecture and knowledge representation scheme that facilitates the flexible, scalable, and reliable combination of modular sensing and actuation components for the purpose of forming composite sensing devices with motion capability. Each modular component provides a core sensing or actuation functionality (such as temperature or pressure measurement) and contains embedded knowledge of its capabilities (such as its operating range and response time), which is communicated to other modules within its environment. The design of the architectural framework should fulfil the following criteria:

- **Heterogeneity** — Support the connection of sensor and actuator modules possess-

ing diverse functionality and capabilities.

- **Autonomy** — Support the autonomous discovery of the capabilities of networked modules, and the autonomous configuration of these modules based on their discovered capabilities.
- **Pose/Geometry Determination** — Support the determination of the absolute or relative *pose* (position and orientation) of individual modules, and by extension the overall geometry of a set of connected modules.
- **Assumption of a Collective Identity** — Facilitate the assumption of a collective identity by successfully connected modules, based on their capabilities and relative positions and orientations.
- **Process Distribution** — Support the splitting and distribution of a complex task among a group of networked modules.
- **Resource Management** — Manage the hardware resources on each module in an efficient, intuitive, and simple manner.
- **Scalability** — Maintain reliable operation with an increasing number of connected sensor and actuator modules.
- **Robustness** — Adapt automatically to the addition, removal, or failure of modules in real-time.

1.5 Thesis Outline

This thesis is divided into eight chapters that progressively describe the design and operation of the software architecture from its lowest level interactions with the module

hardware to its highest level software components. A brief synopsis of the contents of each chapter is provided as follows:

- **Chapter 1 — Introduction**: This introductory chapter. Outlines the motivation behind the development of the software architecture as well as its design criteria, and provides a survey of existing modular sensing system architectures as well as standards facilitating sensor interoperability.

- **Chapter 2 — Architecture Description**: Outlines the software architecture itself and the hardware on which it executes. Topics covered include the utilized real-time operating system and file system, as well as an overview of the various layers present within the architecture.

- **Chapter 3 — Communication Layer**: Describes the communication layer of the software architecture, which provides a reliable, connection-oriented service for wired and wireless communication to the layers above it, and also facilitates time synchronization between modules.

- **Chapter 4 — Middleware Layer**: Outlines the middleware layer of the software architecture, which defines a standard application programming interface (API) that facilitates interoperability between the modules on which the architecture executes and enables them to request services from each other.

- **Chapter 5 — Virtual Machine**: Describes the virtual machine utilized within the software architecture, based on Sun Microsystems' Java Virtual Machine, that promotes the straightforward portability of collaborative intelligence algorithms between diverse module hardware platforms.

- **Chapter 6 — Composition Layer**: Outlines the composition layer of the software architecture, where intelligence is implemented and utilized in the form of platform-

independent algorithms that enable a group of modules to collaborate and form composite entities.

- **Chapter 7 — Architecture Evaluation**: Provides results and analysis of the behaviour and performance of the software architecture once deployed on actual module hardware.

- **Chapter 8 — Conclusions**: Provides a final overview of the software architecture and also provides recommendations for further improvement.

Chapter 2

Architecture Description

2.1 Introduction

This chapter provides a description of the module hardware on which the software architecture executes, as well as the layered model on which the architecture is based. The real-time operating system utilized for the purpose of concurrent task management is then described, followed by information on the file system driver and the organization of the standard file structure. A description of the core data types utilized throughout the software architecture is then provided.

2.2 Module Hardware Overview

2.2.1 Transducer Interface Modules

The basic module used to construct modular sensing systems is the *transducer interface module* (TIM). Each is capable of a single sensing or actuation function, and is uniquely identified by a 64-bit address. This address possesses a most significant bit of zero, since addresses with a most significant bit of one are reserved for assignment to composite

module entities termed *logical modules* that are comprised of a set collaborating TIMs. The zero address and the address of all ones are also reserved, and may not be assigned to single nor composite modules. The address of all ones is utilized for broadcasts to all the modules comprising a modular sensing system. With this addressing scheme, up to approximately 9.2×10^{18} physical transducer modules may be uniquely addressed, a value which is likely to exceed the number of modules created throughout the lifetime of the technology. As specified in the IEEE 1451 standard for smart transducers [6], each module possesses one or more *Transducer Electronic Data Sheet* (TEDS) specifications in non-volatile memory, from which a description of the characteristics of its associated sensors or actuators may be obtained.

TIMs are cubical in shape, and thus each possesses six faces to which up to five other modules may be connected, as shown in Figure 2.1. One face is reserved for use by the transducer associated with the module. The hardware which comprises a TIM, shown in Figure 2.2, includes the associated transducer; a high-speed NXP Semiconductors LPC2148 ARM-based microcontroller [20]; a Nordic Semiconductor nRF24L01 [21] wireless transceiver supporting high-speed data transmission, multi-channel operation and carrier detection; a Secure Digital™ (SD) flash memory card providing high-capacity, non-volatile storage for data and algorithms; a power supply capable of providing a voltage of 3.3 volts to 9 volts; and five *module connectors*, which are proprietary interfaces used to physically connect additional modules. The interfaces are designed such that the relative orientation between any two connected modules may be determined. Further details on the electrical and mechanical design aspects of the TIMs may be found in [22].

2.2.2 Other Module Types

A modular sensing system may consist of two other types of modules significant to the software architecture. These modules perform tasks unrelated to sensing and actuation;

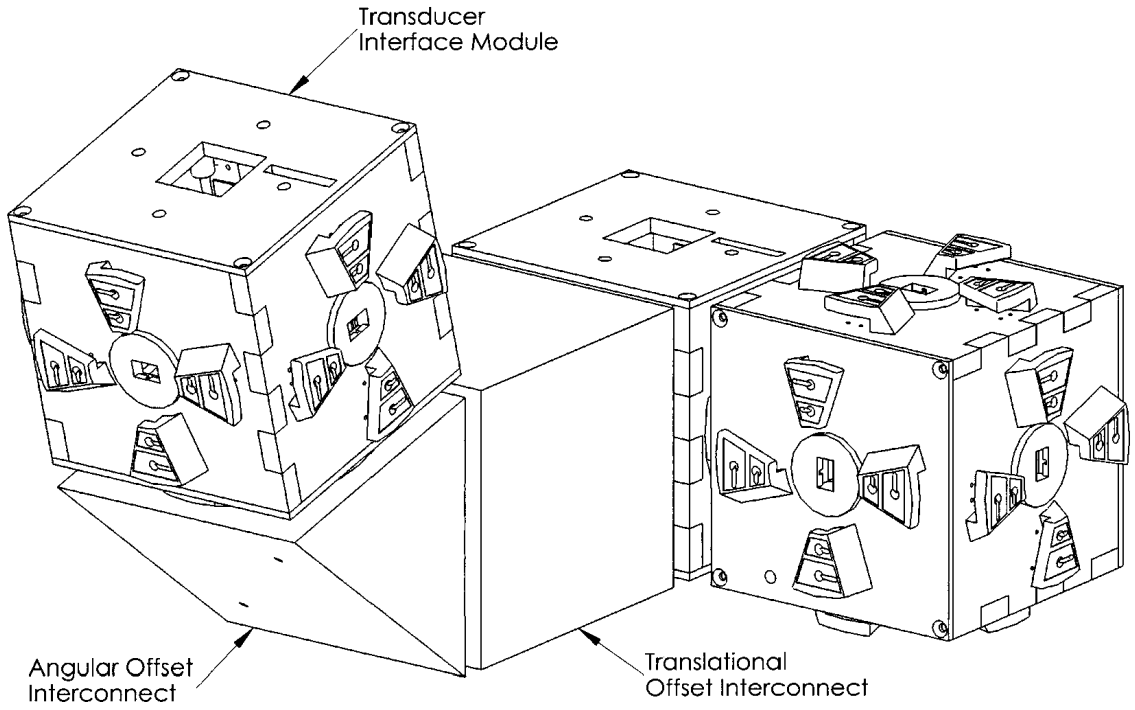


Figure 2.1: Transducer Interface Modules and interconnects.

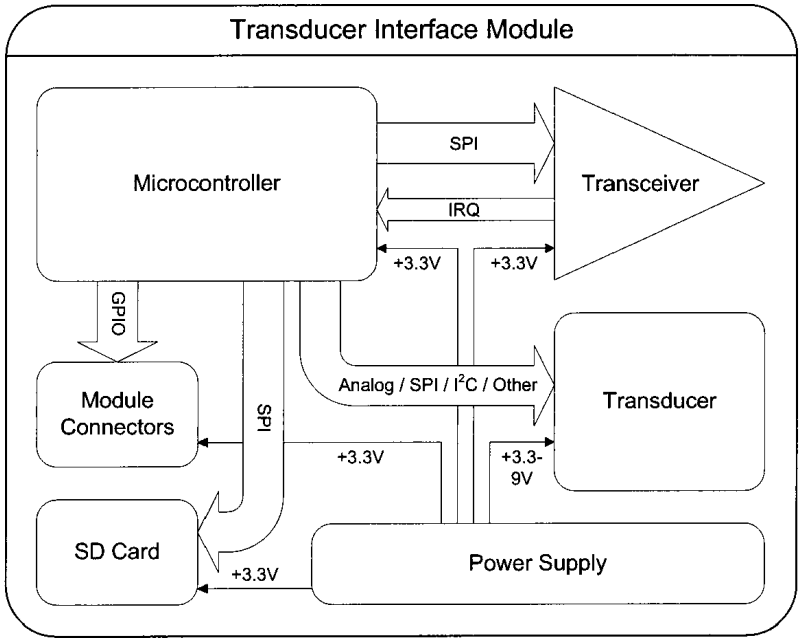


Figure 2.2: Transducer Interface Module block diagram.

instead, they support the inter-operation of a group of TIMs.

Administration Module

An *administration module* is used by the system user to detect and manage TIMs within its vicinity. It possesses only a power supply, a microcontroller, and a transceiver. It may be integrated into a complete computer system, or be a small, self-contained console with a user interface. Administration modules may also act as a sink for transducer readings and as a gateway for communication with a larger network, such as the Internet. All TIMs, however, may run an optional shell task which provides the system user with administrative functionality.

Interconnect Module

Interconnect modules are each built to assume one of a variety of non-standard shapes, and are used to provide angular and translational offsets between connected TIMs which would otherwise not be possible due to the cubical shape of the TIMs. An example of an interconnect module which provides an angular offset is shown in Figure 2.1. They possess only a microcontroller and module connectors, and draw power from the TIMs to which they are connected. The nature of the offset provided by a particular interconnect module is stored in its TEDS, and may be accessed through its module connectors.

2.3 Software Architecture Stack

The software architecture described in this thesis is a distributed architecture based on the *Open Systems Interconnection* (OSI) reference model [17], and consists of six layers (one of which is divided into two sub-layers) as shown in Figure 2.3. The use of a *distributed* architecture ensures that no single point of failure exists within a modular

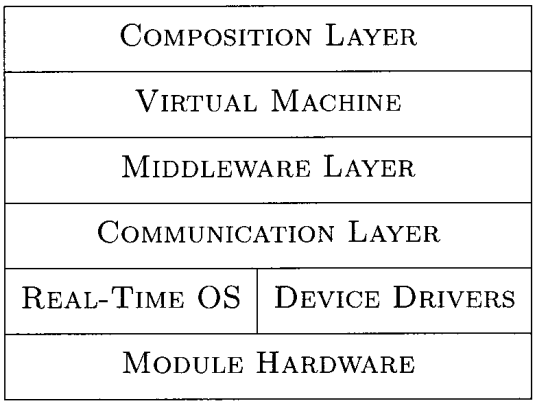


Figure 2.3: Software architecture stack.

sensing system and also facilitates architecture scalability, unlike *centralized* architectures, in which a single point of failure is often introduced that can also limit scalability in large systems where communication between nodes mostly occurs through this point. The use of a layered architecture model allows the implementation of any layer to change independently of the others, since the implementation of each layer is encapsulated from the layer above, to which it provides service. This information-hiding technique also facilitates a more robust software architecture, and makes each of the architecture layers easier to implement, modify, and debug. The function of each layer is defined as follows:

- **Module Hardware** — Contains the physical components of a module needed for execution of the operating system, sensing and actuation functionality, as well as wired and wireless communication.

- **Real-Time Operating System/Device Drivers** — Provides resource management functionality and an environment for concurrent task execution, as well as the low-level software routines needed to manipulate and manage the hardware resources present in the module.

- **Communication Layer** — Provides an interface to the wireless transceiver driver

that automatically accounts for transmission problems such as packet loss and synchronization. This layer also provides an interface through which modules may communicate using their face connectors.

- **Middleware Layer** — Provides the commands and services through which the member TIMs comprising a *logical module* may interact and communicate with each other in order to achieve a specific goal. A logical module is an abstraction of one or more collaborating TIMs, a representation of which is present locally on each of the TIMs comprising the logical entity. More than one logical module representation may be present on a single TIM.

- **Virtual Machine** — Provides a platform-independent execution environment for the algorithms utilized in the composition layer. This enables the behaviour of a group of collaborating TIMs to be specified in a manner that is completely decoupled from their underlying hardware architecture. Platform independence is facilitated through the use of a compact implementation of Sun Microsystems' *Java Virtual Machine* [23].

- **Composition Layer** — Encompasses one or more *logical module template algorithms* that process the transducer state and module pose messages transferred among a group of collaborating TIMs and enables them behave as a logical entity. Each template algorithm, in the form of a Java class, is accompanied by a *logical module template TEDS* that describes the basic characteristics of a logical module entity derived from on it.

2.4 Real-Time Operating System

The software architecture utilizes a *real-time operating system* (RTOS), which enables it to be implemented in a modular fashion through the concurrent execution of various *tasks*. As a result, the management of the hardware resources of a module, as well as the development and debugging of the software architecture, is vastly simplified. Tasks are implemented as independent functions, each with its own stack and register set, that appear to be running simultaneously, but are actually sharing the execution time of the microcontroller through the use of *scheduling* mechanisms implemented within the operating system.

2.4.1 Scheduling Policies

In an RTOS, concurrently executing tasks may be scheduled using either a *pre-emptive* scheduling policy or a *cooperative* scheduling policy. In pre-emptive scheduling, CPU time is automatically shared between tasks based on their assigned priority, while in cooperative scheduling each task maintains control of the CPU until it explicitly yields control. Pre-emptive scheduling is advantageous since it prevents long-running, low-priority *background* tasks from blocking shorter, higher-priority *foreground* tasks from executing, thus improving system response speed to external events. In the popular *TinyOS* [11] RTOS, which utilizes a cooperative scheduler, all tasks must run to completion. Long-running background tasks are therefore prohibited, and care must be taken to ensure that each task completes in a reasonable amount of time.

2.4.2 Choice of RTOS

Two real-time operating systems, *FreeRTOS* [24] and *TNKernel* [25], were considered for use in the software architecture. Due to the limited 40 kilobytes of internal RAM (random-

access memory) present on the LPC2148 microcontroller, more complex operating systems such as eCos and RTLinux could not be assessed. Both FreeRTOS and TNKernel are free, open source, compact, and well documented; they also possess a large user base and code that has been heavily tested on a variety of embedded architectures, including the ARM architecture. In addition, FreeRTOS and TNKernel both contain a priority-based pre-emptive task scheduler, and make provisions for message passing and synchronization between concurrently executing tasks.

The real-time operating system chosen for use in the software architecture presented herein is TNKernel. This RTOS was chosen over FreeRTOS due to its more compact and easily modifiable code base. In addition to not relying on the standard C library, TNKernel does not utilize any form of dynamic memory allocation internally, thereby allowing the implementation of a simplified dynamic memory allocator to be used exclusively by the upper layers of the software architecture where necessary.

2.4.3 Task Types

Three standard background tasks, and at least one message handling task, are created and executed upon startup and initialization of a module. Multiple message handling tasks may also be created and executed by the software architecture at any point during its execution, depending on the type of modules that are within close proximity. These various task types are briefly described below.

- **Network Communication Task** — Performs various duties related to communication on the various wireless data channels. These duties include broadcasting control packets which indicate the presence of the module in the network; transmitting, receiving, and processing data packets; and maintaining synchronization with other modules in the environment. This task is described in further detail in Section 3.6.

- **Face Communication Task** — Manages the communication of the module with others physically connected to its faces. This task detects the physical connection and disconnection of other modules to any of the five connectible module faces, handles the transfer of data through the connectors on the faces, and calculates the relative *pose* (position and orientation) between the module and those to which it is physically connected. This task is described in further detail in Section 3.7.3.

- **Administrative Interface Task** — Allows the system user to monitor and administrate any module, or logical group of modules, within the modular sensing system. Some of the functionality available to the system user includes listing the modules in the environment, manually forming a logical module entity, reading and writing to module sensors and actuators, modifying the local clock of the module, and various debugging features such as suspending and resuming tasks and monitoring wireless channel activity.

- **Module Message Handler Task** — Handles middleware layer messages placed within the incoming message queues of a TIM or a logical module entity of which it is a member. This task is also responsible for performing various status checks on its associated module before each received message is processed, such as determining if a queued write message was received from a source with the appropriate write permissions. At least one message handler task is created by default for a TIM upon initialization, to process and transmit messages related to its local hardware. Other instances may be created as the TIM becomes a member of one or more logical module entities. For physical TIMs, the message handler task is fully implemented natively, while for logical modules it is mostly implemented using platform-independent Java classes.

2.5 File System

A *file system* is a set of data structures that facilitates the storage, organization, and retrieval of files from a data storage device. A file system is employed within the software architecture to provide an efficient, high-level interface to information and algorithms stored on SD flash cards that determine the identity and behaviour of a particular module in a network. Utilizing a lightweight and widely adopted file system enables the system user to easily access and modify these files using standard computers and operating systems.

SD flash cards to be utilized by the software architecture are formatted with the *FAT32* (32-bit File Allocation Table) [26] file system and initialized with a standard file structure that is described in Section 2.5.2. The FAT32 file system was chosen due to its wide support on numerous mainstream operating systems for personal computers, particularly Microsoft Windows, GNU/Linux, and Mac OS X. FAT32 is also lightweight as compared to other popular file systems such as *ext3* (Third Extended Filesystem) [27] and *NTFS* (New Technology File System) [28] that utilize additional data structures and memory in order to reduce file fragmentation and to provide features such as *journaling* that aid recovery from file corruption, neither of which is crucial to the operation of TIMs. Finally, the FAT32 file system is also well understood, resulting in the availability of a number of stable and mature FAT32 file system drivers for embedded devices.

2.5.1 Choice of FAT32 File System Driver

Two popular FAT32 file system drivers, the *FAT File System Module* [29] and the *Embedded Filesystems Library* (EFSL) [30], were considered for use in the software architecture. Both are open source, compact, and easy to use and modify. The FAT File System Module was chosen for use in the software architecture due to the availability of the extremely

compact *Tiny-FatFs* version of its standard *FatFs* driver. It consumes much less flash memory space and utilizes much less RAM as compared to FatFs and EFSL, and is also easily adaptable to any type of input/output (I/O) device assuming a block I/O driver is provided. Unlike FatFs and EFSL, Tiny-FatFs does not support I/O transfers to more than one storage device at a time; however, the SD card is the only storage device present in a TIM.

2.5.2 Standard File Structure

As previously mentioned, an SD card formatted for use with a TIM is initialized with a standard file structure for organizational purposes. A standard file structure is utilized to ensure that the software architecture is consistently able to locate and access the files necessary for its operation from predictable locations irrespective of the underlying hardware on which it executes or the storage medium on which these files are located. Access to these files by the users of the system is also made more convenient. The file structure designed for the purposes of the software architecture is depicted in Figure 2.4, and consists of four directories as well as up to four different types of files. These directories and files are described below.

- **Template Class Directory** — The *template class directory* `amss/algo` is the directory in which the Java classes, termed the *logical module template classes*, are placed. These classes provide the platform-independent intelligence that enables connected TIMs to collaborate with each other. The template class directory path `amss/algo` adheres to the Java naming convention for package paths, and corresponds to the package `amss.algo` that all template classes are declared a member of. Unlike standard Java classes, template classes possess no `.class` extension since the Tiny-FatFs driver requires strict adherence to the *8.3 file naming convention* [31]. In this convention files may only possess a name consisting of up to eight characters

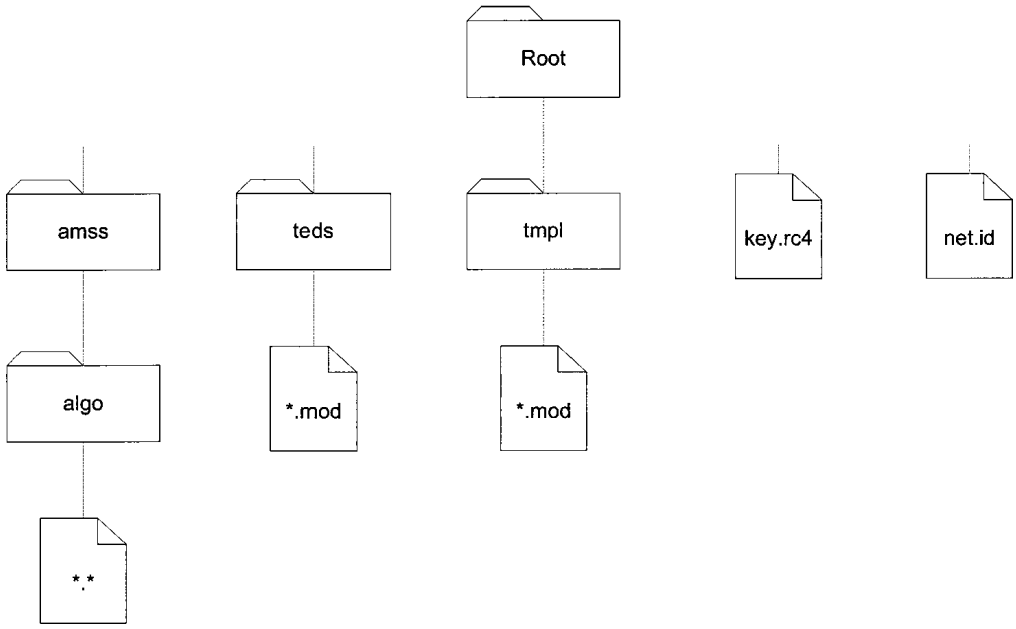


Figure 2.4: File system block diagram.

optionally followed by a period and an extension of three characters. This limitation was imposed by the original FAT file system used in older Microsoft operating systems, and is only non-obligatory in the more complex implementations of FAT32 utilized in mainstream operating systems.

- **Module TEDS Directory** — The *module TEDS (Transducer Electronic Data Sheet) directory* `teds` consists of one or more text files termed *module TEDS*, each possessing the extension `.mod`, that identify and describe the characteristics of the transducers associated with a particular physical TIM in the form of a list of *property-value pairs*. The format of these text-based specifications is further described in Section 2.6.

- **Template TEDS Directory** — The *template TEDS directory* `tmpl` consists of zero or more text files termed *template TEDS* that identify and describe the characteristics of a combination of collaborating TIMs known as a *logical module*. Tem-

plate TEDS are specified using the same format as module TEDS and possess the same `.mod` extension. Each template TEDS specification is associated with one template class in the template class directory. In addition to the standard characteristics of the logical module, a template TEDS specification also describes the various *roles* (further outlined in Section 6.2.2), also in the form of property-value pairs, that may be fulfilled by a particular TIM within the logical entity. The usage of template TEDS files within the software architecture is further outlined in Chapter 6.

- **ARC4 Key File** — The *ARC4 key file* `key.rc4` stores the variable-length *key* required by the *Alleged Rivest Cipher 4* (ARC4) cryptographic stream cipher [32] utilized by the software architecture for the secure transmission of packets. If this file is not found, a default key is used. All packets transmitted by TIMs on the wireless communication medium are encrypted and decrypted in software using the ARC4 algorithm, and modules are only able to communicate with others that are utilizing the same key. Packets received from modules utilizing different keys will be indecipherable upon reception and are dropped. More details on the ARC4 algorithm and key file may be found in Sections 3.2.4 and 3.5 respectively.

- **Network Identifier File** — The *network identifier file* `net.id` stores the 5-byte *network identifier* used to indicate that a particular TIM is a member of the network of TIMs possessing the network identifier specified. If this file is not found, a default identifier is used. Packet transmissions from modules with different network identifiers are completely ignored, thus reducing packet processing overhead within the software architecture. More details on the network identifier may be found in Sections 3.3 and 3.5.

2.6 TEDS Specification Format

As described in Section 2.5.2, *TEDS (Transducer Electronic Data Sheet) specifications* are text files that are defined to identify and describe the characteristics of the transducers associated with a particular physical TIM, or to identify and describe the general characteristics of a combination of collaborating TIMs known as a *logical module*. A TEDS specification always possesses the extension `.mod`, and consists of a list of *property-value pairs*, each on a separate line, with whitespace used to separate each property name from its value. Comments are supported within the file format; all characters on a line after and including the # character are ignored. The usage of a text format enables the TEDS to be specified in an easily human-understandable form, unlike the binary TEDS format normally utilized within the IEEE 1451 standards for smart transducers [8], which is typically incomprehensible to human readers. The *Extensible Markup Language* (XML) [33], a popular format for representing data through the use of text, is not utilized for the purpose of module TEDS specification since it introduces redundant, verbose syntax that often makes information of interest difficult to locate by human readers. XML is also non-trivial to parse, especially on resource-constrained embedded devices.

Sample module TEDS specifications may be seen in Section B.2, and sample template TEDS specifications may be seen in Section B.3. A standard TEDS specification consists of three sections. The first section is the *AMSS (Adaptive Modular Sensing System) TEDS*, which is specific to the software architecture and describes the essential attributes of a module necessary for the software architecture to utilize it. The other two sections, the *Basic TEDS* and the *Standard Template TEDS*, facilitate the definition of manufacturer-specific information and transducer properties as defined within the IEEE 1451 standards, and are outlined in [8]. An additional *Roles* section, also specific to the software architecture, is present only within template TEDS specifications. As described in Section 2.5.2, the Roles section defines the *roles* within a logical entity derived from

the template TEDS that may be fulfilled by its member modules, and is further outlined in Section 6.2.2. The fields contained within an AMSS TEDS specification are defined as follows:

- **Module Address** — The 64-bit address assigned to a module to uniquely identify it. This field is not present within template TEDS specifications, since the addresses of the logical modules derived from these templates are automatically assigned by the software architecture at runtime.

- **Module Type** — A constant indicating the type of the module, which may be a *sensor* module, an *actuator* module, an *interconnect* module, or an *administrator* module.

- **Module Class** — A constant indicating the *class* of the module. A *class* refers to a family of sensors or actuators that may be used to sense a particular physical quantity or facilitate a specific type of motion respectively. Currently, the supported module classes are *acceleration* modules, *positional* modules, *rotational* modules, *status* modules, *text display* modules, and *voltage* modules.

- **Module Data Type** — A constant indicating the data type of the array of values returned by the module. The return type may be an 8-bit, 16-bit, 32-bit, or 64-bit signed or unsigned *integer*, a 32-bit or 64-bit *floating point* value, a *status string*, an encompassed middleware layer *message* (see Section 4.3), or a generic *object* consisting of raw bytes.

- **Module Data Type Width** — Specifies the number of columns in the array of values returned by the module.

- **Module Data Type Height** — Specifies the number of rows in the array of values returned by the modules.

- **Primary Handler Name** — A string storing the name of the appropriate driver function that handles accesses to the sensing or actuation devices associated with the module. This field is not present within template TEDS specifications, since logical modules derived from these templates indirectly utilize the sensing or actuation resources provided by its member modules.

2.7 Core Data Types

A variety of data structures are used to represent and store information at each layer of the software architecture, and many are utilized throughout multiple layers. Most of these data structures are defined directly using the standard *array*, *structure*, and *union* data types provided within the C programming language used for the development of the software architecture. However, *lists* and *queues*, which are utilized heavily, are not provided as standard constructs within C. Therefore, a custom *vector* data type was implemented to provide the functionality of both a list and a queue. The core data structures utilized throughout the software architecture are described in the following subsections.

2.7.1 Vector Implementation

A vector may be implemented either as a *dynamic array*, in which a single block of memory is allocated for all constituent elements and resized as necessary; or as a *linked list*, in which memory is dynamically allocated for each constituent element as needed. The vectors utilized within the software architecture are based on dynamic arrays. Dynamic arrays were chosen because they do not require each element to be associated with pointers that maintain the links between constituent element nodes. As a result, memory utilization is minimized. In addition, the most common operations performed on vectors

in the software architecture are random access and insertion or deletion at the end of the vector, of which random access is generally much faster when utilizing dynamic arrays. This is seen upon comparison of the *computational complexities* of these operations for linked lists and dynamic arrays.

The computational complexity of an algorithm, typically specified using *big-O notation*, is a theoretical measure of how costly (in terms of execution time or memory requirements) the algorithm is relative to an input of size n. A function $f(n)$ is $O(g(n))$ if $f(n)$ is less than a constant multiple of $g(n)$. Linked-lists have $O(n)$ complexity for random access operations and $O(1)$ complexity for insertion or deletion operations at the end of the list. This indicates that the typical time required to perform a random access operation on a linked list is proportional to the number of elements in the list, while insertion and deletion operations at the end of the list take constant time irrespective of the number of elements. Dynamic arrays have $O(1)$ complexity for both random access operations and insertion or deletion operations at the end of the array, and therefore both operations take constant time irrespective of the number of elements in the array.

2.7.2 Task Data Types

Task Structure

A *task structure* is used to store the properties and data needed to describe and maintain an executing task. Additional data is stored within a task structure beyond that utilized internally by the RTOS, and facilitates the provision of identification and statistical information. This information is utilized by the administrative interface task to report statistics related to the currently running tasks, including itself, on a module. The values contained within a task structure are as follows:

- **Task Control Block** — A structure used internally by the TNKernel RTOS to

manage the execution of the task. The task control block contains data such as task priority and time slice counters.

- **Code** — The function that implements the task algorithm and is provided with its own context in which to concurrently execute.
- **Stack** — A block of memory exclusive to the task that facilitates function calls, recursion, local variables, and servicing of interrupts.
- **Name** — A variable-length string indicating the name of the task, for easy identification by the system user.
- **Identifier** — A 32-bit unsigned integer unique to each task running on the module hardware, used for identification purposes within the software architecture.
- **Parameter** — A pointer used to pass a single parameter or multiple parameters to the task function.
- **Processor Usage** — Two variables used in calculating the percentage of time the task spends executing relative to other concurrently executing tasks, and reporting it to the system user.

Task Structure List

The *task structure list* is a global vector used to store the task structures that represent each concurrently executing task on a module. The provision of a global task structure list makes available to the software architecture a single, standard location from which all the task structures representing the tasks running on a module may be accessed. Newly created and initialized tasks are automatically appended to the task structure list before their execution begins, and are automatically removed from the task structure list when their execution ends.

2.7.3 Module Data Types

Module Structure

A *module structure* is used to store data that represents the state of a single transducer on a physical TIM or the state of a logical entity formed by a group of collaborating TIMs. Upon startup, module structures are created and initialized on a TIM based on each module TEDS specification found in its module TEDS directory. TIMs are required by default to support a single sensing or actuation function, and will normally provide a single module TEDS specification. However, a TIM with additional sensing and actuation capabilities will normally provide additional module TEDS specifications associated with each capability. Each transducer element will appear within the sensing system network as a unique TIM.

As a TIM discovers other modules within its network, it will automatically search for compatible modules with which it can collaborate and form a logical entity. For each logical entity joined or created, a representative module structure is created locally on the TIM. Further details on the usage of module structures associated with logical entities may be found in Section 6.2. The data that comprises a module structure is described below:

- **Message Handler Task Identifier** — Identifies the *message handler task* (further outlined in Section 4.5) associated with the module structure that processes messages received by the represented module.

- **Execution Flag** — Used to track whether the associated logical module task is running or shut down. Module structures associated with shut down logical module tasks are garbage collected.

- **Synchronization Level** — Indicates the degree to which the local clock of the module should be considered a reference for synchronization. Lower values corre-

spond to more accurate references. Interconnect modules are initialized with the highest synchronization level of 255, standard TIMs are initialized with a synchronization level of 254, and administration modules are initialized with the lowest synchronization level of 1. After each synchronization with a reference, the local synchronization level is set to that of the reference plus one.

- **Status Check Timestamps** — A set of timestamps, each of which is associated with one of a number of status checks that maintain the integrity of the module structure. These checks ensure that the module is not indefinitely locked or otherwise becomes incapable of processing incoming messages. After the invocation of its associated status check, each timestamp is updated by the module task to indicate the time of the next invocation.

- **Locking Address** — If non-zero, indicates the address of the remote module that has *locked* the TIM. Locked TIMs can be read by all modules, but can only be written to by the module that issued the lock until the lock is released.

- **Logical Module Template** — A structure that stores the template class and roles that describe the behaviour of a logical module entity. It is used only if the module structure represents a logical module.

- **Membership List** — Stores *membership structures* that indicate the roles that the module fulfils in one or more logical module entities.

- **Incoming Message Queues** — Three queues in which incoming messages to be processed by the task associated with the module structure are placed. The *Call-At* and *Call-By* message queues store messages to be processed at or by a particular deadline respectively, while the *Return* message queue stores messages returned by other modules in response to an issued command.

- **TEDS Properties** — A collection of variables and a vector of *TEDS entry structures* that collectively describe the properties of a TIM and its associated transducer. The values used to initialize the properties are loaded from the respective TEDS file located in the module TEDS or template TEDS directories.

- **Primary Message Handler** — A pointer to the driver function that handles read and write messages to the sensing or actuation device on the module.

Module Structure List

The *module structure list* is a global vector used to store the module structures that locally represent each physical or logical module entity associated with a particular TIM. The provision of a global module structure list makes available to the software architecture a single, standard location from which all the module structures representing the transducer elements or logical entities available on a particular physical TIM may be accessed. Newly created and initialized module structures are automatically appended to the module structure list before they are utilized, and are automatically removed when they are shut down.

Membership Structure

Membership structures are stored within the membership list of a module structure, and each provides essential information about a single *role* that the encompassing module fulfils within a particular logical entity in a modular sensing system. The fields comprising a membership structure are:

- **Logical Module Reference** — A reference to the local module structure associated with the logical module that the membership structure is representing membership of.

- **Role Number** — Indicates the *role* fulfilled within the referenced logical entity by the module possessing the membership structure.
- **Physical Dependency Address** — The address of a remote physical TIM (that is also a member of the logical entity) to which a direct or indirect physical connection must be present for membership to be considered valid.

TEDS Entry Structure

A *TEDS entry structure* represents a single property-value pair that forms part of a complete module TEDS or template TEDS specification. It consists of two 24-byte strings used to store the name of the property and its associated value respectively. The use of 24-byte strings facilitates acceptably small memory consumption by TEDS specifications, while being sufficiently large enough to allow the name and associated value of each TEDS property to be expressed in an easily human-understandable manner.

2.8 Summary

In this chapter, the *transducer interface module* (TIM) hardware on which the software architecture executes was described, as well as the layered model on which the architecture is based. Also outlined was the TNKernel real-time operating system at the heart of the software architecture, the FAT32-based file system used for non-volatile data storage, and the core data structures utilized throughout the various architecture layers. The use of a real-time operating system and a layered model promotes modularity, resulting in the software architecture being easier to implement, modify, and debug. The following chapter will describe the operation of the *communication layer*.

Chapter 3

Communication Layer

3.1 Introduction

The purpose of the *communication layer* is to provide *logical link control* in the form of a secure, reliable, connection-oriented service; *medium access control* to prevent channel access conflicts; a mechanism for *time synchronization* between modules; and *wireless security* through the encryption of transmitted data, facilitated by a cryptographic stream cipher. The communication layer accepts *messages* from the middleware layer and splits them into discrete *packets*, which are then encrypted and transmitted through the wireless transceiver driver. Conversely, the communication layer also accepts and decrypts incoming packets from the wireless transceiver driver, merges them into messages if necessary, and passes them to the middleware layer. The communication layer also implements a wired protocol that facilitates the direct transmission of data through the faces of physically connected TIMs.

3.2 Communication Layer Services

3.2.1 Logical Link Control

Wireless communication tends to be very unreliable in the absence of an error correction mechanism, mainly due to the regular interference encountered by radio waves during their propagation. Therefore, a reliable, connection-oriented service must be provided within the communication layer to ensure that transmitted packets arrive error-free and in the correct order. This service is facilitated in the form of a *Positive Acknowledgement with Retransmission* (PAR) data link protocol, which is partially implemented within the nRF24L01 transceiver hardware under the Enhanced ShockBurst™ feature set [21].

In this protocol, depicted in Figure 3.1, each packet that requires guaranteed transmission is automatically acknowledged by the receiving module through the use of an Enhanced ShockBurst™ *acknowledgement packet* (ACK). Among the data present within each received packet are a *cyclic redundancy check* (CRC) checksum used to detect packet transmission errors and a *packet identification* (PID) number used to differentiate between new and retransmitted packets. The CRC checksum and PID number are both automatically generated by the transceiver of the transmitting module. A received packet is considered valid by the receiving module only if its CRC checksum is valid and its PID number does not match that of the previously received valid packet. The PID number is incremented after the transmission of each packet, and therefore a repeated PID number indicates that the previous packet was assumed lost by the transmitting module and was therefore retransmitted.

After transmitting a packet the transmitting module listens to the wireless channel for an acknowledgement packet, for a period of time known as the *valid acknowledgement time window*, depicted in Figure 3.1 as an `ACK RX` block. This time period is set to the maximum window length of 4 μs as permitted by the nRF24L01 transceiver. If

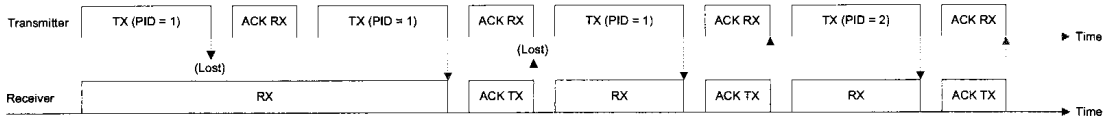


Figure 3.1: Transmission of two packets using an unreliable medium.

the valid acknowledgement time window elapses without the acknowledgement packet being received, the transmitting module assumes that the transmitted packet or its corresponding acknowledgement packet was corrupted and automatically resends the packet. If the maximum 15 retransmissions permitted by the nRF24L01 transceiver occur without success, transmission attempts are halted and the transceiver generates an interrupt indicating the packet transmission failure.

A message which is larger than the 96-bit *maximum transmission unit* (MTU) defined by the communication layer (see Section 3.3) is fragmented into multiple packets before transmission. The message transmission and reception process is further detailed in Sections 3.6.1 and 3.6.2 respectively.

3.2.2 Medium Access Control

Since a modular sensing system will normally be comprised of a number of collaborating modules, a *Medium Access Control* (MAC) protocol needs to be provided within the communication layer to share the single multi-access broadcast channel among the many contenders competing for control of the medium. A MAC protocol may generally be described as being either a *static allocation* protocol or a *dynamic allocation* protocol. In static allocation protocols, channel bandwidth is divided into equally sized portions, with each portion allocated to one transmitting device. In dynamic allocation protocols, channel bandwidth is allocated to each transmitting device on an as-needed basis. The following MAC protocols were considered for use in the software architecture:

- **TDMA** — *TDMA* (Time Division Multiple Access) [17] is a static allocation protocol in which a single communication channel is shared by dividing access to the channel into a number of consecutive *time slots*. Each user is allocated one time slot, and is only permitted to transmit during that time slot. TDMA is efficient at high loads; however, when loads fall, time slots will go unused, wasting bandwidth. Reallocation of time slots also becomes complex when users regularly enter or leave the system. Another disadvantage of TDMA is the need for constant and accurate global synchronization between users to ensure that their time slots do not overlap. Any loss of synchronization results in packet collisions.

- **ALOHA** — *ALOHA* [17] is a dynamic allocation protocol which has two versions: *pure* and *slotted*. In *pure* ALOHA, users are allowed to transmit whenever they have packets to send. If a collision occurs, the users which attempted transmission wait a random amount of time before attempting to transmit again. This results in very high packet collision rates, and thus inefficient usage of bandwidth.

  In *slotted* ALOHA, users are only allowed to transmit at the beginning of discrete time intervals. As in pure ALOHA, if a collision occurs, the users which attempted transmission wait a random amount of time before attempting to transmit again. This results in the bandwidth usage of slotted ALOHA being twice that of pure ALOHA, although the packet collision rate is still high since all users are free to transmit once the next time slot arrives. Like TDMA, slotted ALOHA requires constant and accurate global synchronization between users to prevent overlapping of time slots.

- **CSMA** — *CSMA* (Carrier Sense Multiple Access) [17] is a dynamic allocation protocol in which users are able the sense the communication channel and determine if a transmission is already in progress.

In *1-persistent* CSMA, a user transmits as soon as the channel becomes idle (that is, with probability 1). If a collision occurs, the user waits a random amount of time before attempting to sense the channel and transmit again. In *nonpersistent* CSMA, the user does not immediately transmit once the channel becomes idle; rather, it waits a random period of time before attempting transmission. In *p-persistent* CSMA, the communication channel is slotted. If the user senses the channel is idle, it transmits at the slot interval with probability p, and defers to the next slot with probability $1 - p$. *CSMA/CD* (Carrier Sense Multiple Access with Collision Detection) is an improvement on the standard CSMA protocols in which users cease transmitting as soon as a collision is detected.

CSMA protocols achieve far greater bandwidth utilization than ALOHA protocols. CSMA protocols can fail, however, in wireless systems utilizing short-range radio waves, since in such systems the sender is able to sense all channel activity within its vicinity, but not necessarily all channel activity within the vicinity of the receiver.

- **MACAW** — *MACAW* (Multiple Access with Collision Avoidance for Wireless) is a dynamic allocation protocol designed for wireless networks proposed by Bharghavan *et al.* [34], and is an extension of the earlier *MACA* (Multiple Access with Collision Avoidance) protocol proposed by Karn [35].

  In MACA, a user intending to transmit sends a short RTS (Request To Send) packet, containing the length of the data to be sent, to the receiver before attempting transmission. The receiver responds with a short CTS (Clear To Send) packet, also containing the data length specified in the RTS packet. Any user that detects either the RTS or CTS packets refrains from transmitting, for the time period needed to transmit the amount of data specified in the packets.

  MACAW improved upon MACA by adding an ACK (Acknowledgement) packet

after each successfully transmitted data packet, to ensure faster retransmission of lost data packets at the data link layer, rather than at higher layers. Carrier sensing was also added to prevent multiple users simultaneously attempting to transmit an RTS packet to the same receiver, thus providing MACAW with many of the benefits of pure CSMA protocols. The *CSMA/CA* (Carrier Sense Multiple Access with Collision Avoidance) protocol used in 802.11/WiFi networks is based on MACAW.

- **IEEE 802.4/Token Bus** — *Token Bus* [17] is a dynamic allocation protocol in which the users are organized into a logical ring, rather than a physical ring as in the related *IEEE 802.5/Token Ring* network [17]. In both the token bus and token ring networks, each node is aware of the node immediately before and after it in the ring. Bus arbitration is achieved through the use of a *token* packet, which is passed from user to user in sequence around the ring. Only the user in possession of the token packet may transmit data, if any, after which the token is passed to the next user.

  Unlike the logical ring in the token bus network, the physical ring in the token ring network has a weakness, in that each user is able to communicate directly only with the two users before and after it in the ring. Each user is therefore a point of failure for the entire network. The token bus network avoids this limitation since all nodes are connected to each other using a multi-access communication medium. However, scalability is limited in both the token bus and token ring networks since the delay between successive possessions of the token per user increases as additional users enter the system.

The MAC protocol utilized in the modular sensing system software architecture is a dynamic allocation protocol based on MACAW. The MAC protocol was derived from MACAW because its acknowledgement and carrier sensing features are already imple-

mented in hardware within the nRF24L01 transceiver, facilitating improved performance. In addition, MACAW does not depend on global time synchronization between contenders for the medium in order to operate reliably, which is important because much of the communication layer is implemented in a concurrently executing task running on a preemptive real-time operating system (see Section 2.4). As a result, the task executes at unpredictable intervals, making reliable global synchronization very difficult to achieve. The operation of the MAC protocol is further detailed in Sections 3.6.1 and 3.6.2.

3.2.3 Time Synchronization

The timers used within each module to generate and compare timestamps need to be regularly synchronized during operation. Regular synchronization is necessary since the resonant frequency of the crystal oscillator that controls the local time of each module may be slightly different from its rated value, or may shift slightly over time. Slight imperfections in the crystal manufacturing process cause the discrepancy in rated value, while microscopic changes in the crystal size due to environmental effects such as temperature, humidity, and pressure result in the slight shifts in resonant frequency. These variations in resonant frequency result in varying degrees of *clock drift* between module clocks, and in turn cause a loss of synchronization between the local times of each module. System reliability is therefore reduced, since the reported time of occurrence of an event by a particular module may not necessarily be accurate with respect to the local time of another module.

The protocol used for time synchronization is based on the *Simple Network Time Protocol* (SNTP) developed by Mills [36], which is a subset of the *Network Time Protocol* (NTP) also developed by Mills [37]. Both SNTP and NTP are standard, well-known protocols widely used to synchronize computer clocks over the Internet. The derived protocol is further detailed in Sections 3.6.1 and 3.6.2.

As described in [36], and shown in Figure 3.2, four 64-bit timestamps are needed to calculate the signed *clock offset* θ between differing module clocks. These four timestamps may also be used to calculate the *roundtrip delay* δ of exchanged packets, and each timestamp is relative to the clock of the module on which it was taken. The first timestamp is the *Originate* timestamp T_{i-3}, which indicates the time that a synchronizing module requested synchronization with a remote module. The second timestamp is the *Receive* timestamp T_{i-2}, which indicates the time the remote module received the synchronization request. The third timestamp is the *Transmit* timestamp T_{i-1}, which indicates the time that the remote module responded to the synchronizing module. The fourth timestamp is the *Destination* timestamp T_i, which indicates the time that the synchronizing module received the synchronization response.

The *Originate* and *Destination* timestamps are temporarily recorded in volatile memory by the synchronizing module once taken, while the *Receive* and *Transmit* timestamps are returned to the synchronizing module. The propagation delay of the exchanged synchronization packets is assumed to have remained constant over the negligibly short period of time during which they were exchanged. The roundtrip delay δ, and the signed clock offset θ which is added to the local clock of the synchronizing module, are then accurately calculated by the synchronizing module using Equations 3.1 and 3.2, derived by Mills [37].

$$\delta = (T_i - T_{i-3}) - (T_{i-1} - T_{i-2}) \tag{3.1}$$

$$\theta = \frac{(T_{i-2} - T_{i-3}) + (T_{i-1} - T_i)}{2} \tag{3.2}$$

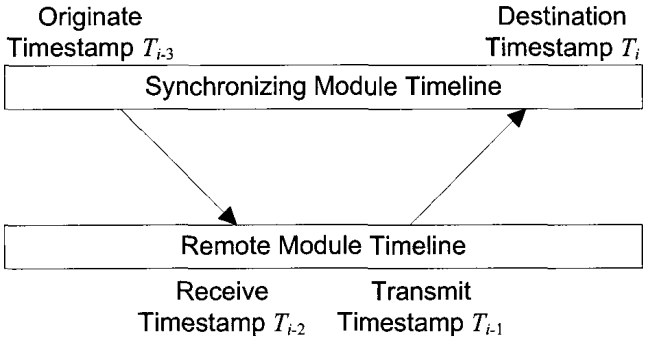


Figure 3.2: Time synchronization packet exchange.

3.2.4 Wireless Security

Unlike wired transmission mediums such as twisted-pair Ethernet cables, wireless transmission is inherently insecure since the modulated radio signals used for data transmission are easily intercepted by any individual possessing a tuner of the appropriate frequency. Wireless security is provided within the communication layer in the form of *encryption* in order to provide confidentiality when privacy of information pertaining to the identification of TIMs and their collected data (which is frequently transmitted wirelessly between modules) is required.

Two common cryptographic algorithms that facilitate information security are *stream ciphers* and *block ciphers*. In a stream cipher, the bits comprising information to be transmitted are combined with a pseudorandom *keystream* of *cipher bits* through the use of the *exclusive-or* (XOR) logical operation, as depicted in Figure 3.3. The cipher bit stream varies with a *key* used to initialize the algorithm. In practice, one pseudorandom byte is generated and used to encrypt one byte of data at a time within each iteration of a stream cipher encryption loop. As a result, stream ciphers are often utilized in applications such as wireless transmission, where the information to be encrypted is of an indeterminable length. Due to the bit-inverting nature of the XOR operation when its input and corresponding output are combined with the same bit sequence, encrypted data

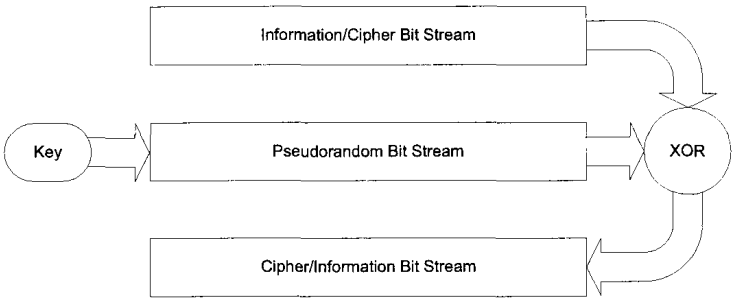


Figure 3.3: Stream cipher operation block diagram.

can also be decrypted using the encryption algorithm itself. The most popular stream cipher currently in use is *Rivest Cipher 4* (RC4), which is also referred to as *Alleged Rivest Cipher 4* (ARC4) [32] in its publicly available reverse-engineered implementation in order to limit trademark concerns. ARC4 is the encryption algorithm used within the *Wired Equivalent Privacy* (WEP) encryption protocol [38], which is widely utilized in IEEE 802.11 wireless networks.

In a *block cipher*, depicted in Figure 3.4, information is processed in fixed-length groups of bits known as *blocks*, which are typically much larger than one byte, thus requiring the length of the information provided for encrypting to be a multiple of the block size. An pair of complementary *transformation functions* are used for encryption and decryption, the behaviour of which is unique to a supplied *key*, and are applied to information blocks to produce encrypted blocks, and to encrypted blocks to produce information blocks, respectively. However, a block cipher may operate within various standard *modes of operation*, some of which enable a block cipher to effectively operate as a stream cipher. In these stream-based modes, the encryption transformation function is instead applied to a sequence of values, which may be as simple as an incrementing counter, that are guaranteed not to repeat for an extensive period of time to produce *keystream blocks*. These keystream blocks are then combined with the bits comprising

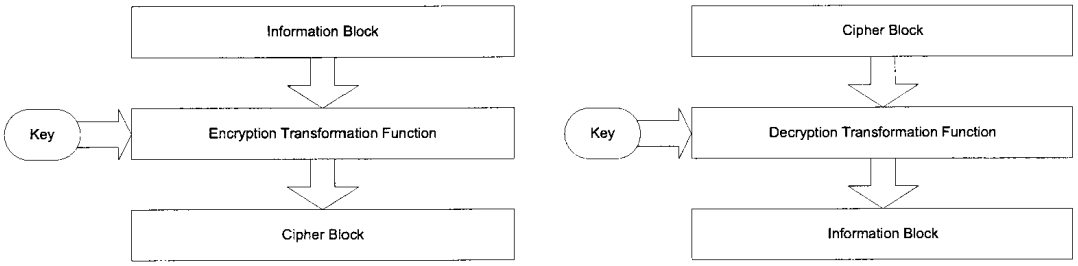


Figure 3.4: Block cipher operation block diagram.

the information to be transmitted using the XOR logical operation, as in a stream cipher. The encryption transformation function may be used for both encryption and decryption in this case, as with stream ciphers. Block ciphers are typically slower than stream ciphers and often require the usage of more memory during their operation; however, block ciphers often facilitate the creation of encryption algorithms that provide a greater degree of security compared to stream ciphers. Popular block ciphers currently in use include *Blowfish* [39] and the *Advanced Encryption Standard* (AES) [40], which is based on the *Rijndael* algorithm. AES is adopted for encryption purposes by the United States government, and is also the encryption algorithm utilized in the *Wi-Fi Protected Access* (WPA) protocol [41], which has superseded WEP as the encryption standard of choice for IEEE 802.11 networks due to major weaknesses that have been discovered in WEP.

All packets generated for transmission by the software architecture are encrypted using the ARC4 encryption algorithm due to its straightforward implementation, excellent speed, minimal memory usage, and relatively strong security. These criteria are important due to the resource-constrained hardware present in the TIMs. The ARC4 pseudorandom keystream generator is depicted in Figure 3.5. It utilizes two 8-bit indices i and j that are initialized to zero, and an array S containing a permutation of all 256 possible bytes, which is initialized by applying a *key scheduler algorithm* to a variable length key of up to 128 bits. In each iteration of the ARC4 pseudorandom keystream generator i is

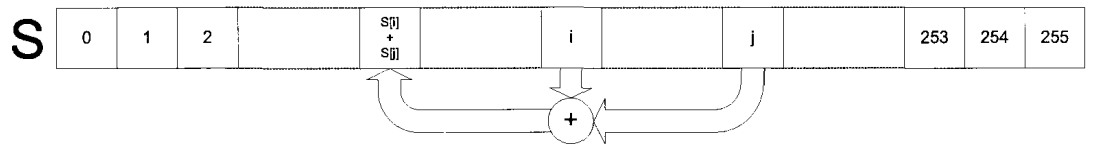


Figure 3.5: ARC4 pseudorandom keystream generator operation.

incremented, after which the value at $S[i]$ is added to j. The values at $S[i]$ and $S[j]$ are then swapped, and the byte at the index $S[S[i] + S[j]]$ is combined with the current byte in the input byte stream using the XOR operation to produce the output byte.

The use of encryption also serves to reduce the bit error encountered during transmissions in which long runs of unchanging bits are present, since the information stream is always replaced by a constantly varying pseudorandom bit pattern before transmission. Long streams of unchanging bits greatly increase the difficulty encountered by the nRF24L01 transceiver in locking onto transmitted radio signals due to its use of *Gaussian Frequency Shift Keying* (GFSK) modulation for transmitting and receiving data [21], in which the frequency of the signals is directly related to the value of the bit being transmitted.

3.3 Packet Format

Data is transferred to and from the transceiver driver in 329-bit *packets*. The packet format, shown in Figure 3.6, is designed to be compatible with that of the nRF24L01 transceiver, which manages the *preamble*, *network identifier*, *packet control field*, and *firmware checksum* fields within its firmware. The 32-byte payload field defined within the nRF24L01 packet format is sub-divided into smaller fields for the purposes of the software architecture. The communication layer packet fields are described as follows:

- **Preamble** (8 bits) — A pattern of alternating ones and zeroes used by a receiving

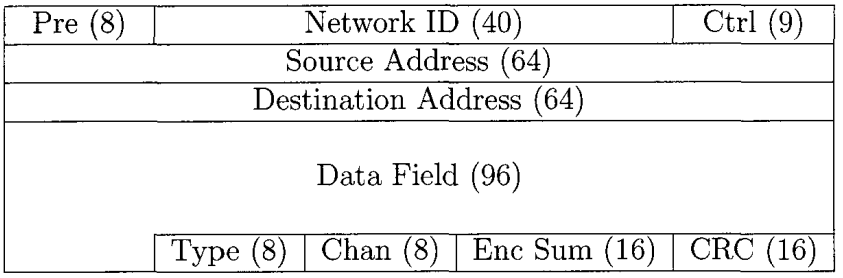


Figure 3.6: Communication layer packet format (field sizes in bits).

transceiver to synchronize its clock with that of the transmitting transceiver.

- **Network Identifier** (40 bits) — A constant value common to all or a subset of modules. Any received packet that does not specify this identifier is automatically rejected by the transceiver firmware.

- **Packet Control Field** (9 bits) — A field used internally by the nRF24L01, which contains the *packet identification* (PID) number used in detecting packet retransmissions as well an *acknowledgement flag* indicating whether or not a particular packet requires acknowledgement.

- **Source Address** (64 bits) — Identifies the physical or logical module that transmitted the packet.

- **Destination Address** (64 bits) — Identifies the physical or logical module that should receive the packet.

- **Data Field** (96 bits) — Contains the data to be transmitted within the packet. Therefore, the *maximum transmission unit* (MTU) of the communication layer is defined to be 96 bits. The data field may be further sub-divided into *parameter fields* used for transmitting various types of data specific to the packet type.

- **Packet Type** (8 bits) — Indicates the type of the packet, through which the

method of interpreting the data field may be determined.

- **Packet Channel** (8 bits) — Indicates the channel on which the packet was transmitted.
- **Encryption Checksum** (16 bits) — Simple checksum used to verify decrypted packets. The checksum is generated by a summation of the 30 bytes comprising the source address to the packet channel. The use of this 16-bit checksum greatly reduces the chance of misidentifying a packet decrypted with an invalid key as valid data.
- **CRC Checksum** (16 bits) — Used to detect packet transmission errors. The checksum is automatically generated by the transceiver firmware using the *Cyclic Redundancy Check* (CRC) algorithm [17].

3.4 Channels and Packet Types

The nRF24L01 transceiver is able to transmit and receive packets on any one of up to 125 distinct radio frequency channels at a time, one of which is reserved by the software architecture for use as a *control channel.* All modules listen to the control channel by default when not transmitting data, and each module can detect the presence of others in its vicinity by listening for packet transmission activity on the channel.

The other 124 channels are utilized as *data channels.* Upon successful reservation of a data channel through the use of the RTS and CTS *medium allocation packets* (described in Sections 3.6.1 and 3.6.2), the transmitting and receiving modules switch to the agreed channel and carry out the transmission. As depicted in Figure 3.7, lengthy transmissions may occur simultaneously on different channels without interference. The various packets types defined for use by the software architecture and transmitted on the control and data

Control Ch.	CTS	RTS	CTS		PRE	
Data Ch. 1		DAT	DAT	DAT	DAT	
Data Ch. 2				DAT	DAT	DAT
			⋮			
Data Ch. 124	DAT	DAT	DAT	DAT		

Figure 3.7: Multi-channel operation.

channels are described below.

3.4.1 Control Channel Packet Types

- **PRE** — *Presence* packets are regularly broadcast by all modules to indicate their continued presence in the sensing system, as well as to facilitate determination of their basic properties. The following module properties are specified in the data field: *type* (sensor or actuator), *class* (temperature, pressure, display, etc.), *data type* (signed or unsigned integer, single-precision or double-precision floating point value, status constant, string, or raw bytes), *data type array width*, and *data type array height*. Also specified in the data field are the *synchronization level* of the module, its *connection type* (local, physical, or wireless; see Section 6.2.2), and a packet timeout counter.

- **MEM** — *Member* packets are regularly broadcast by modules to indicate their continued presence in a particular combination of modules comprising a *logical module*. The hardware address of the module, its *role identifier* within the logical group, as well as a packet timeout counter are specified in the data field.

- **SYQ** — *Synchronization Query* packets are used to initiate time synchronization with a remote module and transfer the *Originate* timestamp in the data field. The synchronization protocol used is based on the *Simple Network Time Protocol*

(SNTP) [36].

- **SYR** — *Synchronization Response* packets are issued in response to a time synchronization request, and contain within the data field the *synchronization level* of the synchronizing module, and a partial calculation used by the unsynchronized module to calculate its relative time offset.

- **RTS** — *Request To Send* packets are used to request transmission of a middleware layer message. The data channel to be used, the *network identifier offset* (an offset to the network identifier used exclusively for the transfer), as well as the message length in bytes, are specified within the data field.

3.4.2 Data Channel Packet Types

- **CTS** — *Clear To Send* packets are issued by a receiving module to the transmitting module to indicate that it may proceed with the transmission.

- **DAT** — *Data* packets contain consecutive 12-byte fragments of a middleware layer message in the data field.

3.5 Initialization

The communication layer initialization process begins with the allocation of the two main vector data structures utilized by the layer. The first is the *environment list*, which stores presence packets detected on the wireless channel, and is used to keep track of the modules within the vicinity as well as their basic properties. The second is the *outgoing message queue*, which is a FIFO (first in, first out) queue that stores messages received from the middleware layer waiting to be transmitted.

To determine which network identifier and ARC4 key the module is to utilize for packet transmissions, the SD card is searched for the 5-byte network identifier file `net.id` and the variable length (up to 16 bytes) key file `key.rc4` respectively, as described in Section 2.5.2. Transmissions from modules with different network identifiers are ignored, and packets received from modules utilizing a different ARC4 key will be indecipherable upon reception and are dropped. If no `net.id` file is found, the default network identifier `0x40414D5353` (which represents the ASCII sequence "@AMSS") is used. If no `key.rc4` file is found, the default 128-bit key `0x414D5353414D5353414D5353414D5353` (which represents the ASCII sequence "AMSSAMSSAMSSAMSS") is used. Upon determining the network identifier and ARC4 key to be used, the wireless transceiver starts listening for transmissions on the control channel.

3.6 Network Communication Task

The *network communication task* is started upon initialization of the software architecture and runs continuously and concurrently with all other tasks in the system. At the heart of the task is an infinite loop in which a number of operations are carried out at different time intervals. These operations are multiplexed into a single task instead of being split into separate tasks due to the limited 40 kilobytes of RAM (random access memory) available on the LPC2148 microcontroller present in the TIMs, which places constraints on the amount of stack space available to be distributed between concurrently executing tasks. The network communication task is required to:

- Transmit presence packets for all modules in the module structure list every two to five seconds.
- Transmit member packets for all modules in the module structure list every two to five seconds.

- Decrement the timeout counters of all presence packets in the environment list and all member packets in the *role environment lists* (see Section 6.2.2) of all logical modules every second.
- Synchronize the local clock with a remote clock of a lower *synchronization level* (see Section 2.7.3) every one to two minutes.
- Perform garbage collection of shut down logical modules and their associated module structures and message handler task structures every two seconds.
- Transmit a single outgoing message, if any is pending, on each loop iteration.
- Handle up to five pending control packets, if any were received as indicated by a transceiver interrupt, on each loop iteration.

The general operation of the network communication task is depicted in Figure 3.8. The nRF24L01 wireless transceiver is locked for the duration of each operation to prevent other tasks from simultaneously modifying its registers. Most of the operations in the network communication task are performed at randomly determined intervals in order to reduce the possibility of all the modules in the environment regularly saturating the control channel at similar times, while otherwise leaving the channel empty. The various standard handlers and packet handlers that perform the previously mentioned operations are further outlined in the following sections.

3.6.1 Standard Handlers

Presence Handler

The *presence handler*, invoked every two to five seconds, is responsible for transmitting the presence packets which indicate that a module is within a particular network. These presence packets also provide a basic overview of the attributes and capabilities of the

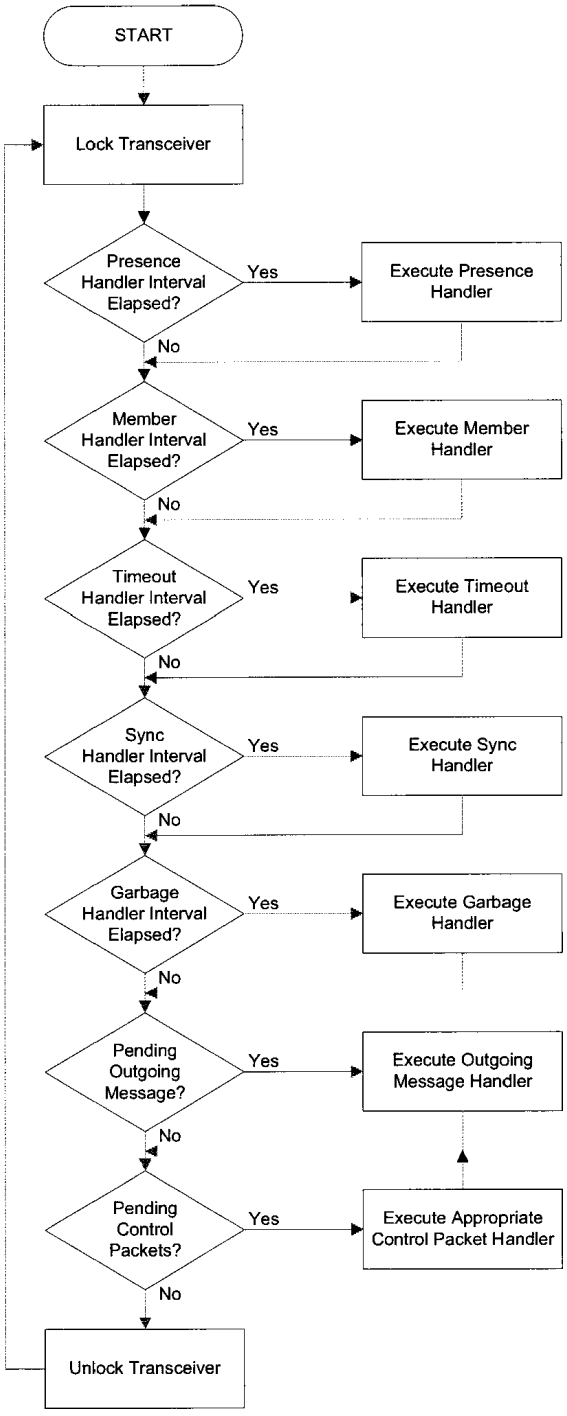


Figure 3.8: Network communication task operation.

module. For each physical module structure or *primary* module structure (which possesses the lowest address in a logical module entity and is responsible for transmitting and processing its messages) found in the module structure list, a presence packet is transmitted through the wireless transceiver and also handled locally. Since presence packets are broadcast packets, the destination field of these packets instead serves the purpose of indicating the *pose base* (see Section 6.5) of the module, which is the address of the remote module to which its pose is relative.

Due to the small size of the nRF24L01 FIFO incoming packet buffer, which is three packets deep, too many presence packets transmitted by one module in rapid succession may result in all presence packets transmitted after the third or fourth to go undetected. In order to account for this problem, each successive search through the module structure list is offset by one, to ensure that the first three presence packets transmitted are almost always in a different order.

Member Handler

The *member handler*, invoked every two to five seconds, is responsible for transmitting the member packets that indicate which logical module entities a particular module is a member of. As mentioned in Section 2.7.3, each module structure possesses a *membership list*, indicating the *roles* it fulfils in the logical modules of which it is a member. This role information is broadcast within the member packets.

For each module structure found in the module structure list, appropriate member packets are transmitted for each role it satisfies in a logical module, which are also handled locally. Similar to the presence handler, successive searches through the module structure list as well as the member-of list are offset by one on each iteration. This ensures that the latter packets in a group of member packets transmitted in rapid succession do not repeatedly get dropped while being placed in the FIFO incoming packet buffers of the

transceivers on other modules.

Timeout Handler

The *timeout handler* is invoked every second. During each invocation, the environment list is scanned and each presence packet found in the list has its timeout counter field decremented. If any timeout counter reaches zero, the presence packet is removed from the environment list and the corresponding module is considered to have left the environment.

The module structure list is then searched for module structures corresponding to logical modules that contain member packets in their *role environment lists.* A role environment list stores the member packets corresponding to the TIMs in the environment that fulfil a particular role in the logical entity. These member packets also have their timeout counters decremented. If the timeout counter of any packet reaches zero, it is removed from its respective role environment list. Removal from the role environment list indicates that the module that transmitted the member packet is no longer considered a member of the logical entity.

Synchronization Handler

The *synchronization handler*, invoked every one to two minutes, is responsible for initiating time synchronization between modules. To determine if synchronization is necessary, the presence packets in the environment list are searched to determine which module has the lowest *synchronization level.* If multiple modules possess the lowest synchronization level, then the lowest address of these is used for synchronization.

Synchronization is only performed if a module with a lower synchronization level is found in the environment, or a module with an equivalent synchronization level, but a lower address, is found. If synchronization is necessary, a synchronization query (SYQ) packet possessing the *Originate* timestamp T_{i-3}, as described in Section 3.2.3, is trans-

mitted on the control channel to the module with the lowest synchronization level that also possesses the lowest address among modules at that synchronization level.

Garbage Handler

The *garbage handler* is invoked every two seconds to reclaim memory allocated by shut down logical modules and their associated task structures and module structures. Garbage collection is necessary since individual tasks cannot deallocate their stack and heap space on their own upon completing their execution. Message handler tasks (see Section 4.5) strictly associated with any of the transducers on the TIM on which it runs are required to execute as long as the TIM is powered, and therefore require no garbage collection. To determine if any logical module message handler tasks and structures need to be garbage collected, the *shutdown flags* of all the module structures corresponding to logical modules in the module structure list are checked. This flag is clear by default, and is only set immediately before the message handler task associated with a module structure completes its execution. If set, the corresponding presence packet for the module in the environment list is removed, the module structure itself is deleted from the module structure list, and its stack and heap memory is reclaimed.

Outgoing Message Handler

The *outgoing message handler* is invoked once in each iteration of the main network communication task loop. Upon its invocation, the front of the outgoing message queue is checked to determine if a message to be transmitted is pending. If a message is pending, its destination address is checked to determine if the destination module structure is local to the module hardware. In this case transmission would be unnecessary and the message is simply moved to the incoming message queue of the destination module structure, otherwise the *message transmission mechanism* is invoked. Only a single message is

transmitted in each invocation of the outgoing message handler in order to minimize the duration of a single iteration of the main network communication task loop, thus improving the latency encountered by the other handlers.

In the message transmission mechanism, a *request to send* (RTS) packet is prepared for medium access purposes. It assumes the same source and destination addresses of the message to be transmitted, a random *network identifier offset*, and the channel number of the first free data channel found after carrier sensing. Handshaking is attempted ten times. In each attempt, the channel is switched to the free data channel and the network identifier offset is temporarily added to network identifier in order to minimize packet handling overhead by the communication layer should two modules end up transmitting on the same data channel. The RTS packet is then transmitted and the module waits up to 500 ms for the corresponding *clear to send* (CTS) packet from the receiver to arrive. This handshaking process is repeated until the CTS packet is received and transmission may proceed safely.

If the handshaking process is successful, the message is then broken into a number of data (DAT) packets that are transmitted sequentially to the receiver. Upon successful transmission, or any error, the wireless channel is set back to the control channel and the network identifier is set back to its initial value.

3.6.2 Control Packet Handlers

In the main network communication task loop, up to five pending packets are handled within each iteration. Depending on the value found in its *packet type* field, each packet is handled by one of six *control packet handlers*, which are described below.

PRE Handler

Before the received *presence* (PRE) packet is processed, its *connection type* field is changed from *wireless* to *physical* if the *pose base* (see Section 6.5) found in its *destination field* is the same as that of the module. Possessing the same pose base as another module indicates that a direct or indirect physical connection exists to that module. If a matching presence packet is already present in the environment list, its timeout counter value is reset to the standard environment timeout time of thirty seconds. If the connection type of the packet was modified, a *logical module template search* is carried out for a new *logical module match* (see Section 6.3). If the presence packet is not found in the environment list, the packet is added to the list and a logical module template search is also carried out.

MEM Handler

On receiving a *member* (MEM) packet, its destination module is determined. Since member packets are only relevant to logical modules, the packet is dropped if its destination is not a logical module. If the destination is found, and it is a logical module, the *role member environment list* of the role corresponding to the *role number* field of the member packet is located. The list is then searched and if a corresponding member packet is already present, it is overwritten and its timeout counter reset to the standard environment timeout time of thirty seconds. If the member packet is not found in the role member environment list, the packet is added to the list.

SYQ Handler

On receiving a *synchronization query* (SYQ) packet, a timestamp of the local clock, the *Receive* timestamp T_{i-2}, is immediately acquired and stored. The *Originate* timestamp T_{i-3} is also extracted from the SYQ packet. A *synchronization response* (SYR) packet

is then returned to the source, containing a newly acquired *Transmit* timestamp T_{i-1} timestamp in the form $T_{i-2} - T_{i-3} + T_{i-1}$, as well as the *synchronization level* of the module. This partial calculation is employed since there is not enough space in the 12-byte data field to transmit both T_{i-2} and T_{i-1} separately within a single SYR packet. The partial calculation also removes the need to transmit two SYR packets, each containing one of the timestamps.

SYR Handler

On receiving a *synchronization response* (SYR) packet, a timestamp of the local clock, the *Destination* timestamp T_i, is immediately acquired and stored. The partial calculation $T_{i-2} - T_{i-3} + T_{i-1}$ is also extracted from the SYR packet. The local clock is then updated through the addition of an offset calculated as $(T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$. The *synchronization level* of the module is subsequently updated to be one more than that specified in the received SYR packet, up to a maximum of 255.

RTS Handler

On receiving a *request to send* (RTS) packet, the *message reception mechanism* is invoked. In this mechanism, the destination module structure of the received RTS medium allocation packet is determined. The request is only handled if the destination address corresponds to a physical module or a *primary* module (which possesses the lowest address in a logical module entity of which it is a member). If a suitable destination is found, the wireless channel is switched to that specified in the RTS packet and the *network identifier offset* specified in the RTS packet is also temporarily added to the network identifier. A *clear to send* (CTS) packet is then transmitted to the source of the RTS packet to indicate that communication may proceed. The CTS packet itself requires acknowledgement and is sent on the data channel to avoid possible interference on the control channel. If

the CTS packet is acknowledged, a buffer large enough to store the incoming message is allocated in memory and transmission proceeds.

The incoming DAT packets from the transmitter, which contain consecutive fragments of the incoming message, are used to locally reconstruct the message within the allocated buffer. Upon successful transmission of all of the DAT packets, or any error, the wireless channel is set back to the control channel and the network identifier is set back to its initial value. If the transmission was successful, the recombined message is then moved to the incoming message queue of its destination module structure.

3.7 Face Connectivity

Provided within the communication layer is a wired protocol that facilitates the direct transmission of data through the faces of physically connected TIMs. The operation of this protocol depends on the electrical contacts present on the four clips located on five of the six faces of a TIM. In order to facilitate the detection of the relative angular offset between two connected TIMs, the TIM faces as well as the electrical contacts located on them are each assigned an *identifier*. Each TIM face is assigned an identifier from 1 to 6, while each face contact is assigned an identifier from 1 to 4. These identifier assignments are shown in the layouts depicted in Figures 3.9 and 3.10. The design of the face communication protocol is such that a TIM can determine the address of any other TIM it is physically connected to, as well as the identifiers of the connected faces and connected contacts between itself and these TIMs. The core data elements facilitating face connectivity are the *face structure*, the *face identification packet*, and the *face communication task*, which are described in the following subsections.

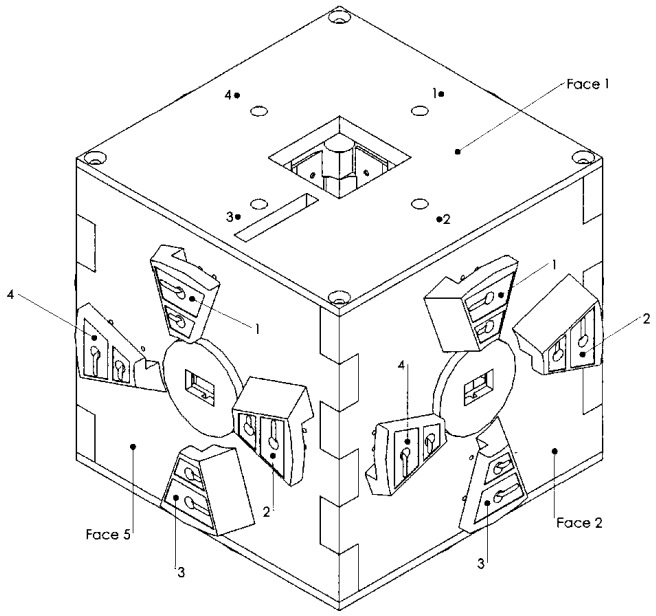


Figure 3.9: Three-dimensional face and contact identifier layout view.

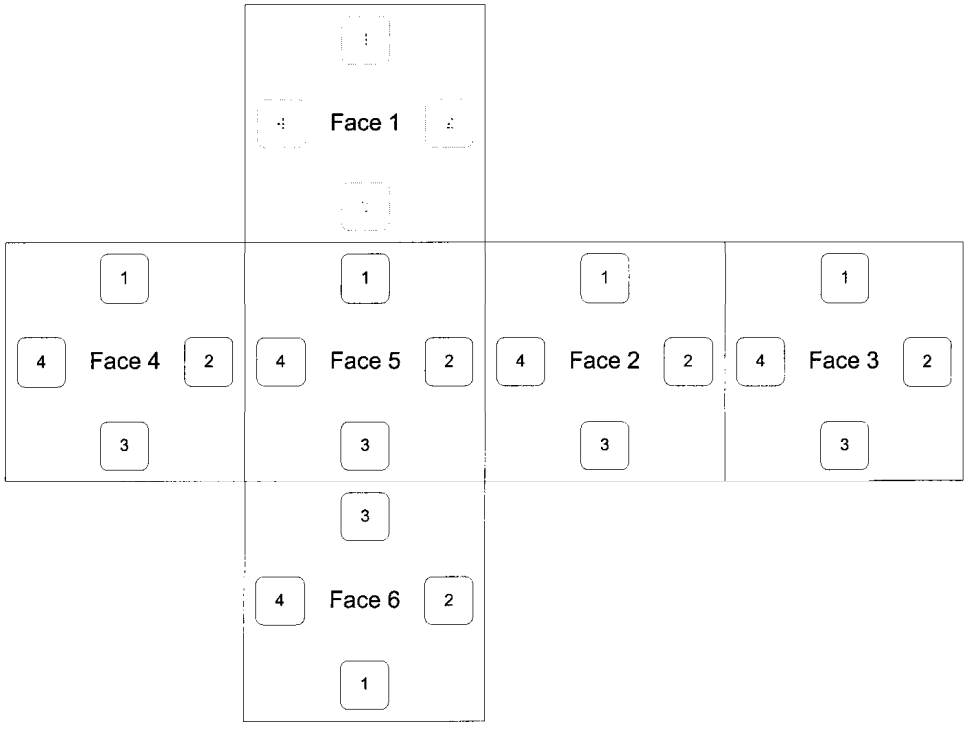


Figure 3.10: Two-dimensional face and contact identifier layout view.

3.7.1 Face Structure

A *face structure* stores the connection state of one of the five faces on a TIM to which another TIM may be connected. Each TIM maintains five face structures in memory, each of which corresponds to one of its faces on which contacts are present. The values contained within a face structure are defined as follows:

- **Face Transform Matrix** — A 4×4 matrix of 32-bit floating point values used to store the transformed state of the face represented by the face structure, relative to its original orientation. Only 48 bytes are actually utilized to store the matrix instead of 64 bytes since the fourth row is understood to always be [0 0 0 1]. The face transform matrix is currently always the identity matrix, since the faces of a TIM are rigid in its current implementation.

- **Remote Address** — The address of the remote module physically connected to the local face represented by the face structure.

- **Remote Face Identifier** — The identifier of the face on the remote module which is physically connected to the local face represented by the face structure.

- **Local Face Contact Identifier** — The identifier of the local contact on the face represented by the face structure through which the last face identification packet was received from the TIM connected to it. Knowledge of this identifier facilitates the detection of the relative angular offset between the connected TIMs (see Section 3.7.3).

- **Timeout Counter** — Indicates the remaining time during which the information contained within the face structure is considered valid. If this counter expires, the face structure is reset to represent an unconnected state. This counter is initialized to 30 seconds.

3.7.2 Face Identification Packet Format

Face identification information is transferred between faces in the form of 20-byte *face identification packets*, the format of which is depicted in Figure 3.11. The information transferred within these packets reveals of the address of the physically connected remote module as well as the identifier of the connected face through which the packet was received. Examination of the local contact through which the packet is received also enables the relative angular offset between the connected TIMs to be determined.

These packets are transmitted on each face of a TIM at regular intervals to serve as a form of *watchdog timer*, indicating the continued presence of a physical connection on the respective face to another module. The packets are transmitted unencrypted since wired transmissions are not easily intercepted, and the data being transmitted is only critical to orientation determination and the detection of the physical connection and disconnection of modules. The face identification packet fields are described as follows:

- **Header** (4 bytes) — Identifies the packet as a valid face identification packet. The header is defined as the byte pattern `0x414D5353` (which represents the ASCII sequence "AMSS").

- **Remote Address** (8 bytes) — Identifies the address of the remote module that transmitted the face identification packet, which would place its own address in this field.

- **Remote Face Identifier** (4 bytes) — Indicates the identifier of the face on the remote module through which the face identification packet was received. Although a single byte would suffice to represent this information, an additional three bytes are reserved for future expansion.

- **Checksum** (4 bytes) — Used to detect packet transmission errors. The checksum is a simple summation of the bytes comprising the header, remote address, and

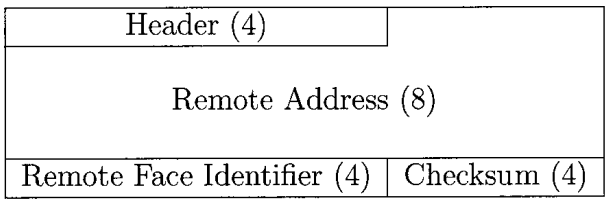


Figure 3.11: Face identification packet format (field sizes in bytes).

remote face identifier. Although two bytes would suffice to store the checksum, an additional two bytes are reserved for future expansion.

3.7.3 Face Communication Task

Like the network communication task, the *face communication task* is started upon initialization of the software architecture and runs continuously and concurrently with all other tasks in the system. The operations performed by the task are carried out within an infinite loop at different time intervals. The general operation of the face communication task is depicted in Figure 3.12. The face communication task is required to perform the following operations, which are further outlined in the following subsections:

- Decrement the timeout counters of all five face structures every second, and trigger an update of the local pose of the module if necessary.
- Transmit face identification packets on each face indicating the address of the module and the respective face identifier every five to ten seconds.
- Receive pending face identification packets, if any, from the remote modules connected to each face on each loop iteration, and trigger an update of the local pose of the module if necessary.

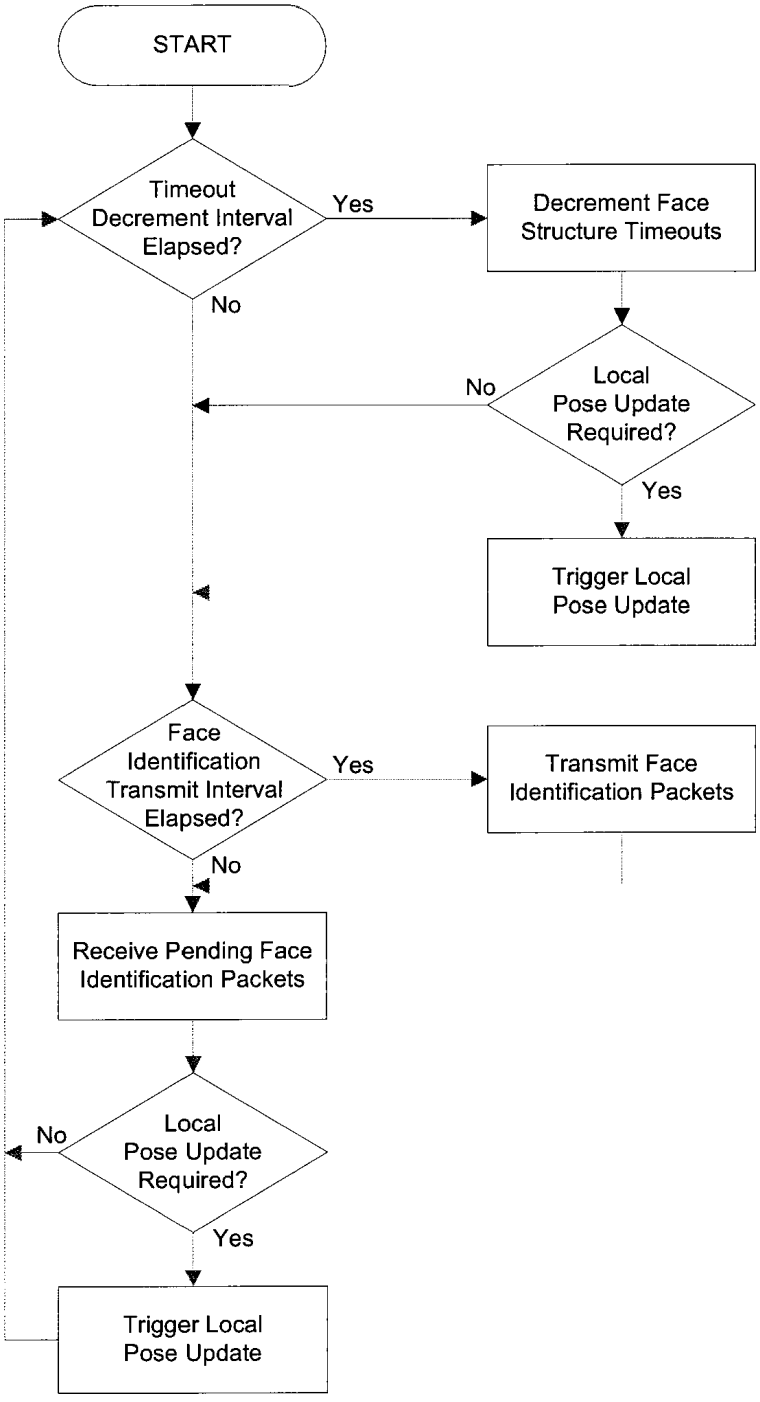


Figure 3.12: Face communication task operation.

Decrementing Timeout Counters

The timeout counters associated with each face structure are decremented once per second. If any counter expires, each field within the associated face structure is reset to a *null* value, representing an unconnected state. If this occurs, a flag maintained by the face communication task is set to indicate that the local *pose* of the module requires updating, since the local pose may be relative to that of the remote module that was disconnected. The pose update process is described in Section 6.5.

Transmitting Face Identification Packets

Face identification packets are transmitted on each face every five to ten seconds, indicating the continued presence of a physical connection on the respective face to another module. Data signals are transmitted in a format similar to that of *RS-232* [42], in that transmissions are composed of an asynchronous timed series of bits.

The signals transmitted on each contact are depicted in Figure 3.13. Before transmission on each face, a face identification packet is allocated and its checksum determined and stored within the packet itself. The face contacts, which are normally configured to receive data, are temporarily configured to generate data. To indicate to the remote module that a transmission is about to occur, a *start symbol* is transmitted. This symbol consists of setting all the contacts on a face to the high logic level for a period of 20 ms, then clearing all the contacts for the duration of one bit length, which is approximately 1.67 μs. The contact with identifier 1 is then set high for one bit length to indicate to the remote module not only the contact on which data will be transmitted, but also the relative angular offset between the two faces, as shown in Table 3.1. The bits of the packet are then transmitted sequentially through Contact 1, starting with the least significant bit, after which the face contacts are reconfigured to receive input.

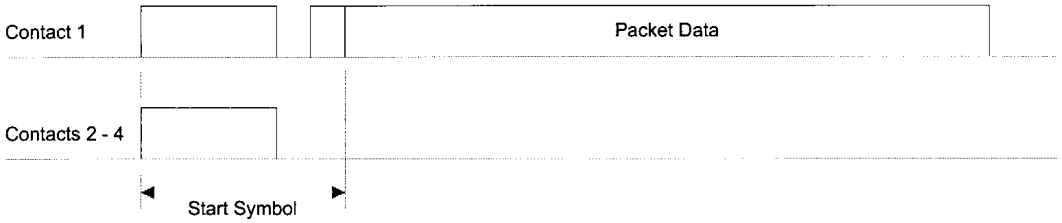


Figure 3.13: Face contact transmission signals.

Table 3.1: Face contact connection patterns and corresponding angular offsets.

Face Contact Connection Patterns					Local Face Angular Offset Relative to Remote Face
Local Face Contact Pattern	**1**	2	3	4	—
Remote Face Contact Patterns	**1**	4	3	2	0°
	2	**1**	4	3	90°
	3	2	**1**	4	180°
	4	3	2	**1**	−90°

Receiving Face Identification Packets

Each face on the module is checked for incoming face identification packets. Reception of a face identification packet on a face currently represented by its corresponding face structure as being in an unconnected state indicates that a new connection has occurred. An incoming packet is indicated by high logic levels being detected on all of the contacts on the face, as set by the remote module transmitting the packet during the *start symbol*. As described in Section 3.7.3, the contact on which data will be transmitted is detected through examination of the start symbol. Data reception only proceeds if a valid start symbol is detected.

Upon detection of a valid start symbol, reception is delayed for half of a bit length. This improves the reliability of the data transfer by ensuring that detection of logic levels occurs as far away from logic level transitions as possible, the optimum location of which is half-way through the length of a bit transmission. The bits comprising the

face identification packet are then read sequentially, starting with the least significant bit, after which the packet is reconstructed in memory. If the header and checksum of the packet are valid, the face structure corresponding to the face on which the packet was received is updated with the *remote address* and *remote face identifier* provided within the packet. Also updated within the face structure are the *local face contact identifier* field, in order to reflect the contact on which the packet was received, and the *timeout counter*, which is reset to 30 seconds. If the address of the local module is lower than that of the remote address, the flag maintained by the face communication task to indicate that the local *pose* of the module requires updating is set.

3.8 Summary

In this chapter, the *communication layer* of the software architecture was described. The communication layer provides a secure, reliable, connection-oriented interface to the unreliable wireless transmission medium. Data is transmitted within 329-bit *packets* that are encrypted using the *Alleged Rivest Cipher 4* (ARC4) stream cipher. 125 wireless channels are available for packet transmission purposes, one of which is reserved for use as a *control channel*; the others are utilized as *data channels*. At the core of the communication layer is the *network communication task*, which manages the transmission and reception of individual packets, as well as performing important duties such as time synchronization and garbage collection. Also present is the *face communication task*, which implements a wired protocol that facilitates the direct transmission of data through the faces of physically connected TIMs. The following chapter will describe the operation of the *middleware layer*.

Chapter 4

Middleware Layer

4.1 Introduction

The purpose of the *middleware layer* is to facilitate interoperability between the various TIMs in a modular sensing system. The term *middleware* refers to software and services that simplify connectivity between software components running on distinct and possibly heterogeneous devices, in turn simplifying the deployment of distributed applications. At the middleware layer in this software architecture, the *application programming interface* (API) for physical and logical modules is defined, which is comprised of a variety of *service functions*. Service functions are the interface through which TIMs, whether homogenous or heterogeneous, request services from, and information about, each other. Data is transferred between TIMs in the form of variable-length *messages*.

4.2 Middleware Types

Middleware implementations exist in a variety of forms, which lie between the operating system and the distributed application as shown in Figure 4.1 and are classified as being either *synchronous* or *asynchronous*. Synchronous systems require that each middleware

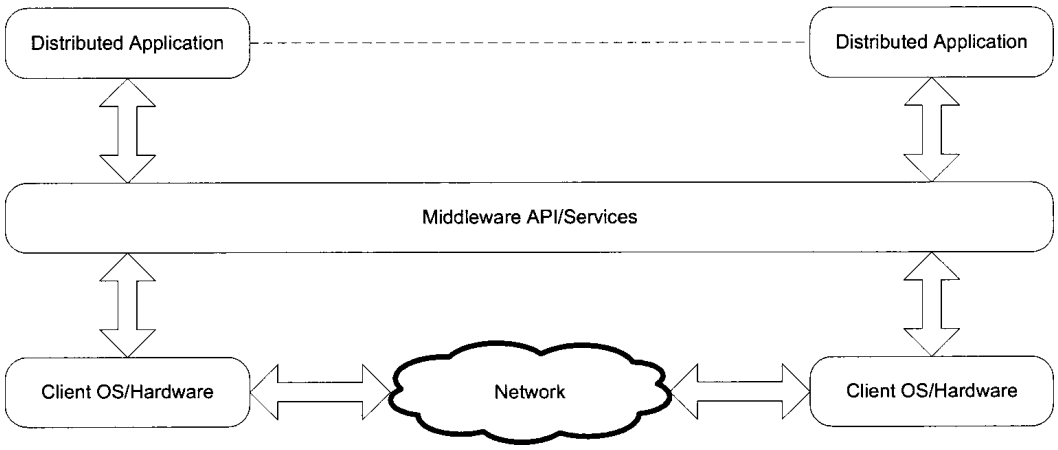


Figure 4.1: Middleware operation block diagram.

request be carried out to completion before any further requests are processed. As a result, multiple threads of execution are necessary for parallelism to occur. Conversely, asynchronous systems allow multiple requests to be issued without requiring the prior completion of any single request. However, responses are not guaranteed to be processed in order within any single thread of execution. The most commonly known types of middleware implementations are described below:

- **Publish/Subscribe** — *Publish/Subscribe* [43] is an asynchronous middleware implementation in which *publishers* of information do not explicitly transfer data to specific recipients. Instead, interested clients termed *subscribers* indicate to the publishers the types of data they want to receive. When relevant data becomes available, all interested subscribers are notified by the publishers of its availability, and each subscriber may thereafter decide to acquire the data. Publish/Subscribe middleware implementations are particularly useful in event-driven applications.

- **Remote Procedure Call** — *Remote Procedure Call* (RPC) [44], available in both synchronous and asynchronous variations, enables a program to invoke functions

implemented within another program running on a remote hardware system over a network. The details of the network transmissions required to carry out the procedure call are transparent to the system user. Asynchronous RPC is highly scalable since little information about the state of any single network transaction needs to be maintained, however synchronous RPC prevents saturation of network bandwidth and provides greater transaction integrity due to the blocking nature of synchronous requests.

- **Message-Oriented Middleware** — *Message-Oriented Middleware* (MOM) [43] is an asynchronous middleware implementation that is based on the passing of *messages* between devices on a network. Messages received by a client are stored in a *message queue* until they are able to be processed. The client may continue processing other data while incoming messages are enqueued. Like Publish/Subscribe, MOM implementations are well suited to event-driven applications. They also provide much flexibility in the implementation of when and how messages are enqueued and dequeued, and may even be designed in a manner that facilitates real-time performance.

- **Object Request Broker** — *Object Request Broker* (ORB) [45] is a synchronous middleware implementation that allows data and services within a distributed system to be abstracted to an *object*-based representation, thus allowing the system to be implemented in an object-oriented manner. Invocations on a remote object are handled by an ORB process, which tracks all the available objects in the system and handles the transmission and any necessary translation of data structures between the requesting process and the service provider. The implementation details of the translational process are completely encapsulated from both the invoker and the service provider. ORB middleware is commonly utilized within mainstream

computer networks, and popular implementations include the Object Management Group's *Common Object Request Broker Architecture* (CORBA) [46], Microsoft's *Distributed Component Object Model* (DCOM) [47], and Sun Microsystems' *Java Remote Method Invocation* (Java RMI) [48].

- **SQL-Oriented Data Access** — *SQL-Oriented Data Access* [43] is a synchronous middleware implementation that allows applications to access diverse database types over a network. Through this middleware, applications may issue generic, database-independent *SQL* (Structured Query Language) queries that are translated, if necessary, to database-specific queries. Like ORB, SQL-Oriented Data Access middleware is often utilized within mainstream computer networks, and popular implementations include Microsoft's *Open Database Connectivity* (ODBC) [49] and Sun Microsystems' *Java Database Connectivity* (JDBC) [50].

The middleware layer of the modular sensing system software architecture is based on the Message-Oriented Middleware implementation due to its support for real-time performance and the low overhead of directly transmitting messages between queues. In addition, the implementation flexibility of MOM-based middleware services facilitated the addition of synchronous message transmission to the middleware layer for the purposes of the software architecture.

4.3 Message Format

A middleware layer *message* consists of a 44-byte header, followed by a single variable-length block containing the data to be transferred in the message, as shown in Figure 4.2. The field sizes of the message format were carefully chosen so as to satisfy the memory alignment requirements of typical 32-bit modern microprocessors, particularly the ARM-based LPC2148 microcontroller utilized in the TIMs. Modules request data and services

Source Address (8)		
Destination Address (8)		
Deadline (8)		
Timestamp (8)		
Message Type (1)	Service Func. (1)	Service ID (4)
Param. Type (2)	Param. Arr. W. (2)	Param. Arr. H. (2)
Data Field (variable-width)		

Figure 4.2: Middleware layer message format (field sizes in bytes).

from other modules by issuing *service call* messages. The requested data or the results of the service call are transmitted back to the caller in the form of *return* messages, either synchronously or asynchronously as demanded by the template class algorithm running on the caller. The middleware layer message fields are described as follows:

- **Source Address** (8 bytes) — The *source address* field identifies the physical or logical module that transmitted the message.
- **Destination Address** (8 bytes) — The *destination address* field identifies the physical or logical module intended to receive the message.
- **Deadline** (8 bytes) — If the message is a service call, the *deadline* field indicates the time at or before which service call should be completed. A deadline timestamp consisting of all bits set except the most significant bit (MSB) corresponds to an effectively infinite deadline.
- **Timestamp** (8 bytes) — The *timestamp* field indicates the time at which a particular message was enqueued for transmission, or the time at which a particular event occurred. The timestamp format, which is also used within the deadline field, is a 64-bit signed integer representing the number of microseconds that have elapsed since midnight on January 1, 1.

- **Message Type** (1 byte) — The *message type* field indicates the method by which the contents of a particular message should be processed. Messages may be synchronous or asynchronous *Call At* or *Call By* service calls, or a *Return* message issued in response to a service call. Call At messages are processed at the time specified in its timestamp field, while Call By messages are processed as early as possible before the time specified in its timestamp field.

- **Service Function** (1 byte) — The *service function* field indicates what service function should be invoked if the message is a service call, or if the message is a return message, what service function was invoked.

- **Service Identifier** (4 bytes) — The *service identifier* field contains a 32-bit unsigned integer that uniquely identifies a service call message and its associated return message, and facilitates the tracking of enqueued return messages corresponding to asynchronous service calls. Each TIM internally maintains a service identifier counter that is incremented once its value is assigned to the next outgoing service call. Upon reaching its maximum value the counter overflows and resets to a value of one, since the zero value is reserved.

- **Parameter Type** (2 bytes) — The *parameter type* indicates the type of data supplied as parameters within the data field of a service call or return message. The supplied data parameters are organized into a two-dimensional array of fixed-sized elements, of which the element size is implied by the parameter type. The supported parameter types are: 8-bit, 16-bit, 32-bit, and 64-bit *signed and unsigned integers*; 32-bit single-precision and 64-bit double-precision IEEE 754 *floating-point values*; a 32-bit *status* type used to transfer various constants indicating the status of modules and service calls; a *null-terminated string* type based on arrays of 8-bit ASCII characters; a *message container* type used to encapsulate other messages

in the form of raw 8-bit bytes; and an *object container* type used to encapsulate generic data in the form of raw 8-bit bytes.

- **Parameter Array Width** (2 bytes) — The *parameter array width* indicates the width of the data field parameter array in terms of the parameter type unit size.
- **Parameter Array Height** (2 bytes) — The *parameter array height* indicates the height of the data field parameter array in terms of the parameter type unit size.
- **Data Field** (variable-width) — The *data field* stores the parameter array data to be transmitted within the message. The size in bytes of the data field of any particular message is given as the product of its parameter array width, its parameter array height, and its parameter type unit size in bytes.

4.4 Service Functions and Service Calls

Service functions enable modules to provide services to and exchange information with each other. These functions may be invoked automatically by other modules within the network, or manually through an administration module. The call/reply mechanism used during the invocation and processing of service functions, known as a *service call*, is based on the standard, widely used *Remote Procedure Call* (RPC) protocol [44]. As seen in Figure 4.3, a service call is invoked by a module through the placement of a *Call By* or *Call At* message in the outgoing message queue common to all the module structures present on a TIM. This message specifies the service function type and contains the relevant parameters to the function.

Once the message is transmitted and received by the TIM on which the target module structure is present, it is placed in the appropriate *Call By* or *Call At* incoming message queue of the module structure. The message is processed by or at the specified deadline

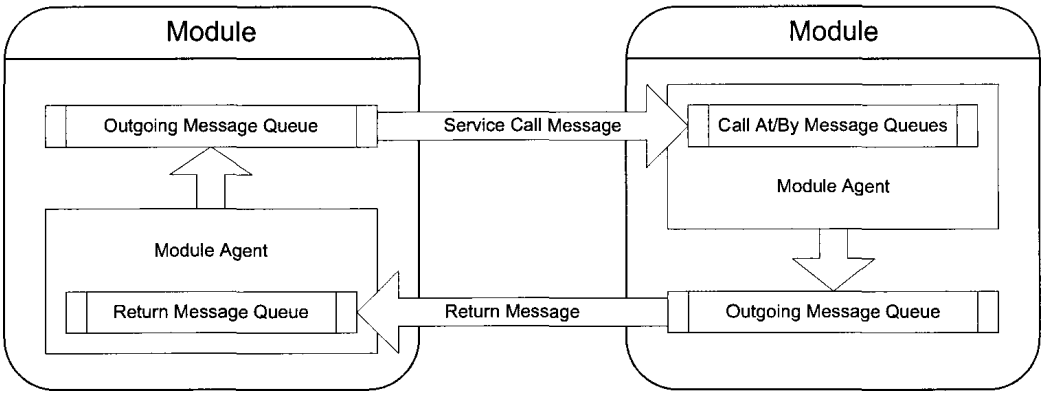


Figure 4.3: Service call operation.

respectively, and a return message containing the results of the service call is placed in the global outgoing message queue for transmission to the invoking module. This return message possesses the same *service identifier* as the call message, enabling it to be identified by the calling module as the results of the service call even if unrelated return messages are received from other modules before the call is completed. Depending on the state of the target module at the time of a service call or the success of the call itself, the return message may contain one of six *status constants*, outlined in Table 4.1, instead of the expected data.

Service calls are issued either *synchronously* or *asynchronously* to a remote module. A synchronous service call causes the real-time operating system to suspend the calling task until the corresponding return message is received from the remote module or the service call times out, while during an asynchronous service call the task is allowed to concurrently continue execution while the call is being processed. To prevent the indefinite blocking of a module message processing task due to a service call in which the return message is not forthcoming, synchronous service calls time-out within five seconds.

Table 4.1: Service function status constants.

Status Constant	Description
SUCCESS	The service call was executed successfully by the target module.
ERROR	An error occurred on the target module while executing the service call.
MISSED DEADLINE	The service call was not executed by the target module before or at the specified deadline.
INVALID PARAMETER	The service call was issued to the target module with one or more invalid parameters.
LOCKED	The target module of the service call is locked by a module other than the caller.
NOT ALLOWED	The service call is not allowed to be issued to the target module.

4.4.1 Service Function Types

The various service functions defined by the software architecture that enable the state and properties of a module to be obtained or modified are as follows:

- **Get** — The *Get* service function is used to obtain the value or state of the transducer associated with the target TIM. The type of the parameter returned to the invoker is dependent on the type of transducer present on the target TIM. If the transducer does not allow its value or state to be obtained, the NOT ALLOWED status constant is returned.

- **Set** — The *Set* service function is used to modify the state of the transducer associated with the target TIM. The type of the parameter supplied in the service call message and the behaviour of the call itself is dependent on the type of transducer present on the target. Depending on the actuation capabilities of the transducer, a *Set* service call may also result in an update of the pose matrix of the TIM with which the transducer is associated. Various status constants may be returned by *Set* service calls. If the TIM is locked by another module, LOCKED is returned. If the

supplied parameter is invalid, `INVALID PARAMETER` is returned. If the service call fails, `ERROR` is returned. If the transducer does not allow its state to be modified, the `NOT ALLOWED` status constant is returned. If the call completed successfully, `SUCCESS` is returned.

- **Append** — The *Append* service function is identical to the *Set* service function with the exception that it is used to add to the state of the transducer associated with the target TIM instead of changing it directly.

- **Reset** — The *Reset* service function, like the *Append* service function, is identical to the *Set* service function with the exception that it is used to reset the state of the transducer associated with the target TIM to its default value instead of setting it to an arbitrary value.

- **Get TEDS** — The *Get TEDS* service function is used to retrieve from the TEDS property list of the target module the TEDS entry value associated with the TEDS entry name provided as a string parameter to the service call. If the provided parameter is not a string, `INVALID PARAMETER` is returned. If the parameter is a string and an associated TEDS entry value is not found, `ERROR` is returned.

- **Get Pose** — The *Get Pose* service function is used to obtain a copy of the local 4×4 matrix of single-precision floating-point values representing the pose (position and orientation) of a TIM.

- **Update Pose** — The *Update Pose* service function is used to force a TIM to update its own position and orientation matrix with respect to the *pose matrix* (see Section 6.5) provided within the pose update structure supplied as a parameter. This service function, which returns no status constant, is used internally by the software architecture to update the poses of a tree of physically connected TIMs

and is not intended to be called within template algorithms.

- **Lock** — The *Lock* service function is used to prevent *Set*, *Append*, or *Reset* calls from being performed on a TIM by any module other than that which issued the lock. Locking a TIM is useful when multiple modification operations issued by another module need to be performed atomically. A *Lock* service call only returns `SUCCESS` to the caller if the target is not already locked, otherwise `LOCKED` is returned.

- **Unlock** — The *Unlock* service function is used to unlocked a previously locked module. An *Unlock* service call only returns `SUCCESS` if issued by the module that holds the lock or if the target is already unlocked, otherwise `LOCKED` is returned.

- **Join** — The *Join* service function is used to incorporate a TIM into a new or existing logical entity. This service function, which returns no status constant, is used internally by the software architecture for logical module composition purposes and is not intended to be called within template algorithms. The target module only processes a *Join* call from a logical entity if it is not already a member of the entity. If it is not a member, a new module structure is created locally on the target module to represent the state of the logical entity. A new membership structure entry is then created and added to the membership list of the newly joined target module, indicating the role it fulfils within the logical entity as well as the address of any physical dependency.

4.5 Module Message Handler Task

Each combination of a *module structure* (see Section 2.7.3) and a *module message handler task* present on a TIM is termed a *module agent*. The module structure represents the

state of the module agent, while the *module message handler task* represents its *behaviour*. The module message handler task continuously examines and updates the queues and status fields of its associated module structure and generates messages in response to received messages and other events that occur within its environment. This intelligence is provided by either a native message handling function, associated with self-contained modules representing a single transducer on a TIM, or by platform-independent template classes (see Section 2.5.2), associated with logical module entities and used to facilitate collaboration between its member modules. Through the use of a real-time operating system within the software architecture, multiple module message handler tasks may execute concurrently on a single TIM, each receiving and transmitting messages in real time.

The general operation of a module message handler task is depicted in Figure 4.4. Similar to the network communication task discussed in Section 3.6, each module message handler task contains at its core an infinite loop in which a number of operations are carried out at differing time intervals. In the message handler task of logical module agents the message loop is invoked within the virtual machine (see Chapter 5), which facilitates the execution of platform-independent logical module template classes. A self-contained module agent representing a transducer on its local TIM hardware instead utilizes a built-in native message loop within its message handler task to reduce the overhead incurred by the interpretive mechanism employed by the virtual machine. The various operations carried out within each iteration of a native or platform-independent module message handler task loop are as follows:

- Obtain the next service call to be processed, if any, from the *Call-At* and *Call-By* message queues. Call-At and Call-By messages are inserted into their respective queues by deadline, but since Call-At messages need to be checked more regularly for a deadline match, using separate queues ensures that the next Call-At message

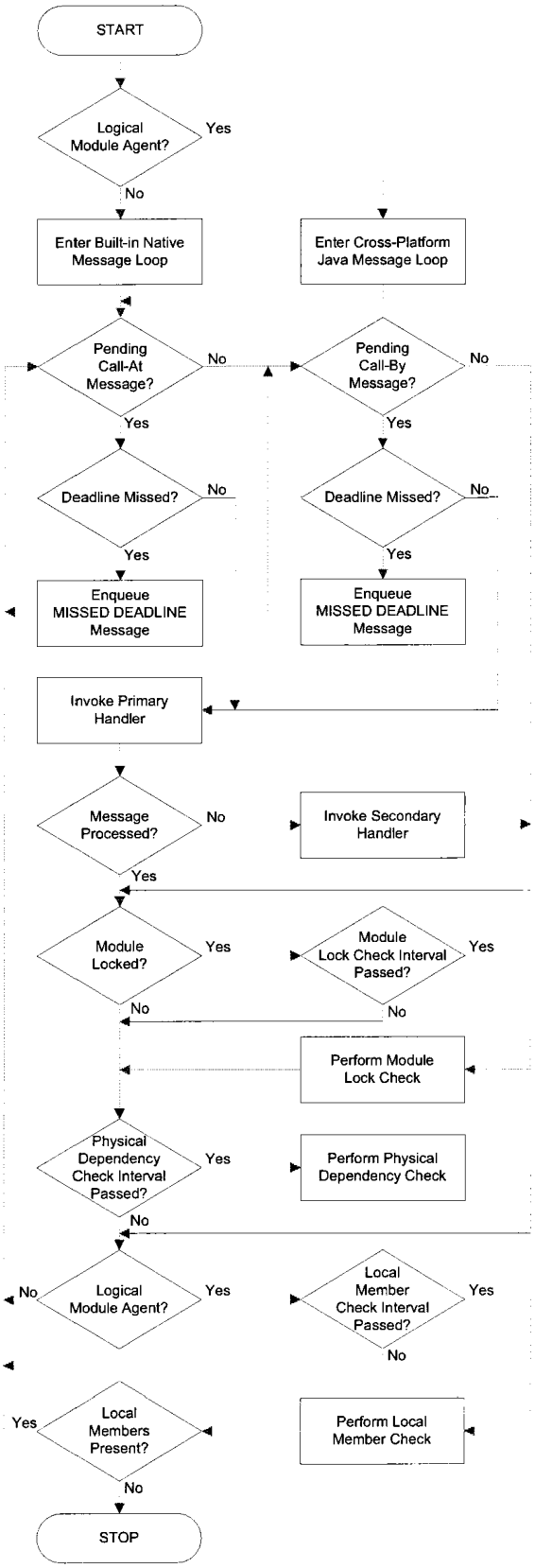


Figure 4.4: Module message handler task operation.

is always accessible within a single operation. This might not always be possible if a single queue was used for both message types, in the scenario that numerous Call-By messages are ahead of the next Call-At message in the queue. The message acquisition process is further outlined in Section 4.5.1.

- Process the service call by invoking the *primary handler*, and if necessary, the *secondary handler*. For a module agent associated with a transducer on the TIM hardware on which it executes, the primary handler is the relevant driver function for the transducer. For a module agent that represents a logical module, the primary handler is its associated cross-platform template algorithm. The primary handler is expected to provide implementations for processing at least the *Get*, *Set*, *Append*, and *Reset* service calls. The secondary handler is invoked to process the service call if the primary handler provides no suitable implementation for handling it, and provides default implementations for processing the *Get TEDS*, *Get Pose*, *Update Pose*, *Lock*, *Unlock*, and *Join* service calls. If the service call is still not handled after invocation of the secondary handler, it is deemed to possess an invalid service function type and is deallocated. The general execution processes of primary and secondary handlers are depicted algorithmically in Algorithms 4.1 and 4.2 respectively.

- Perform various standard *status checks* that ensure the integrity of the module agent is maintained throughout changes to the environment in which it executes. These status checks are outlined in Section 4.5.2.

4.5.1 Message Acquisition

The next service call to be processed is obtained through a continuous search of the *Call-At* and *Call-By* incoming message queues present in each module structure. These queues

Algorithm 4.1 Primary handler execution process.

```
procedure PRIMARYHANDLER(module module, message call)
    if call.servicefunction = Get then
        return ⇐ module.get(call)
    else if call.servicefunction = Set then
        return ⇐ module.set(call)
    else if call.servicefunction = Append then
        return ⇐ module.append(call)
    else if call.servicefunction = Reset then
        return ⇐ module.reset(call)
    end if
    if call was handled then
        deallocate call
        if return was generated then
            enqueue return in outgoing message queue
        end if
    end if
end procedure
```

Algorithm 4.2 Secondary handler execution process.

```
procedure SECONDARYHANDLER(module module, message call)
    if call.servicefunction = GetTEDS then
        return ⇐ module.getTeds(call)
    else if call.servicefunction = GetPose then
        return ⇐ module.getPose(call)
    else if call.servicefunction = UpdatePose then
        return ⇐ module.updatePose(call)
    else if call.servicefunction = Lock then
        return ⇐ module.lock(call)
    else if call.servicefunction = Unlock then
        return ⇐ module.unlock(call)
    else if call.servicefunction = Join then
        return ⇐ module.join(call)
    end if
    deallocate call
    if return was generated then
        enqueue return in outgoing message queue
    end if
end procedure
```

are *priority queues*, and upon reception, messages are inserted into the appropriate queue in order of deadline. Since the queues are searched by deadline, the number of operations needed to search the queues in each iteration of the message loop is reduced from the order of $O(n)$ to $O(1)$ operations.

Call-At messages are given precedence over Call-By messages since Call-By messages may be invoked at any time before their deadline, while Call-At messages need to be continuously checked with as little delay as possible to determine exactly when their deadline timestamp has been matched. Thus, the Call-At message queue is always searched first. The message at the front of the queue (which will always have the closest deadline), if any, is examined, and the time difference between the message deadline and the local TIM system clock is determined. If this time difference is within the one millisecond threshold that indicates message validity, the message is dequeued and returned to the message handler. If the time difference is not within the threshold and is less than the current system time, the message is dequeued and deallocated. Subsequently, a message containing the status constant `MISSED DEADLINE` is enqueued within the outgoing message queue, after which the Call-At message queue is searched again. The method in which a missed deadline is handled is delegated to the calling module which, depending on its current state, may choose to reissue an identical call, issue a different call, or drop the call altogether.

If no suitable Call-At message is obtained, the Call-By message queue is searched. The message at the front of the queue, if any, is immediately dequeued. The message is returned to the message handler for immediate processing unless the message deadline is less than the current system time, in which case it is deallocated and a message containing the status constant `MISSED DEADLINE` enqueued within the outgoing message queue. As with Call-At messages, the method in which a missed deadline is handled is delegated to the calling module, whose state may have changed since the call was issued. The Call-By

message queue is then searched again.

4.5.2 Standard Status Checks

Module Lock Check

The *module lock check*, performed every fifteen seconds, is used to determine if the module agent should be automatically unlocked due on the absence of the locking module in the global environment list (see Section 3.5). This check prevents the module agent from being locked indefinitely if the locking module does not reappear within the environment. If the module agent is locked, the environment list is searched. If a presence packet corresponding to address of the locking module is not found in the environment list, the agent is unlocked by setting the locking address in its corresponding module structure to the reserved *zero address*.

Physical Dependency Check

The *physical dependency check*, performed every fifteen seconds only if the module agent is a member of a physically dependent logical module entity, is used to determine if the module agent should automatically withdraw from the logical entity due to the lack of a physical connection between the local TIM on which the module agent executes and a remote TIM that is also a member of the logical entity. This check ensures that a logical entity whose correct behaviour depends on physical connections between its member modules does not try to utilize member modules that have lost physical connectivity to others within the logical entity. In this check, each membership structure entry (if any) in the membership list associated with the module agent is examined. If any membership structure has a non-zero *physical dependency address* (indicating that membership in the logical entity depends on a physical connection to the TIM possessing the specified address), the environment list is searched to locate a presence packet corresponding that

address. If an associated presence packet is not found, or the packet does not indicate that a physical connection between the respective TIMs is present, the membership structure is removed from the membership list. Removal of the membership structure indicates that the module agent is no longer a member of the collaborating group of modules that comprise the logical entity.

Local Member Check

The *local member check*, performed every fifteen seconds only within logical module agents, is used to determine if the message handler task of a particular logical module agent should be automatically terminated due to a lack of member module agents located on the same TIM hardware on which it executes. Each logical module agent must have at least one of its member module agents executing on the same TIM hardware in order to guarantee that at least one member module is available that is capable of providing sensing or actuation functionality to the logical entity. To determine if this requirement is satisfied, the membership lists of all the other module agents executing on the TIM are searched. If no other locally executing module agent possesses a membership structure entry corresponding to the logical module agent, its associated module message handler task is terminated, and its module structure is garbage collected.

4.6 Summary

In this chapter, the *middleware layer* of the software architecture was described. The middleware layer provides the commands and services through which *module agents* may interact and communicate with each other. To exchange data, variable-length *messages* are transferred between module agents. These messages are also used to invoke the services provided by module agents in the form of *service calls* to *services functions*.

Each module agent is represented by a *module structure* used to maintain its state as well as a *module message handler task* that processes incoming messages and generates outgoing messages as specified by an associated native or platform-independent message processing algorithm. The following chapter will describe the operation of the *virtual machine.*

Chapter 5

Virtual Machine

5.1 Introduction

A *virtual machine* (VM) is a program which interprets and executes high-level, hardware-independent abstract *bytecodes*. Each bytecode is a sequence of one or more bytes that represents an instruction to be executed by the VM. Algorithms defined using these bytecodes are therefore completely decoupled from the underlying hardware architecture on which they execute. A lightweight Java-based VM supported by an architecture-specific standard class library was implemented within the software architecture stack, enabling the *logical module template algorithms* that define the behaviour of the collaborating TIMs comprising a logical module entity to be specified once and then used, without recompilation, in the dynamic reprogramming of a variety of heterogeneous modules and hardware architectures as application requirements change.

5.1.1 Choice of Dynamic Reprogramming Mechanism

To facilitate the adaptability and reconfigurability of a group of physically or wirelessly collaborating heterogeneous TIMs, a dynamic reprogramming mechanism is necessary

that allows the TIMs to automatically source, load, and execute logical module composition algorithms at run-time without disrupting the operation of other algorithms executing on any particular TIM. In addition, the reprogramming mechanism should utilize a minimum of system resources during the loading of the algorithms as well as during their execution. The composition algorithms themselves should be implementable in a hardware-independent format such that they are easy to create, debug, and maintain by the system user, and should also expose the behaviour of the group of collaborating modules rather than aspects of the underlying hardware architectures of the modules themselves. Various dynamic reprogramming mechanisms as referred to in [51] were considered for use in the software architecture, and are described below:

- **Monolithic Binary Update** — In *monolithic binary update* dynamic reprogramming mechanisms, the operating system, software architecture stack, as well as the various executing algorithms comprise a single binary image. Therefore this reprogramming mechanism, although providing the maximum flexibility in terms of behavioural modification, is very expensive in practice due to the need for the entire binary image to be overwritten in non-volatile memory with each change in application requirements. In addition, the operation of any executing algorithms must be suspended during the overwriting process, thus introducing further latency into the system. Well known implementations of monolithic binary update systems include *Crossbow Network Programming* (XNP) [52], *Deluge* [53], *Multihop Network Reprogramming Protocol* (MNP) [54], and *Multihop Over-the-Air Programming* (MOAP) [55], all of which target network reprogramming of wireless sensor network platforms based on the Crossbow *MICA2* [10] motes.

- **Modular Binary Update** — In *modular binary update* dynamic reprogramming mechanisms, the various executing algorithms are decoupled from the operating system and software architecture stack into binary modules that are dynamically

linked into the execution environment at runtime. Although slightly less flexible as compared to monolithic binary update mechanisms, programming latency is greatly reduced due to the much reduced size of the linked binary modules and the now unnecessary need to reprogram software components such as the OS and architecture stack, which are often static. The smaller module size also greatly reduces network transmission latency of the binary module to multiple nodes, while also reducing network transmission power consumption on resource-constrained nodes. However, the binary modules are specified using the native instruction set of the microprocessor of the node on which it is intended to execute, and thus remain hardware dependent. Well known implementations of modular binary update systems include *Contiki* [56] and *SOS* [57].

- **Virtual Machine** — In *virtual machine* dynamic reprogramming mechanisms, algorithms specified in the form of architecture-independent abstract *bytecodes* are executed through the use an interpretive loop. Each bytecode is a numeric constant that corresponds to a single machine instruction, some of which may require other constants and data to be supplied as parameters. Unlike the *opcodes* (operation codes) that comprise the native instruction sets of physical microprocessors, the bytecodes that comprise virtual machine instruction sets are designed to be portable and are easily translated to the instruction sets of most microprocessors. Since the algorithms are completely decoupled from the operating system as well as the underlying hardware on which they execute, the behaviour of a node may be easily modified without the necessity for its operation to be suspended, and a single algorithm specification may be used to dynamically reprogram a variety of heterogeneous modules. Virtual machine instruction sets, and by extension the algorithms specified using them, tend to be compact and thus also facilitate reduced network transmission power consumption on resource-constrained nodes. This is especially

true for *stack-based* virtual machine implementations in which data is manipulated on a common stack, as opposed to *register-based* virtual machine implementations in which data is manipulated using discrete registers [58]. A disadvantage of virtual machine dynamic reprogramming mechanisms is the moderate execution overhead incurred due to the need for an continuously executing interpretive loop. However, this overhead is negligible on sufficiently fast hardware, such as that utilized in this software architecture. Popular virtual machines for high-end computer systems and embedded platforms include Sun Microsystems' *Java Virtual Machine* [23] and Microsoft's *Common Language Runtime* (CLR) [59]. Well-known implementations of virtual machine dynamic reprogramming mechanisms for resourced-constrained embedded platforms include *Maté* [60] and its successor *Application Specific Virtual Machines* (ASVM) [61], *Scylla* [62], and *VM** [63].

- **Query/Parameter-Based Configuration** — In *query/parameter-based configuration* dynamic reprogramming mechanisms, the behaviour of communicating nodes is modified though the issuing of various queries and parameters associated with the desired behavioural change. Query/parameter-based configuration frameworks are the most inflexible in terms of the extent to which the operation of a given sensing system may be defined, since the range of behaviours supported by the sensing system is limited to the possible combinations of the queries and parameters understood by the nodes in the system. However, because there is no need to transmit an entire native or cross-platform algorithm specification, the latency associated with behavioural modification is extremely low, and the reprogramming mechanism is conducive to even extremely resource-constrained platforms. A well-known implementation of a query/parameter-based configuration dynamic reprogramming mechanism is *TinyDB* [64], which provides an SQL-based query interface to MICA2-based platforms.

A stack-based virtual machine dynamic reprogramming mechanism is utilized within this software architecture and is based on Sun Microsystems' *Java Virtual Machine* [23] (JVM). The use of a Java-based virtual machine provides the software architecture with a powerful and well-established platform in which hardware-independent algorithms may be specified. Due to the widespread adoption of the JVM, there is great flexibility in the choice of algorithm specification language. Other than the standard compiler for the Java programming language itself, compilers are available for other scripting and programming languages such as *Groovy* [65] and *Jython* [66] (a Python implementation) that directly generate JVM classes from their source files. A subset of the JVM instruction set is utilized within the virtual machine of this software architecture and includes bytecode instructions for loading and storing data, performing arithmetic and logic operations, converting between numeric types, creating and manipulating class instances, manipulating frame operand stacks, branching, and performing method invocations and returns.

The previously mentioned VM* dynamic reprogramming architecture is also based on the JVM. In VM*, Java classes are not executed directly; rather, the classes are preprocessed by a base station and used to synthesize an interpreter that is suitable for execution on resourced-constrained MICA2 motes. Unlike VM*, the virtual machine implemented for use within this software architecture requires no external preprocessing of classes, and instead loads and executes the classes on the TIMs directly. However, due to flash memory and RAM constraints, the extensive standard class library provided within a full Java implementation is not present in its entirety. Provided instead is a very lightweight and useful subset of the standard class library as well as architecture-specific classes that encapsulate the functionality necessary to support the collaboration of a group of TIMs.

5.2 Class Loading

As previously mentioned in Section 2.5.2, the platform-independent *logical module template classes* that enable connected TIMs to collaborate with each other are found in the *template class directory* `amss/algo`. When a class is to be loaded from this directory, a global vector termed the *class list* in which loaded classes are placed is first searched to determine if the specified class is already loaded. By searching the class list first, memory is conserved through the prevention of loading duplicates of a particular class. The complete Java class file format specification is as described in [67]. The main components of interest are described below, and are loaded and processed from the class file in the order shown.

- **Constant Pool** — The *constant pool* of a class is a table of structures that contains all the unchanging values referred to by the bytecode instructions of the class. These constants include class names, field and method references, 32-bit and 64-bit integers, single-precision and double-precision IEEE 754 floating-point values, and UTF-8 (8-bit Unicode Transformation Format) strings.

- **Field Information Table** — The *field information table* of a class is a table of structures that describes the *fields* present within the class. A field is any *class variable*, which are static and common to all class instances, or *instance variable*, which are non-static and unique to each class instance.

- **Method Information Table** — The *method information table* of a class is a table of structures that represent the *methods* present within the class. These structures contain among other properties the name, maximum stack size and total local variable size of the method, as well as the actual bytecode in which the method is implemented.

Once the class is loaded successfully, it is added to the class list. Its constant pool is then searched for field and method references external to the class. If any external references are found, they are resolved by recursively loading the class which possesses the referenced field or method if it is not already within the class list. However, if the referenced class is within the architecture-specific *standard class library* (described in Section 5.4) that supports the operation of the software architecture, a *stub class* is instead created to resolve the reference. This is necessary because the standard library classes, which are unchanging, are actually implemented using the native machine language of the TIM microcontroller, instead of platform-independent Java bytecode. The standard library classes are implemented in this manner in order to minimize flash memory and RAM usage. In addition, the speed of calls to methods within these classes, which occur very frequently and occupy much of the processing time, is vastly increased due to the removal of interpretive overhead.

After all external field and method references are resolved, the method information table of the loaded class is searched to determine if the special *class initialization method* `<clinit>` is provided. This function, generated only by the Java compiler if necessary, is immediately executed by the virtual machine before beginning or resuming the execution of any other method in order to initialize the static fields of the class before it is utilized.

5.3 Class Execution

Upon the creation of a logical module agent message task, an appropriate class from the template class directory is loaded to provide the intelligence necessary for the associated TIM to function as a member of the logical entity, as described in Section 4.5. The standard entry point for this class is the method `main`, as is normal for Java classes that are intended to be executed. However, unlike typical executable Java classes that accept a

`String` array as a parameter to the `main` method, logical module template classes accept a `Module` class reference specific to the software architecture. This reference provides an interface through which the contents of the module structure associated with the logical module agent may be accessed from within the template class.

Execution of a method in the Java virtual machine, and by extension the virtual machine used in this software architecture, is dependent on three main types of *runtime data areas* that are described below:

- **Program Counter** — The *program counter* register is an integer variable unique to each thread of execution that always contains the *address* of the next bytecode instruction to be executed. After the execution of each bytecode, the program counter is updated to point to the next bytecode instruction.

- **Runtime Stack** — The *runtime stack* is a LIFO (last in, first out) data structure composed of *frames*. Each frame is a dynamically allocated node structure in a doubly linked list that collectively comprises the runtime stack. Frames are created whenever a new method is invoked, and are used to store data such as local variables and also provide an operand stack for storing partial calculations. Frames also facilitate method invocations and recursion.

- **Runtime Heap** — The *runtime heap* is a region of memory in which class instances and arrays are stored. It is common to all instantiations of the virtual machine and comprises a portion of the total heap memory utilized for dynamic memory allocation throughout the layers of the software architecture.

The general execution process of a method, carried out within the *virtual machine execution function*, is depicted algorithmically in Algorithm 5.1. Firstly, the program counter is reset to zero, where the first bytecode instruction of the method is always located. A new local frame is then allocated and pushed onto the runtime stack to

Algorithm 5.1 Method execution process.

```
procedure EXECUTEMETHOD(method method, frame invoker)
    pc ⇐ 0
    frame ⇐ allocate local frame
    for i ⇐ parameters in invoker.stack−1 to 0 do
        frame.locals[i] ⇐ invoker.stack.pop()
    end for
    if method is native then
        method.nativecode(frame)
    else
        loop
            instruction ⇐ method.bytecode[pc]
            pc ⇐ pc + 1
            if instruction is an invocation then
                index ⇐ method.bytecode[pc, pc + 1]
                invoked ⇐ method in constant pool at index
                pc ⇐ pc + 2
                EXECUTEMETHOD(invoked, frame)
            else if instruction is a return then
                if return value is present then
                    invoker.stack.push(frame.stack.pop())
                end if
                break loop
            else
                execute instruction
            end if
        end loop
    end if
    deallocate frame
end procedure
```

support the execution of the method. Any parameters passed from a calling method are acquired from the calling frame and stored in the local frame as local variables. If the method is determined to be a *native method*, bytecode interpretation is skipped since the method is comprised of native machine code instructions intended to be executed directly on the microprocessor hardware without interpretation. After execution of the native method, the local frame is then popped from the runtime stack and deallocated, and the virtual machine execution function returns. If the method is not a native method, the *bytecode interpretation loop* is entered.

The bytecode interpretation loop is generally an infinite loop containing a switching mechanism, and implements a *fetch, decode, execute* cycle similar to that found in modern microprocessors. In each iteration of the loop, the next 8-bit bytecode instruction to be executed is fetched from the location indicated by the program counter, and its value used to select and execute the section of the switching mechanism corresponding the actions associated with the instruction. If the instruction is a method invocation instruction, any values to be passed to the invoked method will have been pushed onto the local frame by the preceding instructions. The reference to the method to be executed is obtained from the constant pool at the 16-bit index provided within the invocation instruction. The virtual machine execution function is then recursively called to execute the invoked method. If the instruction is a return instruction, any value to be returned to the calling method is stored in the calling frame. The local frame is then popped from the runtime stack and deallocated. The bytecode interpretation loop is then broken and the virtual machine execution function returns.

5.4 Standard Class Library

The software architecture provides a standard collection of classes and methods, grouped into *packages*, designed specifically for use within template classes. As described in Section 5.2, these classes are not physically present on the local storage of the modules; rather, references to methods in these classes due to invocations are caught and handled at runtime in order to minimize flash memory and RAM usage and also substantially increase performance. The classes in the packages comprising the standard class library of the software architecture, and their purpose, are described in the following subsections.

5.4.1 The java.lang Package

The official Java class library provided within distributions of the Java Virtual Machine is comprised of numerous packages and classes which provide a base for the development of Java applications. One of the most important of these packages is `java.lang`, which provides fundamental classes and methods facilitating features such as mathematics and string processing. A subset of the complete `java.lang` package, the comprising classes of which are depicted in Section A.2, was included within the standard class library of the software architecture to provide these two features, which are considered critical to its operation. These classes are outlined below:

- **Math** — The class `java.lang.Math` contains methods that facilitate the calculation of trigonometric, hyperbolic, logarithmic, exponential, randomization, rounding, minimum and maximum, and absolute value operations. It also defines the mathematical constants e and π.

- **String** — The class `java.lang.String` is used to represent an immutable string of characters, and contains methods that facilitate a variety of commonly used string

comparisons and operations such as determination of string length, tests for string equality, substring generation, and case conversion.

5.4.2 The amss.system Package

In addition to the `java.lang` package, a package of classes unique to the software architecture are provided to support and provide an interface to its core operations. This package is `amss.system`, the comprising classes of which are depicted in Section A.3, and consists of five classes that provide access to various internal software architecture functions that facilitate operations such as manipulating messages, modules, pose matrices as well as vectors. These classes are outlined below:

- **AMSS** — The class `amss.system.AMSS` is a collection of static methods and constants that support various system-level operations provided by the software architecture. It is analogous to the class `java.lang.System` in the official Java class library. Operations to perform explicit garbage collection (since automatic garbage is unsupported within the lightweight virtual machine), atomic operations, obtaining the local system time, and setting the current task to sleep are provided, along with commonly utilized time constants. A variety of data input and printing functions are also provided mainly for debugging purposes.

- **Message** — The class `amss.system.Message` contains methods that facilitate performing operations on and obtaining data from messages. Methods are provided for creating and enqueuing new messages, obtaining information about a particular message such as its source and destination, and retrieving and setting elements within its parameter array. Also provided are static constants that represent the various types of messages, service functions, parameters, and status values utilized within the software architecture.

- **Module** — The class `amss.system.Module` contains methods that facilitate performing operations on and obtaining information about modules. Methods are provided for obtaining the *pose matrix* (see Section 6.5) of a particular module, retrieving and modifying its TEDS properties, and querying the state of the module, such as if it is running or is located on the primary member module hardware. Role-specific methods for logical modules are also provided, such as methods for determining the number of roles, determining the number of matched modules in the environment for a particular role, and issuing service calls to and retrieving results from matched members.

- **Pose** — The class `amss.system.Pose` contains methods that facilitate the acquisition of five *pose vectors* that are derived from the matrices representing the poses of one or a pair of modules. These pose vectors are utilized within *template algorithms* (see Chapter 6) to facilitate the simple determination of the relative positions between any two indirectly or directly connected modules in a logical entity, irrespective of their orientations. The first vector represents the position of a module, and the other four vectors are depicted in Figure 5.1. The *face normal*, which initially points along the positive y-axis, is perpendicular to the top face of the module and indicates its upright orientation; the *face north* vector, which initially points along the negative z-axis, and *face east* vector, which initially points along the positive x-axis, both lie within the plane of the top face of the module and represent their respective cardinal directions relative to this plane; and the *module separation* vector indicates the direction and magnitude of the displacement between the centres of a pair of modules.

- **Vector3D** — The class `amss.system.Vector3D` contains methods that facilitate operations on three-dimensional vector quantities. These operations include obtain-

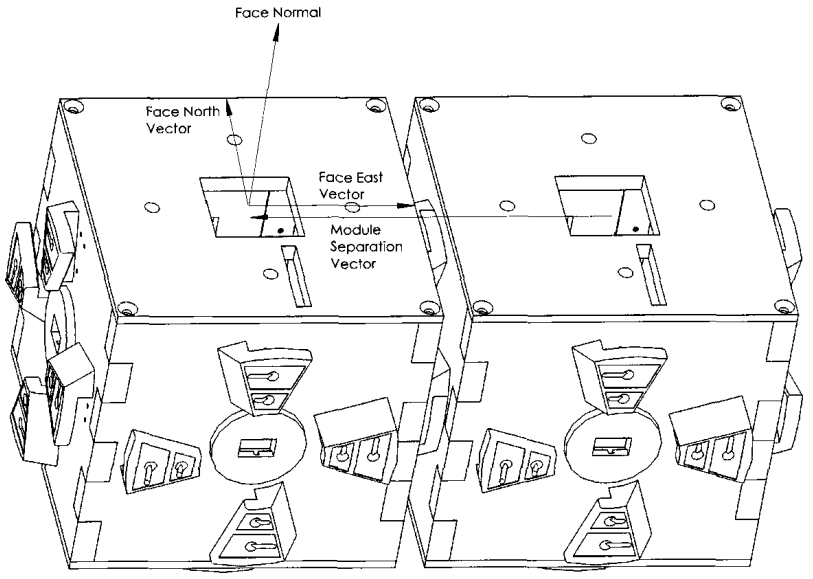


Figure 5.1: Module pose vectors.

ing the constituent components and magnitude of a particular vector, normalization of the vector, common operations between two vectors such as addition and scalar product, as well as methods to determine the relationship between two vectors, such as the angle and parallelism between them.

5.5 Summary

In this chapter, the *virtual machine* of the software architecture as well as the *standard class library* that is specific to the software architecture and supports its operation were described. The Java-based virtual machine facilitates the execution of the platform-independent *template classes* that describe the behaviour of the collaborating TIMs in a logical module entity. The platform-independence of the template classes enables them to be created once and then used in the dynamic reprogramming of a variety of TIMs irrespective of their underlying hardware architecture, the capabilities of which are ab-

stracted through the various classes and methods provided within the standard class library of the software architecture. The following chapter will describe the operation of the *composition layer*.

Chapter 6

Composition Layer

6.1 Introduction

At the composition layer, *logical module template algorithms* are loaded and executed that enable a group of TIMs to collaborate and behave as a logical entity known as a *logical module*. The intelligence needed to facilitate module collaboration is encompassed within a Java class that is interpreted by the virtual machine, and is accompanied by a *logical module template TEDS* that describes the standard characteristics of the composite logical module entity. Within each template TEDS, various *roles* are also defined.

Each module agent within the sensing system environment continually tests the others in its environment against the roles defined within its locally stored logical module template TEDS, as well as its currently loaded logical module structures, in order to locate a match. If a module agent is found that is capable of providing the sensing or actuation behaviour outlined by at least one of the specified roles, the matched agent will be assimilated into the existing or a newly created logical entity containing the matched role. Physically connected TIMs within a logical entity will also intelligently relay their position and orientation to each other, ensuring that all the member TIMs comprising

the logical entity possess a representation of their position and orientation that is relative to the pose of one of the members, designated the *pose base*.

6.2 Template Data Types

Logical module template TEDS specifications are similar in structure to standard module TEDS specifications. The two TEDS specification types differ in that template TEDS support the specification of a set of *role descriptors* included after the list of standard TEDS properties. Sample module TEDS and template TEDS specifications may be seen and compared in Sections B.2 and B.3 respectively. Upon loading a required template TEDS file, it is parsed, stored, and utilized through the data structures described in the following subsections.

6.2.1 Template Structure

The *template structure* is a representation of a locally stored template TEDS specification from which a logical module may be created, and is a transformation of the text-based template TEDS specification into a structural form usable by the software architecture. The fields contained within a template structure are defined as follows:

- **File Name** — Stores the name of the template TEDS used to construct the template structure, and is used for identification purposes.
- **Class Name** — Stores the name of the Java class that provides the intelligence necessary for the members of the composite entity to communicate and collaborate, and is also used for identification purposes.
- **Roles List** — A vector used to store the *role structures* (described in Section 6.2.2) specific to the template, as found in the template TEDS specification.

- **Template Class** — A structure used to store the actual Java class bytecodes that implement the platform-independent intelligence facilitating collaboration between module agents in a logical entity. This class is found at the path provided by the `RoleTemplateClass` property of the template TEDS. The main components of the template class structure are described in Section 5.2.

6.2.2 Role Structure

A *role structure* represents the characteristics required of a particular module agent within the environment to satisfy a particular behavioural *role* within a logical module entity. A template TEDS specification contains one or more *role descriptors* that are used to create corresponding role structures within an encompassing template structure. One or more module agents may be assigned to each role. Like the template structure, each role structure is a transformation of a corresponding role descriptor contained within a text-based template TEDS specification into a form usable by the software architecture. The fields contained within a role structure are defined as follows:

- **Role Environment List** — A list of *member* (MEM) packets corresponding to the module agents currently detected within the environment that are assigned to the role.

- **Role Number** — An unsigned integer assigned to the role that is used to uniquely identify it.

- **Assignment Limit** — Specifies a numerical range indicating the number of modules that may be utilized to satisfy the role. The number of modules may be less than, less than or equal to, equal to, greater than or equal to, or greater than the provided limit.

- **Connection Type** — A bit field indicating the types of connections allowed between current member modules of the logical entity and candidate non-member modules that satisfy the role. The connection type may be *local*, which corresponds to candidate module agents being present on the same TIM hardware as a member of the composite entity; *physical*, which corresponds to candidate module agents being present on TIM hardware that is physically connected to that of a member of the composite entity; *wireless*, which corresponds to candidate module agents being present on TIM hardware that is not physically connected to but within the environment of that of a member of the composite entity; or any combination of the three.

- **Module Type** — A bit field indicating the types of modules that satisfy the role. The module type may be a *sensor* module, an *actuator* module, an *interconnect* module, an *administrator* module, or any combination of the four.

- **Module Class** — A bit field indicating the *classes* of modules that satisfy the role. As previously described in Section 2.6, a *class* refers to a family of sensors or actuators that may be used to sense a particular physical quantity or facilitate a specific type of motion respectively. Currently, the classes include any combination of an *acceleration* module, a *positional* module, a *rotational* module, a *status* module, a *text display* module, or a *voltage* module. Unused bits are left within the bit field for future expansion.

- **Module Data Type** — A bit field indicating the acceptable data types for the array of values returned by modules that satisfy the role. The return type may be any combination of an 8-bit, 16-bit, 32-bit, or 64-bit signed or unsigned *integer*, a 32-bit or 64-bit *floating point* value, a *status string*, an encompassed middleware layer *message*, or a generic *object* consisting of raw bytes.

- **Module Data Type Width** — Specifies a numerical range indicating the number of acceptable columns in the array of values returned by modules that satisfy the role. The width of the data array may be less than, less than or equal to, equal to, greater than or equal to, or greater than the provided number of columns.

- **Module Data Type Height** — Specifies a numerical range indicating the number of acceptable rows in the array of values returned by modules that satisfy the role. The height of the data array may be less than, less than or equal to, equal to, greater than or equal to, or greater than the provided number of rows.

6.2.3 Local and Remote Join Structures

A *join structure* is issued by members of a logical module entity to candidate module agents that they have discovered within their environment, providing information to the candidate module agents indicating the actual logical module template it should load from its local template TEDS directory, as well as the matched role it should perform. *Remote* join structures differ from *local* join structures in that they are used to transmit the full template TEDS specification text of the logical module itself along with its associated template algorithm class to a matched candidate module, which are then stored on the candidate module. This ensures that the library of template TEDS specifications and processing algorithms available locally on each module within any particular sensing system is updated in an automatic, peer-to-peer fashion, without requiring user intervention. Local join structures only specify the filename of the template TEDS specification to be used, and conserve bandwidth when the specification is known to be available locally on the candidate module. The fields contained within a join structure are as described below, where all values except the *template data* field are common to both local and remote join structures:

- **Role Number** — An unsigned integer identifying the role in the respective logical module template that is to be fulfilled by the candidate module.

- **Physical Dependency** — Indicates the address of a member module within the logical module entity to which the candidate module is physically connected, if the role requires a physical connection between member TIMs to be satisfied. If the physical connection between the matched candidate module and the physical dependency is lost at any time, the matched module will no be longer considered a member of the logical module entity.

- **Template Filename** — Stores the name of the template TEDS specification that was used to construct the logical module entity that is to be joined by the candidate module.

- **Template Data** — A 6 kilobyte array present only in remote join structures that is used to store the matched template TEDS specification and its associated template algorithm class. The size of the array is sufficiently large to store typical TEDS specifications and algorithm classes without consuming excessive bandwidth during transmission. Contains in the specified order: an unsigned integer indicating the length of the template TEDS, the bytes comprising the encompassed template TEDS data, an unsigned integer indicating the length of the template algorithm class, and the bytes comprising the encompassed template algorithm class.

6.2.4 Pose Update Structure

A *pose update structure* is transmitted from a TIM to other TIMs physically connected to its faces, providing the information necessary for the connected TIMs to update their position and orientation relative to that of the TIM that issued the pose update. The calculations utilized in performing this update are outlined in Section 6.5. Pose update

calls are propagated throughout the tree of physically connected modules within a logical entity, ensuring that each member module within the entity may query the pose of the others knowing that the returned pose will be relative to a given reference orientation, termed the *pose base*, which is common to the entire entity. The fields contained within a pose update structure are defined as follows:

- **Remote Pose Base** — Indicates the assigned *pose base* of the remote module, which is then assigned to the local module. Two TIMs possessing the same pose base indicates that a direct or indirect physical connection exists between them.
- **Remote Face Contact Identifier** — The identifier of the contact on the face of the remote module through which the local module transmitted its last *face identification packet*. Knowledge of this identifier facilitates the detection of the relative angular offset between the connected TIMs (see Section 3.7).
- **Remote Face Identifier** — The identifier of the face on the remote module that is physically connected to the local face of the TIM receiving the pose update structure.
- **Local Face Identifier** — The identifier of the face on the local module, as detected by the remote module, that is physically connected to the remote face of the TIM transmitting the pose update structure.
- **Remote Effective Pose** — Stores the *effective pose* of the remote TIM issuing the pose update, which is generated through the matrix multiplication of its *absolute pose* by the *face transform matrix* of its face through which it is connected to the local TIM. As stated in Section 3.7.1, the face transform matrix is currently always the identity matrix, since the faces of a TIM are rigid in its current implementation. Thus, the remote effective pose is always the absolute pose of the remote TIM.

6.3 Template and Role Matching

Creation of new logical module entities or the addition of module agents to existing logical modules is facilitated through the matching of the various *roles* in a logical module template TEDS. Each TIM in a sensing system observes the *presence* (PRE) packets being transmitted by the modules within its environment, from which it selects candidate modules that may be used in the formation or augmentation of logical modules. As described in Section 3.4.1, various fields are contained within each presence packet that reveal the capabilities of the module that transmitted it. This information is then compared to the corresponding fields present in the *role structures* contained within all loaded logical module structures. If no existing role is matched, the templates in the template TEDS directory are then searched for new matches.

Template match attempts occur only when new modules are detected within the environment, or the connection state between any two modules in the sensing system changes. This state change may be due to a connection between disconnected modules (a *wireless* to *physical* connection state change), or a disconnection between connected modules (a *physical* to *wireless* connection state change). Connection state changes trigger template match attempts because, as outlined in Section 6.2.2, connection type is one of the criteria that determines the validity of a role match.

6.3.1 Matching Existing Logical Modules

The process through which an existing logical module is matched is shown algorithmically in Algorithm 6.1. This process consists of iterating through each role structure within each logical module structure present in memory. In order to prevent infinitely recursive matching, a logical module is not allowed to match itself. This is detected by examining the address of all presence packets being tested for a role match within the logical module.

Algorithm 6.1 Existing logical module matching process.

```
procedure MATCHEXISTINGLOGICAL(packet presence)
    matched ⇐ false
    for each logical module module in module structure list do
        if module.address ≠ presense.source then
            for each role role in module do
                if role.match(presence) then
                    if module is primary module structure then
                        role.join(presence)
                    end if
                    matched ⇐ true
                    break for
                end if
            end for
        end if
    end for
    if matched = false then
        if presence.connection = 'physical' or 'local' then
            MATCHNEWLOGICAL(presence)
        end if
    end if
end procedure
```

If this address is the same as that of the logical module, testing is discontinued for that presence packet. In a role match test, each property specified in the presence packet being tested is compared to the corresponding fields within each role defined by the logical module. If the value of each property matches or falls within the bounds specified by the corresponding role field (starting with the *Connection Type* field; see Section 6.2.2), the match is considered successful. Assuming the *assignment limit* for the role is not exceeded, the search ends and a *remote join* message is issued by the *primary* module of the logical entity to the remote module that transmitted the matched presence packet. The primary module possesses the lowest address of all detected member modules comprising the logical entity and is responsible for managing its operation.

If no existing role matches are found for the tested presence packet, the local template TEDS directory is searched for a satisfactory template that may be used to form a new logical module containing a role that matches the presence packet. In addition, a module agent local to the TIM on which the test is being performed must also be matched, and the connection type between the matched remote TIM and local TIM must be either *physical* or *local*. The local module agent match is required in order to satisfy the *local member check* of the logical module (see Section 4.5.2). *Wireless* connections do not automatically trigger searches for new template matches because presence packets indicating wireless connections will be frequently detected within the environment of a sensing system comprised of a substantial amount of TIMs. Frequent template TEDS directory searches are expensive in terms of processing time and memory usage, especially when the directory consists of numerous TEDS specifications. Also, unlike with physically connected TIMs, automatic collaboration between unconnected TIMs will often be undesirable by the system user. If desired, wireless logical module formation may be invoked by the system user through an administrative interface.

6.3.2 Creating New Logical Modules

Searching For Template TEDS Matches

The process through which a new logical module is created is shown algorithmically in Algorithms 6.2 and 6.3. When a search for a new template TEDS match is initiated, each template TEDS specification file (possessing a `.mod` extension) within the template TEDS directory is processed while sub-directories and other file types are disregarded. Each template TEDS specification is loaded, parsed, and stored in memory in the form of a *template structure*, previously described in Section 6.2.1. As with an existing logical module role search, each role within the loaded template structure is iterated through and tested against the supplied presence packet. In addition, a module agent local to the TIM on which the test is being performed must also be matched. However, new logical module searches differ from existing logical module searches in that all the other presence packets within the environment list are also tested against the roles of the template in order to assimilate other candidate modules.

During each role test, a count is maintained of the number of matches found in the environment. This count is thereafter compared with the assignment limit for the role, and the assignment limit for every role must be satisfied in order for logical module structure creation to proceed. The presence packet which triggered the template search, and at least one module agent local to the TIM on which the test is being performed, must contribute to the role match count. As with existing logical module role searches, the local module agent match is required in order to satisfy the *local member check* of the logical module (see Section 4.5.2). If, however, the template search was invoked manually through an administrative interface, the local module agent match is not required since administrative interfaces do not provide sensing nor actuation functionality. If matching is unsuccessful, the loaded template structure is deallocated, and the remaining template

Algorithm 6.2 New logical module matching process.

```
procedure MATCHNEWLOGICAL(packet presence)
    for each file file in template TEDS directory do
        if file is not a directory and name of file ends with '.mod' then
            template ⇐ create template structure from file
            fullymatched ⇐ true
            for each role role in template do
                matches ⇐ 0
                for each packet environmentpresence in environment list do
                    if role.match(environmentpresence) then
                        matches ⇐ matches + 1
                    end if
                end for
                if role.assignmentlimit not satisfied by matches then
                    fullymatched ⇐ false
                    deallocate template
                    break for
                end if
            end for
            if fullymatched = true then
                if presence matched and local module agent matched then
                    CREATENEWLOGICAL(template)
                else if presence matched and matching invoked by system user then
                    CREATENEWLOGICAL(template)
                else
                    deallocate template
                end if
            end if
        end if
    end for
end procedure
```

Algorithm 6.3 New logical module creation process.

```
procedure CREATENEWLOGICAL(template template)
    address ⇐ random logical module address
    while address exists in environment list do
        address ⇐ random logical module address
    end while
    module ⇐ create logical module structure from template and address
    for each role role in module do
        matches ⇐ 0
        for each packet environmentpresence in environment list do
            if role.match(environmentpresence) then
                matches ⇐ matches + 1
                if role.assignmentlimit satisfied by matches then
                    role.join(presence)
                else
                    break for
                end if
            end if
        end for
    end for
    if local module agent matched then
        create module message handler task for module
    else
        deallocate module
    end if
end procedure
```

TEDS specifications in the template TEDS directory are tested for matches.

Creating A New Logical Module Structure

When a loaded template structure is successfully matched, iteration ends and a random 64-bit *logical module address* is generated. The most significant bit of this address, termed the *logical module bit*, is always set, and differentiates logical entities from standard module agents. The address will be assigned to a newly created logical module structure that is based on the loaded template. Before assignment, the environment list is searched, and if the address is detected within the environment, random address generation is repeated until a unique logical module address is discovered. Once a suitable address is obtained, a new logical module structure is allocated and initialized using the matched template structure. As with successful existing match searches, *remote join* messages are thereafter issued to each local and remote match found for the roles in the template structure. These transmissions need not be carried out by a primary module, since the primary module of the newly created logical module entity is currently undefined. To ensure the assignment limit of each role is satisfied, a count is maintained of the number of matches assimilated thus far in the environment.

If the template search was invoked manually by the system user through an administrative interface, the newly created logical module structure is deallocated, since an administrative interface is not allowed to become a member of a logical entity. In this case, the matched module in the environment with the lowest address always assumes the position of primary member of the logical module entity. Otherwise, a *module message handler task* (see Section 4.5) is created to process the messages received by the entity during the times in which the local module assumes the position of primary member.

6.4 Transducer Composition

6.4.1 Logical Module General Operation

The mechanism behind transducer composition in logical modules is shown in Figure 6.1 and described algorithmically in Algorithm 6.4, utilizing the example of a logical module possessing two defined roles. As described in Section 6.2.2, each role within a template from which a logical module is formed possesses a *role environment list*, which stores *member* (MEM) packets corresponding to the currently detectable module agents within the environment that were assigned to the role after having been issued a *join* request.

As shown in Figure 6.1, the logical module `F3` (8-bit addresses are depicted for simplicity), which has representative module structures present on all its members, consists of two roles. Module `1A` is the *primary member* due its possession of the lowest address among the members of the logical entity. Therefore `1A` currently processes messages transmitted to the logical module, and also generates messages on behalf of the logical entity. Since only four of the eight other modules that comprise `F3` are within range of `1A`, the regularly transmitted member packets of these four modules are the only ones that are detectable by `F3`. Thus, from the perspective of `1A`, only five modules, including itself, are currently available to fulfil the specific roles within `F3` that they were matched and assigned to.

6.4.2 Logical Module Primary Handler Operation

The *primary handler* of a typical logical module such as `F3` is shown algorithmically in Algorithm 6.4. Because all service function types are handled in a similar manner, only the processing methodology for *Get* service calls is depicted within the handler. Within the call to the *Get* service call handler, the *role environment count* for each defined role is determined. The role environment count is the number of detected members in the

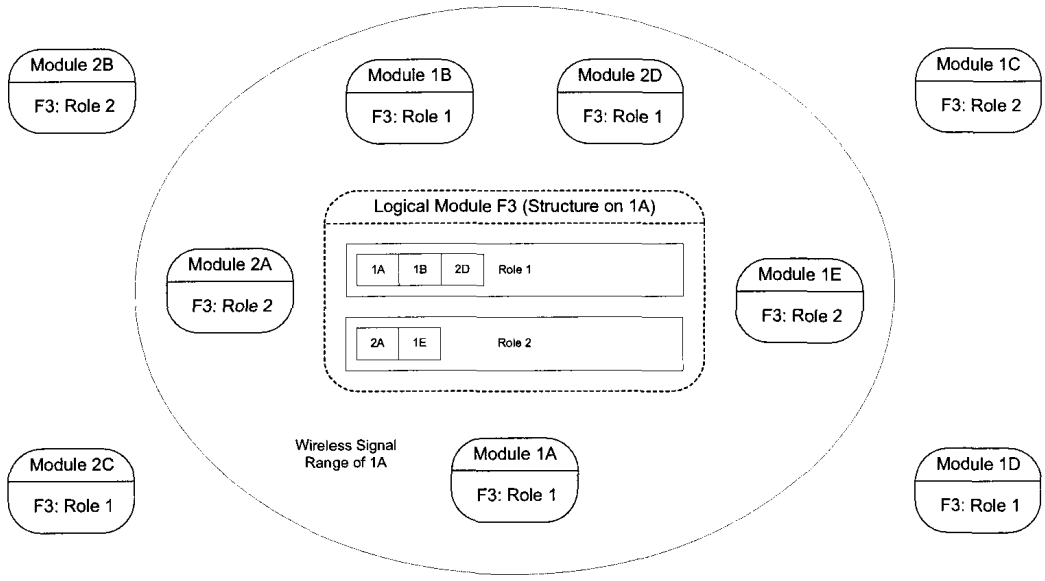


Figure 6.1: Logical module operation block diagram.

environment that satisfy a particular role. The method that facilitates determination of role environment counts within a logical module is `roleEnvironmentCount`, found within the standard library class `amss.system.Module` (see Section A.3.3). This function takes a *role number* as its only parameter.

The role environment counts are then utilized within loops in order to acquire transducer readings from all the member modules within each role, which are stored within a vector or processed on the fly during acquisition. Acquisition of member module readings is performed through the invocation of a number of *Get* service calls. The invocation of service calls is facilitated through the methods `serviceCall` and `serviceCallAsync`, also found within the standard library class `amss.system.Module`. These methods take as parameters the service call message to be transmitted, the role number to which the target module is assigned, and the index of the member packet associated with the target module (within the role environment list of the specified role). These service call methods, combined with the ability to dynamically determine role environment counts,

Algorithm 6.4 *Get* processing in primary handler for logical module with two roles.

```
procedure LOGICALPRIMARYHANDLER(module module, message call)
    if call.servicefunction = Get then
        rolecount1 ⇐ module.roleEnvironmentCount(1)
        rolecount2 ⇐ module.roleEnvironmentCount(2)
        getreturns ⇐ allocate new vector
        for i ⇐ from 0 to rolecount1 do
            getcall ⇐ create Get service call message for role 1 member i
            getreturn ⇐ module.serviceCall(getcall, 1, i)
            getreturns.append(getreturn)
        end for
        for i ⇐ from 0 to rolecount2 do
            getcall ⇐ create Get service call message for role 2 member i
            getreturn ⇐ module.serviceCall(getcall, 2, i)
            getreturns.append(getreturn)
        end for
        compositedata ⇐ process role member returns in getreturns
        deallocate getreturns contents and vector
        return ⇐ create return message for call containing compositedata
    end if
    if call was handled then
        deallocate call
        if return was generated then
            enqueue return in outgoing message queue
        end if
    end if
end procedure
```

provide convenient access to the services of the member modules available to a particular role without necessarily knowing how many members are accessible or their addresses.

After all member readings are acquired, manufacturer-provided or user-provided intelligence within the primary handler produces a composite reading based on the properties of the messages in which the readings were returned, as well as the assigned roles of the member modules from which they were acquired. This composite reading is then enqueued in the outgoing message queue to be returned to the module that invoked the initial service call on the logical module. If other logical modules are assigned to fulfil roles within the logical module, the composition process as described is recursively invoked on each member logical module until the composed readings produced by each are generated and returned up the tree of logical modules. These composed readings are then themselves composed and returned.

6.5 Pose Composition

6.5.1 Pose Representation and Theory

The *pose* (position and orientation) of each TIM in a modular sensing system is represented locally on each TIM in the form of a 4×4 *pose matrix* $\mathbf{P}$. Other popular methods as described in [68] of representing orientation itself include *Euler angles*, which are a set of three angular rotations (*roll*, *pitch*, and *yaw*) about mutually perpendicular axes, and *quaternions*. Quaternions are an extension of complex numbers in which each number possesses a single real part (w) representing a scalar, and three imaginary parts (i, j, and k) representing a vector. From each number, information describing a single angular rotation about a completely arbitrary axis may be ascertained.

A matrix is chosen to represent position and orientation rather than Euler angles and quaternions because among these, only a matrix can provide a unique representation

of a given orientation. Euler angles in particular are susceptible to the phenomenon of *gimbal lock*, where a degree of freedom may occasionally be lost due to a pitch rotation of 90 degrees causing the roll and yaw rotations to effectively occur about the same axis. Matrices also facilitate the convenient representation of position and orientation as an atomic, combined entity, enabling operations to be performed on both a position and its associated orientation simultaneously. In addition, to facilitate transformations between coordinate spaces, representations such as Euler angles and quaternions must, in any case, be converted to matrix representations. As with the *face transform matrix* (see Section 3.7.1), only 48 bytes are actually utilized to store the matrix instead of 64 bytes since the fourth row is understood to always be [0 0 0 1]. The data stored within a pose matrix is as shown in Equation 6.1:

$$\mathbf{P} = \begin{bmatrix} ax_x & ay_x & az_x & p_x \\ ax_y & ay_y & az_y & p_y \\ ax_z & ay_z & az_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.1}$$

Each column within the pose matrix represents a three-dimensional geometric vector, defined relative to the cardinal axes within a right-handed coordinate system. In this coordinate system, a positive rotation about an axis is defined as a counter-clockwise rotation from the point of view of an observer facing the opposite direction of the axis. The first three columns of the pose matrix respectively represent the x, y, and z axes defining the *object coordinate space* of the TIM, represented in terms of the standard coordinate space defined by the cardinal axes. The standard object coordinate space of a TIM is depicted in Figure 6.2. The fourth column represents the absolute position of the TIM (more specifically, the *origin* of its object coordinate space) relative to the cardinal axes, in centimetres (cm). The matrix representation chosen facilitates the transformation of

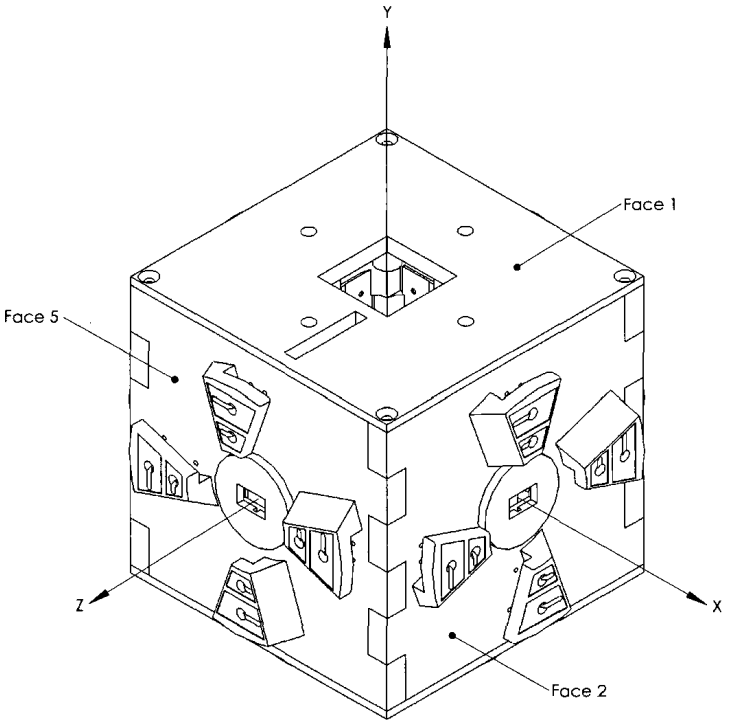


Figure 6.2: Standard TIM object coordinate space.

one TIM coordinate space into another to be performed through consecutive pose matrix multiplications applied from right to left.

Within a logical module comprised of physically connected TIMs, the pose of all member modules must be defined relative to the object coordinate space of a single member module defined as the *pose base*. The use of a pose base allows each member module to access the pose of any other physically connected member knowing that the pose matrix returned will be within the same coordinate space as the locally represented pose. This may be done even if the TIMs are indirectly connected through any number of physically connected members. The transformation of the pose matrix coordinate space of member TIM **A** into that of member TIM **B** is facilitated through the matrix

multiplication shown in Equation 6.2, applied from right to left:

$$\mathbf{P_{A_{new}}} = \mathbf{P_B} \times \mathbf{P_A} \tag{6.2}$$

6.5.2 Pose Composition Process

Pose Transformation Theory

As described in Section 3.7.3, *pose updates* are triggered whenever physical connections or disconnections between TIMs occur. The necessary information facilitating the update of a pose matrix is transferred within *pose update structures*, outlined in Section 6.2.4. This information is used to transform the local pose matrix of the TIM receiving the pose update structure through a series of rotations and translations. A pose rotation through the cardinal z, y, and x axes of angles γ, β, and α respectively, in that order, followed by a pose translation along these axes of Δz, Δy, and Δx centimetres respectively, is performed through the matrix multiplication shown in Equation 6.3, applied from right to left:

$$\mathbf{P_{new}} = \mathbf{T} \times \mathbf{X_{rot}} \times \mathbf{Y_{rot}} \times \mathbf{Z_{rot}} \times \mathbf{P} \tag{6.3}$$

where:

$$\mathbf{Z_{rot}} = \begin{bmatrix} cos\ \gamma & -sin\ \gamma & 0 & 0 \\ sin\ \gamma & cos\ \gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.4}$$

$$\mathbf{Y_{rot}} = \begin{bmatrix} cos\ \beta & 0 & sin\ \beta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\ \beta & 0 & cos\ \beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.5}$$

$$\mathbf{X_{rot}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\ \alpha & -sin\ \alpha & 0 \\ 0 & sin\ \alpha & cos\ \alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.6}$$

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

Local Pose Update Transformations

Upon receiving a pose update structure as transmitted within an *Update Pose* service call received from a physically connected TIM, a series of transformations are carried out from which the new pose of the local TIM relative to the physically connected TIM will be determined. Firstly, the starting pose matrix on which the transformations will be carried out is set to the inverse of the *face transform matrix* associated with the connected face as indicated in the *local face identifier* field of the pose update structure. This is done since, for convenience, subsequent transformations are applied to the face as if it was in its default position and orientation within the object coordinate space of the local TIM. As stated in Section 3.7.1, the face transform matrix is currently always the *identity matrix*, since the faces of a TIM are rigid in its current implementation. Thus, the starting pose matrix is also always the identity matrix, which is its own inverse.

The first transformation applied to the pose matrix rotates it about the origin in a manner such that the positive x-axis in the object coordinate space of the local TIM passes directly through the centre of the locally connected face and is perpendicular to it. This requirement is already satisfied by Face 2 within the object coordinate space of the TIM. The locally connected face is that possessing the *identifier* (see Section 3.7)

specified in the *local face identifier* field of the pose update structure. This transformation prepares the locally connected face to be trivially rotated about the x-axis in a subsequent transformation based on its angular offset relative to the remote connected face. The rotations necessary to achieve this transformation, depending on the identifier of the locally connected face, are shown in Table 6.1.

Table 6.1: Rotations required to bring x-axis perpendicular to locally connected face.

Local Face Identifier	Required Rotation About Origin
2	None
3	$-90°$ about y-axis
4	$180°$ about y-axis
5	$90°$ about y-axis
6	$90°$ about z-axis, then $90°$ about x-axis

The second transformation applied to the pose matrix rotates it about the x-axis in a manner such that the relative angular offset between the locally connected face and the remote connected face matches that implied by the *contact connection pattern* (see Section 3.7.3) between the two faces. The contact connection pattern may be immediately determined from the value contained within the *remote face contact identifier* field of the pose update structure, which indicates the contact on the remote connected face to which Contact 1 on the local connected face is attached. The angular offset associated with each contact connection pattern, and thus the rotation to apply, is as shown in Table 3.1. The rotations necessary to achieve this transformation, based on the remote face contact identifier, are shown in Table 6.2.

The third and fourth transformations complete the movement of the local pose from the object coordinate space of the local TIM into the object coordinate space of the remote TIM. The third transformation is a 12 cm (equal in magnitude to the length of the edge of a TIM) negative translation of the local pose along the x-axis. This transformation places

Table 6.2: Rotations about x-axis required to achieve correct relative angular offset.

Remote Face Contact Identifier	Required Rotation About x-axis
1	None
2	90°
3	180°
4	−90°

the local and remote TIMs directly adjacent to each other within the object coordinate space of the remote TIM, with the locally connected face of the local TIM in contact with Face 4 of the remote TIM. This may not necessarily be the remote face to which the local face is actually physically connected.

The fourth and final transformation applied to the pose matrix is a rotation within the object coordinate space of the remote module that moves the local TIM and its locally connected face, if necessary, adjacent to the remote face on the remote TIM to which it is actually physically connected. The rotations necessary to achieve this transformation, depending on the identifier of the remote connect face given in the *remote face identifier* field, are shown in Table 6.3.

Table 6.3: Rotations required to move locally connected face adjacent to correct remote face.

Remote Face Identifier	Required Rotation About Origin
2	180° about y-axis
3	−90° about y-axis
4	None
5	90° about y-axis
6	90° about z-axis, then −90° about y-axis

After the preceding transformations are applied, the local pose is now completely defined in terms of the object coordinate space of the remote TIM, and specifies an appropriate position and orientation based on the face on the remote TIM to which the

local face is connected. However, as described in Section 6.2.4, the coordinate spaces of all physically connected TIMs must be defined relative to the TIM in the composite entity designated the *pose base*. Since the pose of the remote module would have already been transformed such that it is defined in terms of the coordinate space of the pose base, the local pose may also be brought into the coordinate space of the pose base by multiplying it by the remote pose, as per Equation 6.2. This operation is valid even if the pose base is only indirectly connected to both the local and remote TIMs, due to the accumulative effect of multiplying transformation matrices in which all previously applied transformations are carried over into successive transformations.

Recursive Pose Update Invocation

Upon completion of the pose update process, the pose of other TIMs physically connected to the other faces of the local module will require updating. For each of the physically connected TIMs (except the TIM that issued the initial pose update service call to the local module), an appropriate pose update structure is allocated as part of an *Update Pose* service call message. These service call messages are then enqueued into the outgoing message queue for transmission.

6.6 Summary

In this chapter, the *composition layer* of the software architecture was described. In this layer, platform-independent *logical module template algorithms* are loaded and executed in order to provide intelligence to a collaborating group of TIMs. The *primary module* of the logical entity, which processes and transmits messages on its behalf, actively seeks new candidate module agents within the environment that may fulfil a behavioural *role* within the logical module. Physically connected TIMs within the logical entity intelligently

transmit their position and orientation to each other and ensure that the pose of all physically connected members is defined relative to that of a member designated the *pose base*. The following chapter will provide a description of the process used to evaluate the operation and performance of the software architecture as well as a description of the results observed and an analysis of these results.

Chapter 7

Architecture Evaluation

7.1 Introduction

This chapter presents an evaluation of the behaviour and performance of the software architecture when utilized on actual TIM hardware. This evaluation will be facilitated through two tests, in which select homogeneous and heterogeneous sensors and actuators will be associated with TIMs and connected together. Upon assuming a composite representation, the interactions of the TIMs are then logged locally on the non-volatile storage present on each TIM and examined thereafter. The module TEDS specifications, template TEDS specifications and template algorithm classes utilized for the purposes of evaluating the software architecture are listed in Appendix B.

The first test will be used to evaluate the operation of a logical module in which the constituent TIMs interact entirely through wireless communication. The second test will be used to evaluate the behaviour of a logical module in which the constituent TIMs are physically connected in various orientations, and interact through both wireless communication as well as through their physically connected faces. During these tests, performance criteria that strongly impact the real-world performance of a composite sensing system

are considered for each constituent layer of the architecture implemented on top of the real-time operating system, drivers, and module hardware. These criteria are outlined below:

- **Communication Layer** — Channel reservation latency during execution of the Medium Access Control (MAC) protocol, and message transmission speed between TIMs during execution of the Positive Acknowledgement with Retransmission (PAR) protocol.
- **Middleware Layer** — Latency encountered between the invocation of a service function on a remote TIM and the reception of the associated return message by the invoking TIM.
- **Virtual Machine** — Speed of bytecode execution in template class methods during which service calls may occur intermittently.
- **Composition Layer** — Logical module agent startup memory utilization and correctness of the expected behaviour of the logical entity being evaluated.

7.2 Wireless Collaboration Behaviour

7.2.1 Evaluation Setup

The purpose of this test is to evaluate the behaviour of a modular sensing system when its constituent TIMs are wirelessly connected. A modular sensing system will be created in which a servo motor TIM, an digital accelerometer TIM, and an analog light-dependent resistor (LDR) TIM are placed within range of each other. The behaviour of the composite entity comprising the heterogeneous sensor and actuator modules is then examined and evaluated according to the performance criteria specified in Section 7.1. Throughout

the collaboration tests, the TIMs relay their status to and receive data from the system user through a console-based administrative interface running on a personal computer workstation.

The apparatus utilized to perform testing of a wirelessly interacting modular sensing system is depicted in Figure 7.1. For the purposes of TIM identification throughout the evaluation, each module utilized is assigned a letter. In this particular test, the LDR TIM is designated Module A, the accelerometer TIM is designated Module B, and the servo TIM is designated Module C. Interfaces to three 5 V adaptors and an RS232 port connected to the personal computer workstation are provided through means of a standard breadboard. These interfaces solely serve the respective purposes of providing a stable voltage to the TIMs and providing a link to their administrative task through which their behaviour may be monitored, and otherwise have no influence of the behaviour of the composite system.

Each TIM is equipped with an SD card that stores within the *module TEDS directory* (see Section 2.5.2) the *module Transducer Electronic Data Sheet specifications* (TEDS) that identify and describe the characteristics of the transducer associated with it. The module TEDS specifications describing the characteristics of the accelerometer, LDR, and servo TIMs are listed in Sections B.2.1, B.2.3, and B.2.4 respectively. Also stored on the SD card are a collection of *template TEDS specifications* and their associated *template algorithm classes*, located in the *template TEDS directory* and *template class directory* respectively (see Section 2.5.2). As stated in Section 4.5, a template TEDS and an associated template class together define the identity, characteristics, and behaviour of a particular combination of specific classes of collaborating TIMs. The template TEDS specification and the source code of the template algorithm class applicable to the TIMs utilized in this test are listed in Sections B.3.2 and B.4.2 respectively.

7.2.2 Evaluation Procedure

Upon discovering each other, the modules in the sensing system are expected to be capable of utilizing an appropriate template specification and class to automatically form a composite entity that implements a new behaviour. To prevent multiple template matches from limiting the free memory available for the logical module to operate, the only suitable template provided within the non-volatile storage on the TIMs during this test is that of rotational actuator control through the averaging of voltage sensor readings (see Sections B.3.2 and B.4.2), corresponding to the TIMs available within in the system. Thus, a single composite system should be created that finds and utilizes the average readings of all the available voltage-based sensing TIMs (specifically, the LDR in a potential divider configuration and the accelerometer) in the system to influence the position of all the available rotational actuator TIMs (the servo motor), as per the provided template specification and template class. Module C (the servo TIM) is the module possessing the lowest address in the system and thus should automatically designate itself the *primary module*, responsible for executing the template class upon formation of the logical entity.

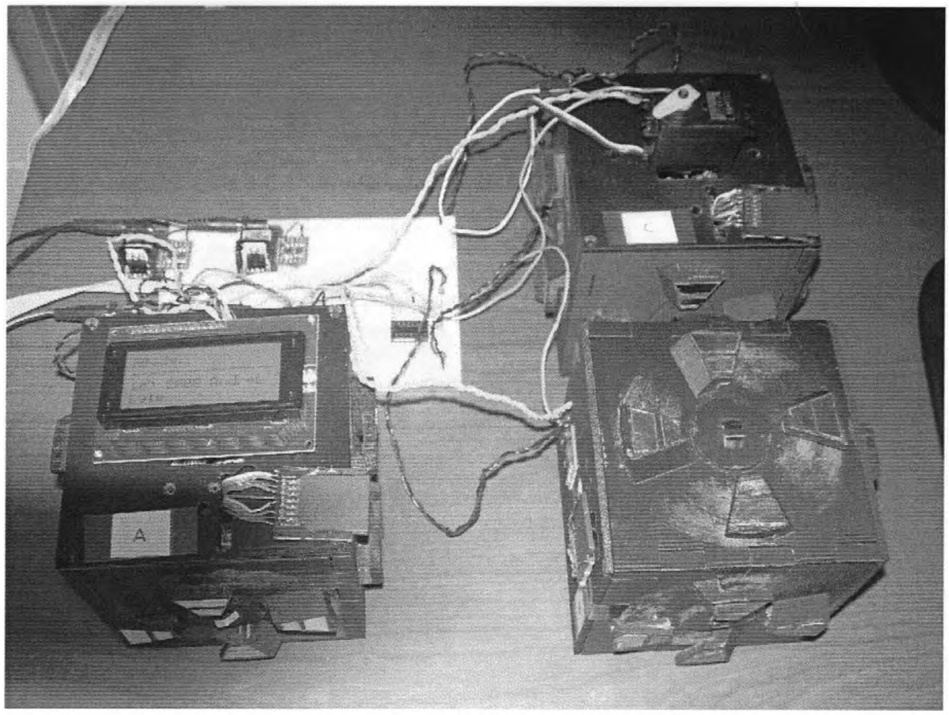
To test if the described system behaviour is realized, the accelerometer TIM is physically rotated through five increasing angular degree positions of 0°, 45°, 90°, 135°, and 180° once the formation of a logical module entity is confirmed through the administrative interface. For each accelerometer angle, the LDR TIM is also exposed to high and low levels of ambient light. The angular position assumed by the servo motor TIM in response to changes to the duty cycle of the input pulse-width modulation (PWM) signal (which itself changes due to variations in the readings continuously acquired and averaged by the primary module during execution of the template class) provided by its associated TIM is then examined and analyzed in order to determine the correlation between the accelerometer TIM angular positions, LDR TIM ambient light voltage readings, and servo

TIM angular positions. The angular position of the servo motor is more easily identified through a black indicator attached to the rotating head.

7.2.3 Results and Analysis

Figure 7.1 depicts the behaviour of the composite sensing system formed in the first evaluation setup, which consists entirely of wirelessly interacting TIMs. As previously outlined in Section 7.2.2, the data acquired from two sensor TIMs (LDR module A and accelerometer module B) is used to wirelessly control the behaviour of an actuator TIM (servo motor module C). Having detected that the TIMs present in the environment, including itself, satisfy the requirements for matching the `ServoCon.mod` template (see Section B.3.2), Module C loads the template into memory from the directory `tmpl` in its non-volatile storage as well as its associated template class `ServoCon` (source code listed in Section B.4.2) from the directory `amss/algo`. A logical module with a randomly determined logical address is formed locally on Module C, with the most significant bit in the logical address correctly set. The logical module thereafter issues *Join* service calls to the module agents running on Modules A, B, and C that represent the interface to the associated transducers on these TIMs. Local representations of the logical entity are also created on Modules A and B. Correctly identifying itself as the primary module of the composite entity due to its possession of the lowest address in the environment, Module C commences execution of the template class.

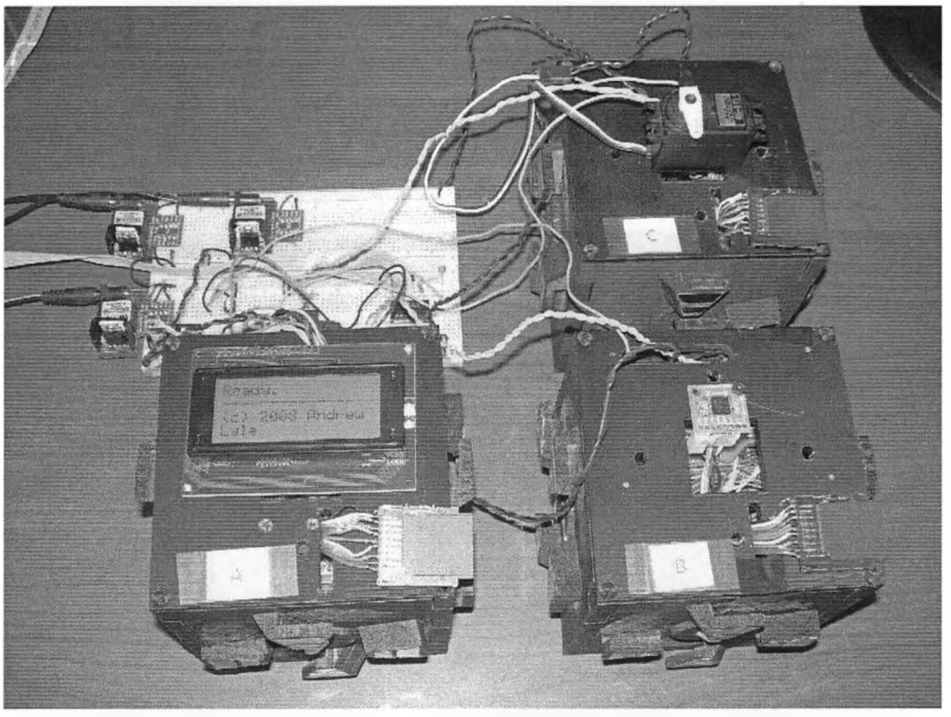
In the `ServoCon` template class executing on Module C, the readings from all of the acceleration (utilizing solely their x-axis readings) and voltage sensing module agents which comprise the logical entity are continuously acquired and averaged, through the use of *Get* service calls, to produce a value that is then applied, through the use of *Set* service calls, to all of the rotational module agents within the entity. The behaviour observed is as depicted in Figure 7.1, where the position of the rotational head of the



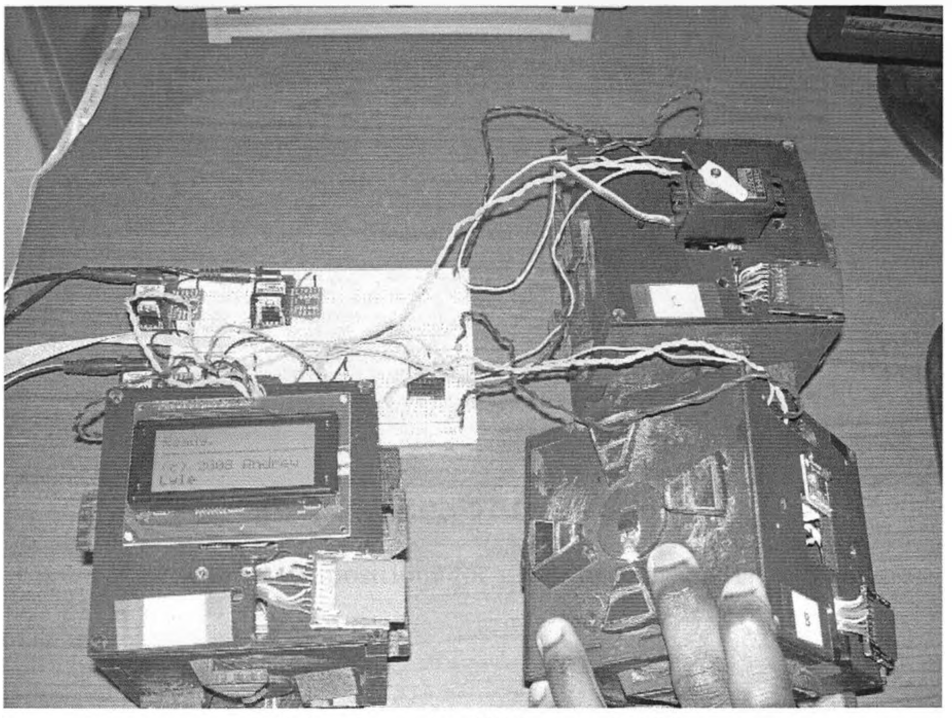
(a) Accelerometer TIM angle of 0°.

(b) Accelerometer TIM angle of 45°.

Figure 7.1: Servo TIM positions for given accelerometer TIM angles.

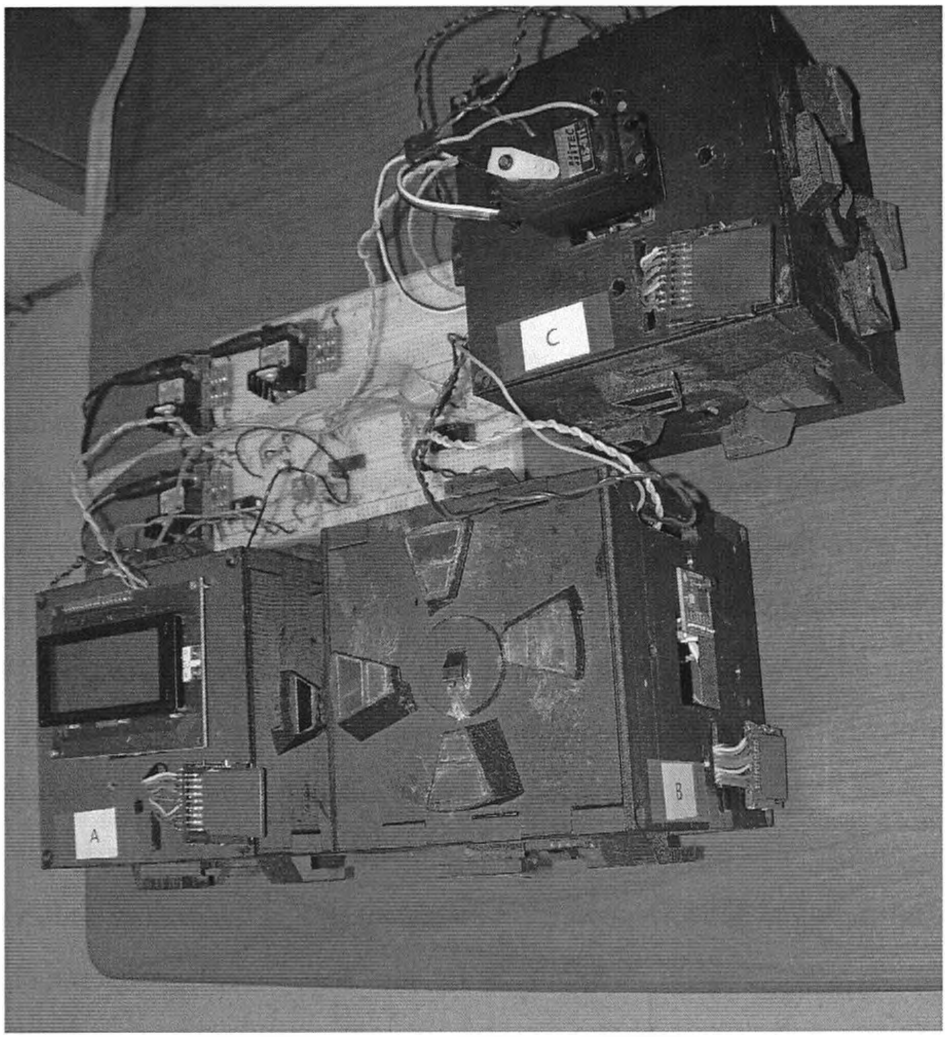


(c) Accelerometer TIM angle of 90°.



(d) Accelerometer TIM angle of 135°.

Figure 7.1: Servo TIM positions for given accelerometer TIM angles.



(e) Accelerometer TIM angle of 180°.

Figure 7.1: Servo TIM positions for given accelerometer TIM angles.

servo motor directly assumes, within the limits of its physical rotational range, an angle proportional to the degree to which the accelerometer is offset about its x-axis. Likely due to its limited sensitivity, varying the ambient light incident on the LDR did not affect the angle of the servo to as great a degree as changing the orientation of the accelerometer; however, minor variations were distinctly noticeable, indicating that the voltage readings returned by the LDR were in fact influencing the average reading applied to the servo motor.

As a result of overhead encountered during the transmission and processing of the continuous stream of service calls issued by the primary module to the sensor module agents within the logical entity, reliable real-time performance was difficult to achieve. An intentional half-second delay was introduced between sensor reading acquisitions and averaging in order to limit the frequency of dropped service call messages and achieve complete stability. Nevertheless, the modular sensing system exhibited correct behaviour, with real-time performance limited mainly by the capabilities of the microcontroller and wireless transceiver utilized in the TIMs. Improvements in real-time performance may be attained through the utilization of a more recent variant of the ARM microcontroller as well as a transceiver capable of higher sustained transmission speeds in a newer version of the TIM hardware. The cost of such attaining such components for prototyping purposes is rapidly falling to reasonable levels, and these components will facilitate greatly reduced latencies in scenarios where service calls are continuously invoked.

7.3 Physical Collaboration Behaviour

7.3.1 Evaluation Setup

The purpose of this test is to evaluate the behaviour of a modular sensing system when its constituent TIMs are physically connected. A modular actuator system will be created

in which two 16 × 4 character HD44780-based liquid-crystal display (LCD) TIMs are physically connected in various orientations. The behaviour of the composite entity comprising the homogeneous modules is then examined and evaluated as per the performance criteria specified in Section 7.1.

The apparatus used to test the physically interacting modular sensing system is depicted in Figures 7.2 and 7.3. As with the wirelessly interacting composite module test, each module utilized in this test is assigned a letter for the purposes of identification. In this particular test, one LCD TIM is designated Module A, while the other is designated Module B. A breadboard is also used within the experiment solely to provide an interface to a single 5 V adaptor to provide power and an RS232 port connected to a personal computer workstation for administrative purposes. As with the wirelessly interacting composite module test, the components on this breadboard otherwise have no influence on the behaviour of the composite system.

The module TEDS specification describing the characteristics of the two LCD TIMs utilized in this test is listed in Section B.2.2. This specification is stored within the standard module TEDS directory located on the SD cards local to each module. The template TEDS specification applicable to the TIMs utilized in this test and the source code of its associated template class are listed in Sections B.3.1 and B.4.1 respectively. The template TEDS specification and compiled template class bytecode are stored in the template TEDS directory and template class directory on the local SD cards respectively.

7.3.2 Evaluation Procedure

After being placed within range of each other, the LCD modules are expected to detect each other and attempt to form a composite entity. To prevent multiple template matches from limiting the free memory available for the logical module to operate, the only suitable template provided within the non-volatile storage on the TIMs during this test is that

enabling physically connected text displays to produce a larger, composite display based on their detected orientation and relative positions (see Sections B.3.1 and B.4.1), which is satisfied by the LCD modules within the system only when they are physically connected. Thus, while the template is matched based on module type, class, and data type, the modules should not form a composite entity until they detect a physical connection to each other through their faces. Upon connection, a composite system should be created that finds and utilizes all of the available text displays (the two LCD TIMs) in order to form a suitable logical entity that effectively functions as a larger display if the displays detect that they are connected in a suitable orientation, as per the provided template specification and template class. The LCD TIM designated Module B is the module possessing the lower address in the system and thus should automatically designate itself the *primary module*, responsible for executing the template class upon formation of the logical entity.

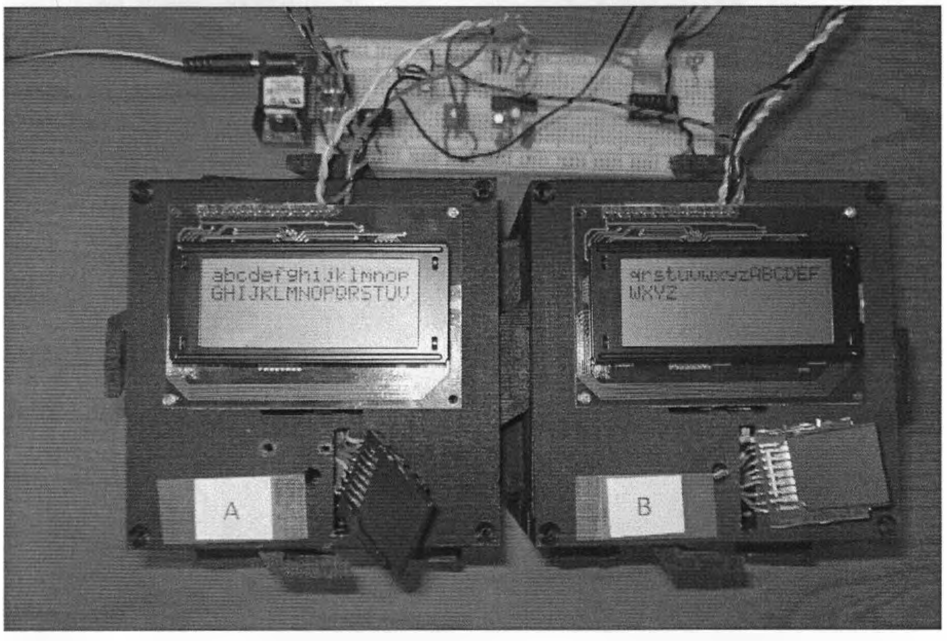
To test if the described system behaviour is realized, the LCD TIMs are connected together in four configurations such that the LCD displays on the modules are aligned in suitable horizontal and vertical orientations. For the horizontal orientation, the LCD TIM designated Module A is connected on the left of the one designated Module B. In this configuration, Face 2 of Module A is connected to Face 4 of Module B, where the assigned face numbers are as described in Section 3.7. Once formation of a logical module entity is indicated through confirmation received on the administrative interface, alphanumeric text strings are then transmitted through the administrative interface to the logical module. The text output on the LCD displays is then examined and analyzed in order to determine the correlation between the orientations of the LCD display modules and the text outputs observed. In this configuration, the two 16 $\times$ 4 character LCD displays should form and behave as a logical 32 $\times$ 4 LCD display, thus possessing double the width. The test is then repeated with Face 2 of Module B connected to Face 4 of

Module A, which should produce the same behaviour.

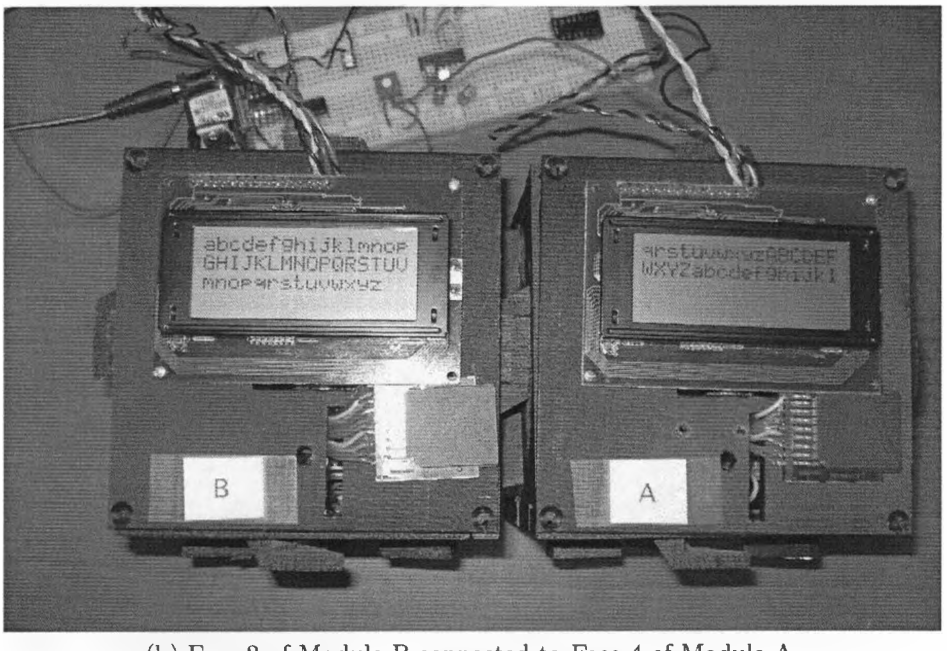
After completion of the second test, the LCD displays are connected in a vertical configuration, in which the LCD TIM designated Module A is connected above the one designated Module B. In this configuration, Face 5 of Module A is connected to Face 3 of Module B. Once formation of a logical module entity is indicated through confirmation received on the administrative interface, alphanumeric text strings are then transmitted through the administrative interface to the logical module, as done previously for the horizontal configurations. The text output on the LCD displays is then examined and analyzed in order to determine the correlation between the orientations of the LCD display modules and the text outputs observed. In this configuration, the two 16 $\times$ 4 character LCD displays should form and behave as a logical 16 $\times$ 8 LCD display, thus possessing double the height. The test is then repeated with Face 5 of Module B connected to Face 3 of Module A, which should produce the same behaviour.

7.3.3 Results and Analysis

Figures 7.2 and 7.3 depict the behaviour of the composite sensing system formed in the second evaluation setup, which consists of TIMs interacting wirelessly as well as physically through their faces. As previously outlined in Section 7.3.2, alphanumeric text strings are transmitted through the administrative interface to the logical module formed upon the physical connection between the two LCD modules A and B present in the system. These strings are used to confirm that the LCD modules are correctly behaving as an effectively larger LCD entity for any valid connection orientation between the modules. Upon each of the four physical connections made between the TIMs in this test, the *pose base* addresses of both modules are updated to indicate that a physical connection now exists between them. Due to the fact that Module B possesses the lower address of the two modules, its orientation in each case correctly becomes the reference orientation for

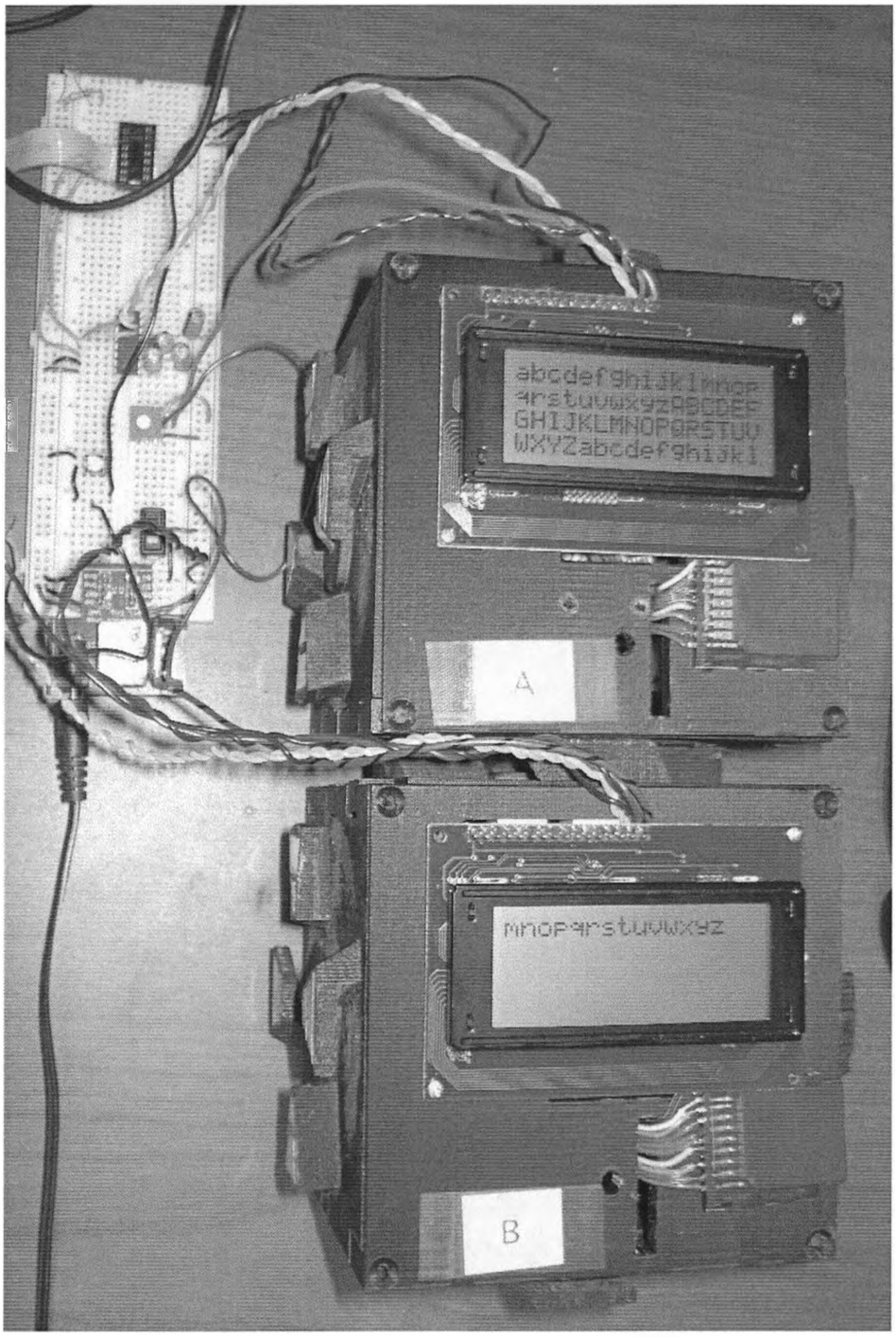


(a) Face 2 of Module A connected to Face 4 of Module B.



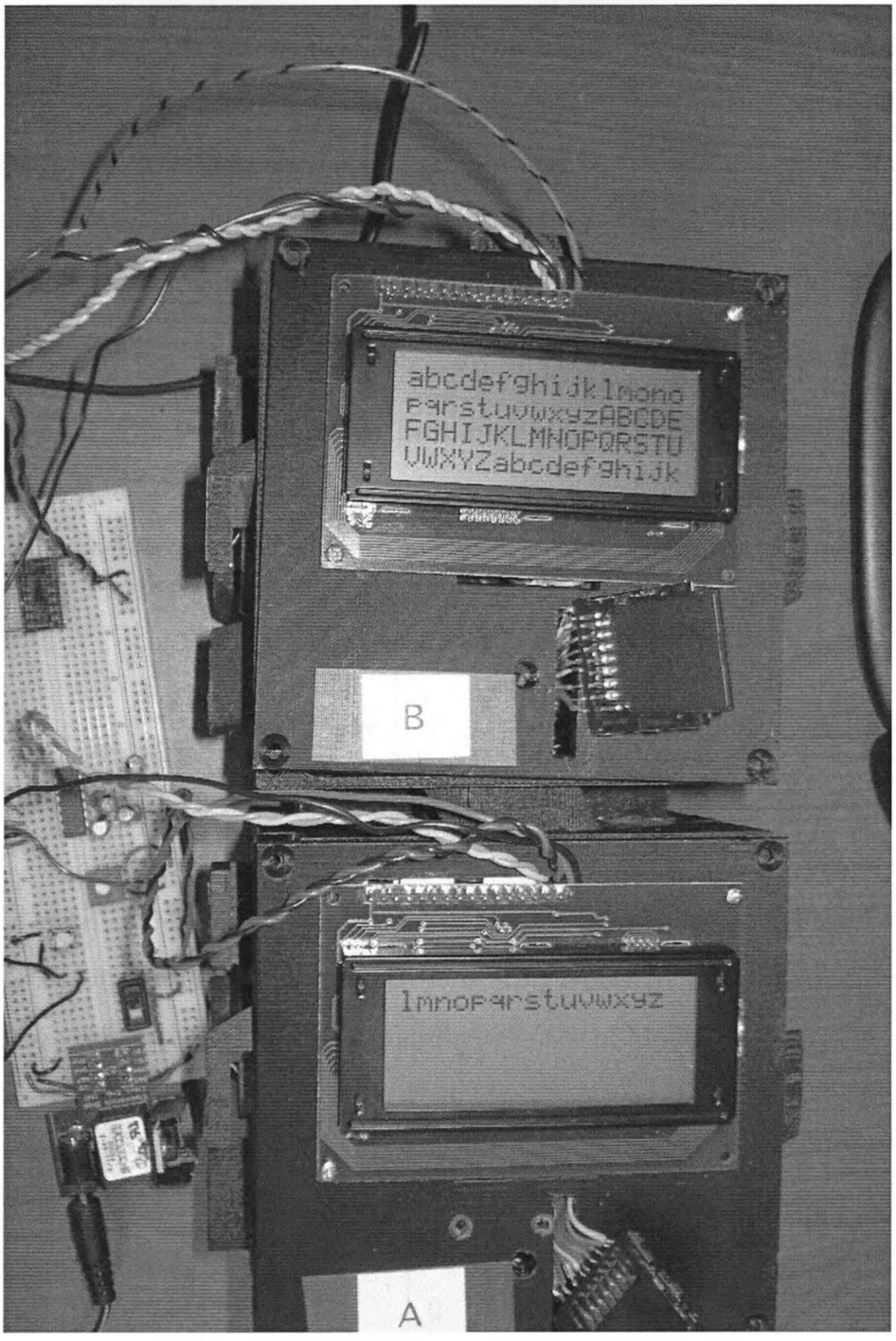
(b) Face 2 of Module B connected to Face 4 of Module A.

Figure 7.2: 32 × 4 character composite LCD TIM configurations.



(a) Face 5 of Module A connected to Face 3 of Module B.

Figure 7.3: 16 × 8 character composite LCD TIM configurations.



(b) Face 5 of Module B connected to Face 3 of Module A.

Figure 7.3: 16 × 8 character composite LCD TIM configurations.

both modules, and thus the pose bases of both modules are set to the address of Module B. As the pose base, the *pose matrix* of Module B correctly remains the identity matrix upon each physical connection made to Module A. The updated pose matrices of Module A as logged by the TIM hardware upon each physical connection to Module B are listed below, with the standard TIM object coordinate space defined as shown in Figure 6.2, the face numbers designated as per Section 3.7, and the magnitude of the length of the edge of a TIM equal to 12 centimetres (cm):

- Pose matrix of Module A when Face 2 of Module A is connected to Face 4 of Module B, as shown in Figure 7.2a, correctly indicating a negative translation of 12 cm along the x-axis relative to the object coordinate space of Module B:

$$\mathbf{P_A} = \begin{bmatrix} 1 & 0 & 0 & -12 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.1}$$

- Pose matrix of Module A when Face 2 of Module B is connected to Face 4 of Module A, as shown in Figure 7.2b, correctly indicating a positive translation of 12 cm along the x-axis relative to the object coordinate space of Module B:

$$\mathbf{P_A} = \begin{bmatrix} 1 & 0 & 0 & 12 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.2}$$

- Pose matrix of Module A when Face 5 of Module A is connected to Face 3 of Module B, as shown in Figure 7.3a, correctly indicating a negative translation of

12 cm along the z-axis relative to the object coordinate space of Module B:

$$\mathbf{P_A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -12 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.3}$$

- Pose matrix of Module A when Face 5 of Module B is connected to Face 3 of Module A, as shown in Figure 7.3b, correctly indicating a positive translation of 12 cm along the z-axis relative to the object coordinate space of Module B:

$$\mathbf{P_A} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 12 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.4}$$

After each physical connection between the two LCD TIMs, Module B detects that both itself and Module A satisfy the requirements for matching the `LCDMerge.mod` template (see Section B.3.1), and loads the template into memory from its local `tmpl` directory as well as the associated template class `LCDMerge` (source code listed in Section B.4.1) from its local `amss/algo` directory. Upon each physical connection, a logical module with a randomly determined logical address is formed locally on Module B, with the most significant bit in the logical address correctly set for each logical module formed. The logical module thereafter issues *Join* service calls to the module agents running on both Module A and Module B that respectively represent the interface to the LCD module on each TIM. For each logical module formed, a local representation of the entity is also created on Module A. Correctly identifying itself as the primary module of the composite entity

due to its possession of the lowest address in the environment, Module B commences execution of the template class in each case.

One of four alphanumeric strings (each of which starts with the letters of the alphabet in lower case followed by upper case) is transmitted through the administrative interface to each of the logical display modules upon formation, through the use of a *Set* service call. The behaviour observed is as depicted in Figures 7.2 and 7.3, where as long as the LCD displays associated with the member TIMs are aligned horizontally or vertically alongside each other in a common plane, the alphanumeric string is always displayed in a consistent left-to-right, top-to-bottom fashion across TIMs. This behaviour is realized even if the relative positions of Module A and Module B are swapped, resulting in an effectively larger display of either 32 × 4 characters as depicted in Figure 7.2, or 16 × 8 characters as depicted in Figure 7.3. Since the member TIMs are physically connected, they possess a common pose base, and the `LCDMerge` template class executing on Module B is therefore able to query and analyze their *pose vectors* (see Figure 5.1), derived from their respective pose matrices, in order to determine their relative orientations and thus the overall geometry of the logical entity. With knowledge of the overall geometry, the original alphanumeric string is internally split (if necessary) into segments by the primary module, each of which is recursively transmitted using *Set* service calls to the appropriate member LCD TIMs in order to achieve the correct behaviour.

Due to limitations in the amount of memory available within the TIMs, which restrict the complexity of the template classes utilized by logical module agents as well as the size of the structures used to maintain the state of the logical module agents themselves, the `LCDMerge` template class utilized in this test is unable to scale beyond two LCD modules. Nevertheless, the logical module entity exhibited the ability to assume a new behaviour based on the relative orientations between its physically connected member modules, and by extension, the overall geometry of the composite entity. Through the

utilization of TIMs possessing greater amounts of memory (which is quickly becoming less cost prohibitive as semiconductor fabrication techniques improve), scaling to three LCD modules and beyond would not be difficult to achieve.

7.4 Collaboration Performance Analysis

In this section, measurements obtained during the evaluation of various performance criteria in the wireless and physical collaboration tests are graphically presented. For each criterion, the maximum, minimum, and mean performance readings are shown, as well as the standard deviation from the mean. These readings are then analyzed in order to characterize the performance of the architecture in each case. Latency and speed performance evaluations were facilitated through the use of the hardware system clock provided by the LPC2148 microcontroller present within each TIM, which is read immediately before and immediately after each event being timed. At least one hundred and up to three thousand event occurrences are timed, logged, and analyzed during the collaboration test runs. Upon the occurrence of each event, the elapsed time is obtained by calculating the difference between each pair of system clock readings and is subsequently utilized to derive performance data. This data is thereafter logged to non-volatile storage for analysis. Reading the system clock requires only a few tens of cycles (23 instructions executed at 60 MHz, most executed in effectively a single cycle due to pipelining), thus completing in about 1 μs or less. This overhead is considerably less than the time duration of events encountered during the operation of the software architecture, which are on the order of milliseconds, and is thus deemed negligible.

7.4.1 Channel Reservation Latencies

In Figure 7.4, the MAC protocol channel reservation latencies encountered at the communication layer during the collaboration tests are depicted. This latency is defined as the time elapsed between the transmission of an RTS packet and reception a CTS packet in response. The results show that the time required to reserve a channel is on the order of tens of milliseconds (about 40 ms to 50 ms on average). In comparison, the channel reservation latencies encountered in typical 802.11/WiFi networks, which employ the MACAW-based *CSMA/CA* (Carrier Sense Multiple Access with Collision Avoidance) protocol that operates similarly to the MAC protocol utilized in this software architecture, are typically around 20 ms when few nodes are contending for access to the wireless channel [69].

The relatively large channel reservation period (which was rarely, but occasionally, on the order of seconds) typically encountered during the operation the software architecture can likely be attributed to the greatly reduced interrupt response sometimes encountered within the software architecture due to its frequent usage of *critical sections* during timing-critical operations and operations that must be performed atomically. Critical sections are facilitated through the disabling of interrupts, during which reception of medium allocation packets by the wireless transceiver may go undetected for a substantial period of time. The impact of the large channel reservation overhead is somewhat mitigated by the fact that it occurs only once per message transmission, is independent of the message length, and only becomes a major issue in real-time scenarios in which data is streamed between TIMs.

7.4.2 Message Transmission Speeds

Figure 7.5 depicts the speeds at which messages are wirelessly transmitted using the PAR protocol at the communication layer during the collaboration tests. The results clearly

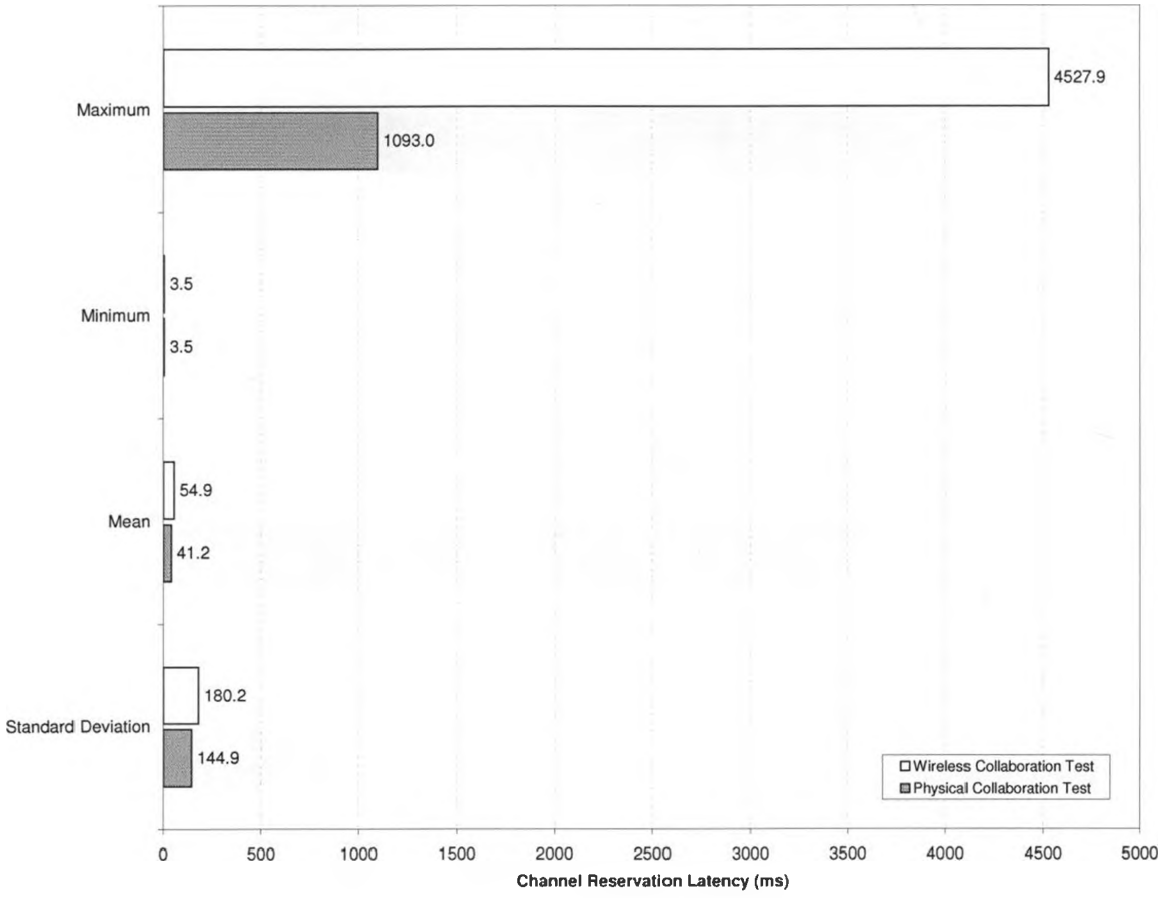


Figure 7.4: MAC protocol channel reservation latencies.

show a maximum transmission speed of about 9,650 bytes per second, or about 9.42 kBps (kilobytes per second), with mean transmission speeds approaching this maximum. The messages transmitted during the physical collaboration test were generally larger than those transmitted during the wireless collaboration test; a message data field of at least 52 bytes was required to store the alphanumeric strings transmitted in the physical collaboration test, compared to the at most 12 bytes required in the wireless collaboration test to store the three floating point accelerometer axis readings transmitted. This results in an increased chance of packet retransmissions during the transmission of the larger messages due to interference encountered in the popular 2.4 GHz radio frequency spec-

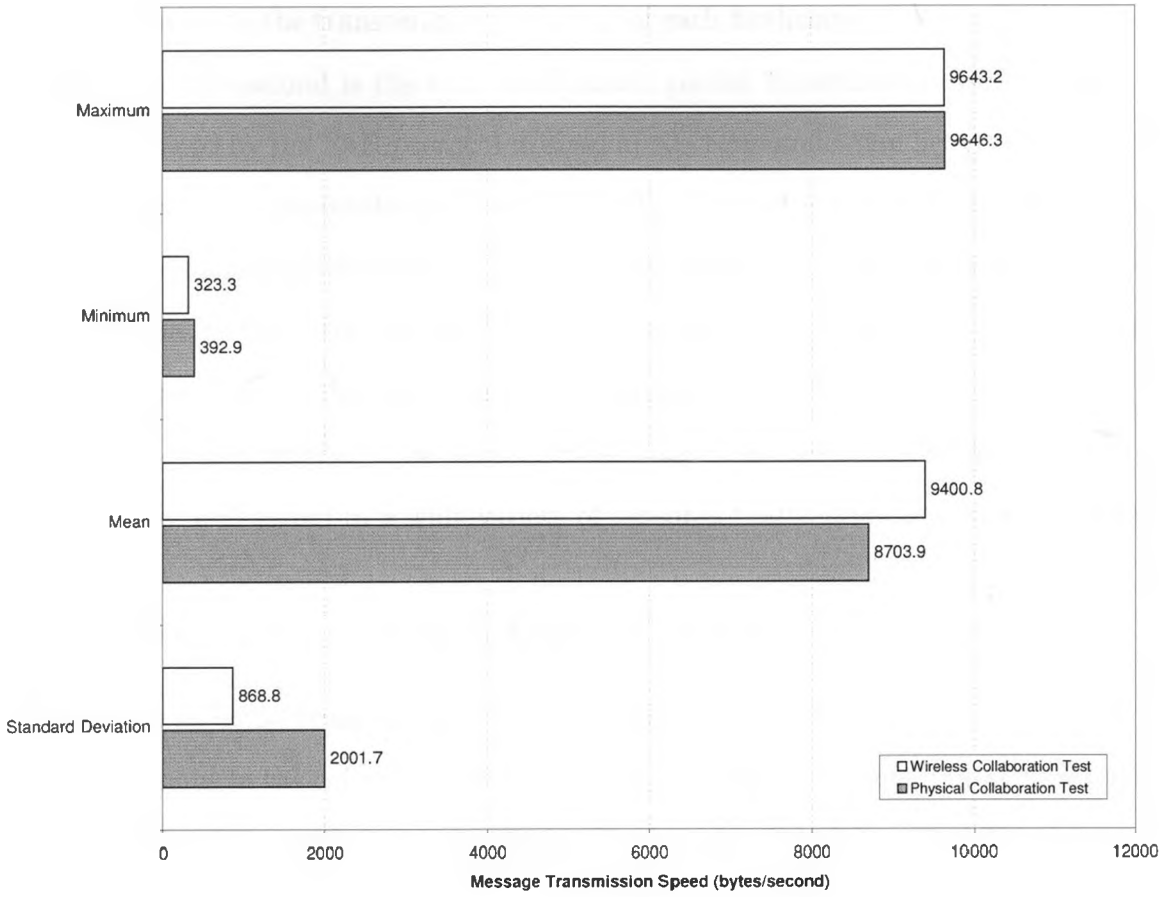


Figure 7.5: PAR protocol message transmission speeds.

trum utilized by the nRF24L01 transceiver within the TIMs. As a result, a lower mean message transmission speed was attained during the physical collaboration test, as well as a noticeably larger transmission speed variance. Nevertheless, the mean transmission speed was far closer to the maximum attained than the minimum.

The 9.42 kBps maximum transmission speed is much lower than the specified theoretical maximum of 2 Mbps (megabits per second), or 256 kBps, attainable by the nRF24L01 transceiver. This performance discrepancy may be attributed to a number of key factors. The first is the speed of the 1.875 MHz SPI bus linking the LPC2148 microcontroller and the nRF24L01 transceiver, the bandwidth of which is partially consumed in the issuing of

commands bytes to the transceiver to inform it of each forthcoming packet payload to be transmitted. The second is the acknowledgement packet transmission overhead encountered as required by the PAR protocol utilized at the communication layer, in which each packet requires an acknowledgement packet (ACK) to be returned by the recipient before the next packet transmission may occur in order to improve the reliability of the wireless link. The final factor is the overhead encountered during the encryption and decryption of each packet through the use of the ARC4 stream cipher, used to ensure the security of transmitted information. The maximum message transmission speed of 9.42 kBps is, however, very well suited to a wide variety of sensor-actuator systems and applications.

7.4.3 Service Call Round-Trip Latencies

Figure 7.6 depicts the latencies encountered during the invocation of synchronous *Call By* service functions in the collaboration tests, which are usually minimal functions intended to be immediately carried out to completion by the target module agent. This latency is defined as the time elapsed, or *round-trip interval*, between the transmission of the service call message to the target module agent and the reception of the associated return message from the target. Service calls that are asynchronous or *Call At* calls are not considered since such service calls are typically subject to lengthy and/or widely varying latencies, and will be utilized less often in practice than synchronous *Call By* service calls.

The results show that the return message reception latencies for synchronous *Call By* service function invocations are typically around 300 to 500 ms on average. However, it is also seen that the latencies may occasionally be on the order of seconds, as is the case with the channel reservation latencies encountered during execution of the MAC protocol. In comparison, the round-trip latencies encountered during the operation of the standard *Remote Procedure Call* (RPC) protocol on which the service call mechanism is based, and which is usually employed on high-speed wired and wireless networks, are typically

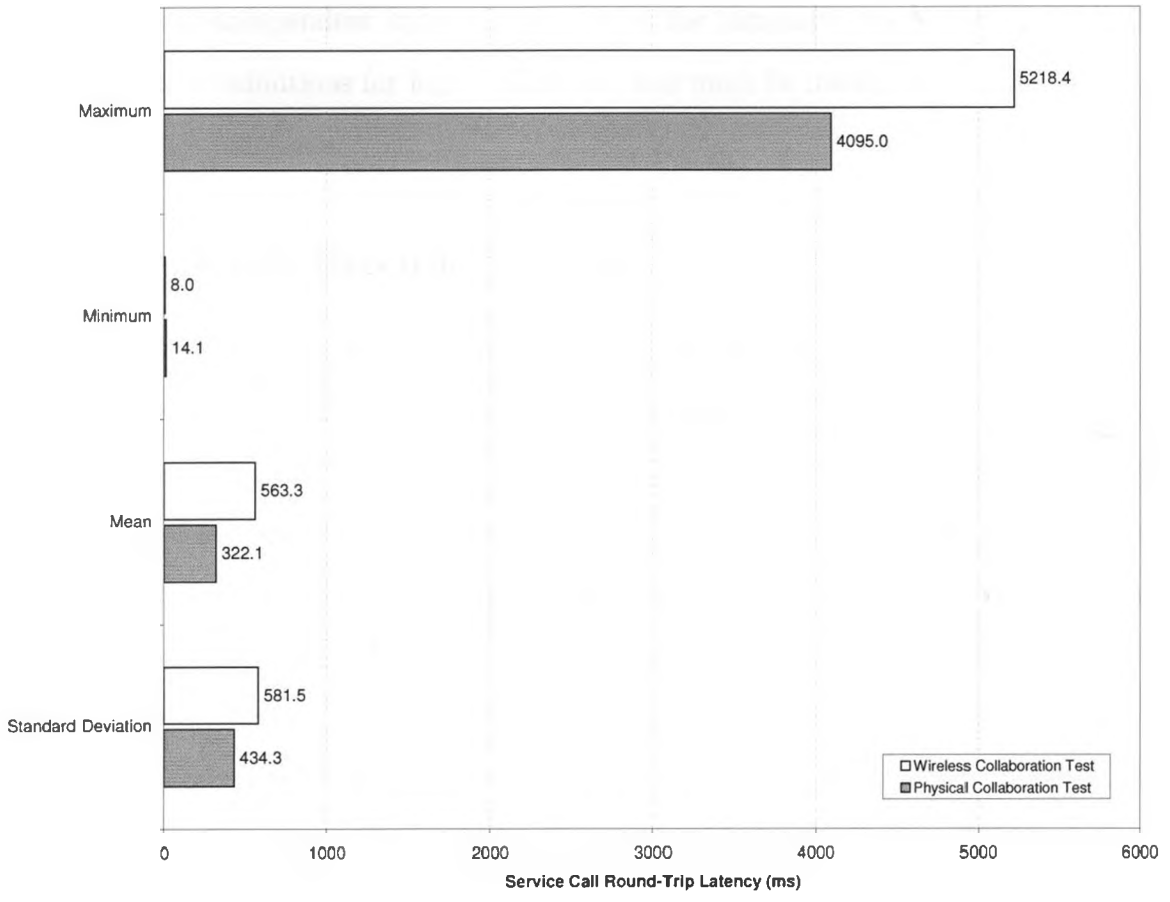


Figure 7.6: Service call round-trip latencies.

below 100 ms for small messages on the order of tens of kilobytes in size [70].

The relatively large service call message reception latencies typically encountered during the operation the software architecture can noticeably limit the responsiveness of a composite module, limiting the ability of the TIMs (in their current form) to be applied to real-time streaming applications. In addition to the previously outlined factors that impact channel reservation latencies and message transmission speeds (which in turn impact the speed of service function invocations), a major factor impacting return message reception latencies is the substantial overhead encountered in the interpretation and execution of Java bytecodes at the virtual machine layer (further outlined in Section 7.4.4).

These platform-independent bytecodes comprise the template classes that provide the service function definitions for logical modules, and must be interpreted during each call to these service functions.

7.4.4 Bytecode Execution Speeds

Figure 7.7 depicts the speeds at which the Java virtual machine bytecode instructions comprising logical module template classes are interpreted and executed at the virtual machine layer. Each bytecode interpretation speed reading is determined by dividing the number of instructions executed during the invocation of a particular template class service function by the length of time required to complete the call. As seen in the results, the mean bytecode execution speed attained during the wireless collaboration test of about 47 instructions/second is noticeably lower than the mean speed of about 266 instructions/second attained during the physical collaboration test; in fact, the maximum speed of about 91 instructions/second is also noticeably lower. During the physical collaboration test, much higher maximum speeds of up to about 886 instructions/second were realized.

This discrepancy may be attributed to the fact that during the wireless collaboration test, synchronous service calls are being issued multiple times per second by the primary module of the modular sensing system to the other constituent TIMs within the logical entity (to perform near real-time acquisition of sensor readings). Synchronous service calls result in the suspension of template class execution until the completion of the call, therefore potentially extending the execution time of the template class method from which the service call originated by a substantial amount. In the physical collaboration test, the primary module of the formed logical entity only issues synchronous service calls to the constituent TIMs within the logical entity upon a change in state of the system (the text being displayed), thus vastly reducing the performance impact of the

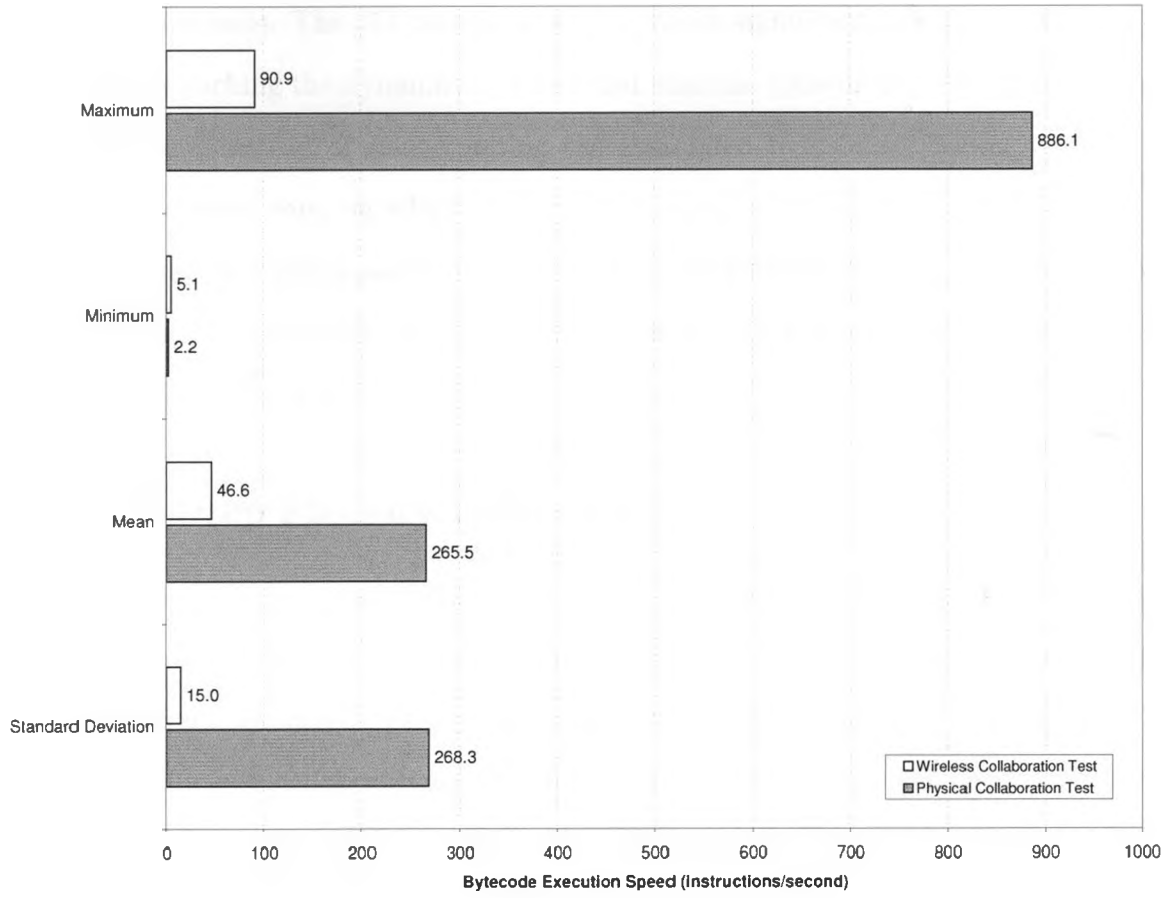


Figure 7.7: Virtual machine bytecode execution speeds.

calls. However, the intermittent occurrence of these synchronous service calls noticeably increased the variance of the bytecode interpretation speed readings observed during the physical collaboration test.

The attained bytecode interpretation speeds during the collaboration tests are considerably slower (on the order of thousands of times) than native machine code execution. The low processor clock speeds and limited memory available in the TIM hardware, as well many other resource-constrained embedded devices, are not conducive to popular acceleration techniques such as *dynamic recompilation*, also termed *just-in-time recompilation* (JIT) [71], in which newly encountered bytecodes are recompiled on the fly into

native machine code. The JIT recompilation approach significantly increases interpretation speed by caching the dynamically generated machine code in memory and executing it as necessary, instead of reinterpreting the associated bytecodes. Newer versions of the ARM processor core, on which the LPC2148 microcontroller present in the TIMs is based, overcome the performance limitations of purely interpretive Java virtual machines by supporting the execution of Java bytecodes directly in hardware, through the use of the *Jazelle* [71] architecture extension.

7.4.5 Startup Memory Utilization

Figure 7.8 depicts the amount of random access memory (RAM) utilized on the TIMs (from an available heap of 31,656 bytes, or approximately 31 kilobytes, integrated into the LPC2148 microcontroller) by the software architecture upon formation of the logical modules within each collaboration test before they begin execution. This memory utilization includes the representative structures for all the main background tasks (the network communication task, face communication task, and administrative shell task), the module message handler task associated with the local transducer on each TIM, as well as for the module message handler task associated with the local representation of the logical module of which the TIM is a member. Also included in the memory utilization is the template class and TEDS entries associated with the logical module.

The results clearly show that about 20 to 24 KB (kilobytes) on average, or about 65% to 77% of the available memory, is typically utilized upon logical module formation. The minimum, mean, and maximum memory utilization readings are shown to be relatively close in each collaboration test, but do deviate by small amounts. This may be attributed to packets and messages that arrive and are buffered during the initialization of the logical module at varying intervals, which introduce minor variations in the memory utilization detected. The consistently larger memory utilization by the physical collaboration test as

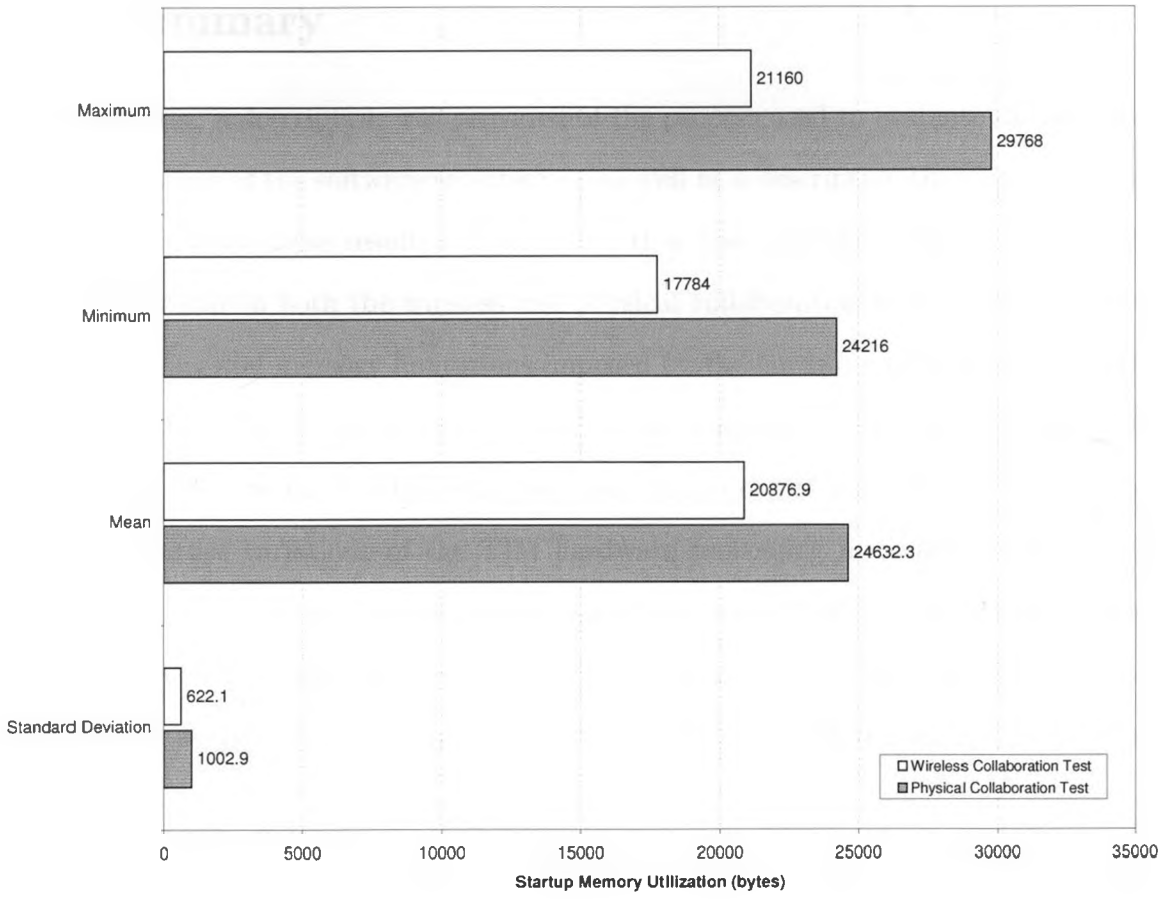


Figure 7.8: Startup memory utilization after logical module creation.

compared to the wireless collaboration test may be attributed to the `LCDMerge` template class utilized in the physical collaboration test, which is larger and more complex than the `ServoCon` template class utilized in the wireless collaboration test. Due to the fact that complex template classes, such as the `LCDMerge` template class, will often require on the order of 4 KB of memory or more just for representation purposes, scalability to greater than one or two logical module representations per TIM is very difficult. Nevertheless, the software architecture is designed to scale with increasing amounts of processing power and available memory, which is quickly becoming less cost prohibitive with improvements in semiconductor fabrication techniques.

7.5 Summary

In this chapter, a description was provided of the process used to evaluate the operation and performance of the software architecture as well as a description the results observed and an analysis of these results. It was seen that the software architecture exhibited correct behaviour in both the wireless and physical collaboration tests performed. However, processing and memory limitations imposed by the hardware present in the current implementation of the TIMs noticeably limited the responsiveness and scalability of the architecture. Due to the hardware-independent design of the software architecture, utilization of future variations of the TIM hardware possessing increased processing and memory resources will lead to immediate improvements in its performance. The following chapter will conclude the thesis and summarize the implementation of the software architecture, as well as provide recommendations for future enhancements of the software architecture.

Chapter 8

Conclusions

8.1 Concluding Remarks

This thesis presented a novel software architecture and knowledge representation scheme that facilitates the flexible, scalable, and reliable combination of heterogeneous modular sensor and actuator components. The ability to combine a sensor with an actuator allows the sensor to be augmented with motion capability, enabling the now *active* sensor to adapt to changing process requirements.

Each modular component provides a core sensing or actuation functionality and also possesses embedded knowledge of its capabilities, which may be communicated to other modules in its environment. This facilitates collaboration among a group of sensor and actuator modules, and enables all or a subset of the group to dynamically exhibit completely new behaviour. The proposed software architecture was implemented and evaluated using a prototype transducer module implementation in order to test its viability, and this work is a first step towards a highly adaptive architecture that will prove useful in applicable domains such as flexible inspection, mobile robotics, surveillance, and even space exploration.

In order to achieve the proposed research objective, the following features were considered for the design of the architectural framework, each of which were incorporated into the resulting design to varying degrees:

- **Heterogeneity** — Support the connection of modular sensor and actuator components of diverse types.
- **Autonomy** — Support the autonomous discovery and intelligent configuration of networked modules.
- **Pose/Geometry Determination** — Support the determination of the *pose* (position and orientation) of individual modules, and therefore the determination of the overall geometry of a group of connected modules.
- **Assumption of a Collective Identity** — Facilitate the assumption of a collective identity by a set of collaborating modules based on their capabilities and poses.
- **Process Distribution** — Support task splitting and distribution among a group of networked modules.
- **Resource Management** — Manage the hardware resources on each module in an efficient and straightforward manner.
- **Scalability** — Maintain reliable operation with an increasing number of connected modules.
- **Robustness** — Automatically adapt to the addition, removal, or failure of modules in real-time.

8.2 Thesis Summary

The software architecture described in this thesis is a distributed, layered architecture composed of six layers. The implementation of each layer is encapsulated from the layer above, to which it provides service. Utilizing a layered architecture enables each layer to be easily implemented, modified, and debugged independently of the others, while the distributed nature of the architecture mitigates against a single point of failure within a system of collaborating modules.

The first and lowest layer, described in Chapter 2, is the *module hardware* layer. This layer comprises the processing, memory, storage, communication and sensing/actuation resources needed by the software architecture to operate. These resources are provided by the modular sensor and actuator components for which the software architecture is designed, termed *Transducer Interface Modules* (TIMs) [22].

The second layer is the *real-time operating system (RTOS)/device drivers* layer, also described in Chapter 2. This layer provides straightforward management of TIM hardware resources and also provides an environment for the concurrent execution of independent *tasks*, which promotes a modular and easily maintained software architecture implementation. The device drivers present within this layer are the low-level software routines through which manipulation of the hardware resources present in TIMs occurs.

The third layer is the *communication layer*, described in Chapter 3. This layer serves as an interface to the wireless transceiver driver, and provides *logical link control* to transform the unreliable wireless medium into a reliable, connection-oriented medium; *medium access control* in order to prevent wireless channel access conflicts; a protocol for maintaining *time synchronization* between modules; and *wireless security* for transmitted packets facilitated by the *ARC4* [32] cryptographic stream cipher. Also provided by this layer is a novel wired protocol facilitating the transfer of data through the faces of connected TIMs for the purposes of pose and overall geometry determination.

The fourth layer is the *middleware layer*, outlined in Chapter 4. This layer defines the various architecture-specific *service functions* through which the heterogeneous or homogeneous member TIMs comprising a *logical module* may request services from and information about each other, thus facilitating distributed operation. These service functions define the *application programming interface* (API) for both physical and logical modules. Variable-length *messages* are used to transfer data between TIMs.

The fifth layer is the *virtual machine layer*, outlined in Chapter 5. This layer provides a environment for the execution of the platform-independent template algorithm classes that define the behaviour dynamically assumed by the collaborating TIMs comprising logical modules. The definition of platform-independent algorithms enables the behaviour of a group of collaborating TIMs to be specified in a manner that is completely decoupled from their underlying hardware architecture. The virtual machine itself is a custom lightweight implementation of Sun Microsystems' well documented and widely utilized *Java Virtual Machine* [23] designed specifically for this software architecture.

The sixth and final layer is the *composition layer*, outlined in Chapter 6. In this layer *logical module template TEDS specifications* as well as their associated *logical module template algorithm classes*, both unique to this software architecture, are loaded by a TIM from its non-volatile storage in response to modules it detects within its environment that satisfy at least one *role* definition within any of the specifications. If all of the roles within a particular template TEDS specification are matched, a logical module task and structure representation is created locally on each matching module. The template class associated with the template TEDS is then executed on each module in order to realize the defined behaviour of the logical entity through the collaboration of its members.

Evaluation results showed that the software architecture exhibited correct behaviour in both the wireless collaboration and physical collaboration scenarios considered. However, the responsiveness and scalability of the architecture was notably restricted as a

result of the processing and memory limitations imposed by the current TIM hardware implementation. Key latencies were encountered during channel reservation at the communication layer as well as during service call message round-trips at the middleware layer. Limited packet and message transmission speeds at the communication layer also impacted these latencies. The limited processing capability noticeably affected virtual machine bytecode interpretation performance, particularly since acceleration techniques such as dynamic recompilation and direct bytecode execution could not be leveraged with the current TIM hardware. The memory footprint upon initialization of a logical module at the composition layer was also noticeably large. In spite of these performance restrictions, the software architecture was designed to scale with increased processing and memory resources, and therefore the utilization of future variations of the TIM hardware possessing increased processing and memory resources will lead to tangible performance improvements.

8.3 Recommendations

Although the adaptive modular sensing system software architecture exhibited correct behaviour when implemented and evaluated on actual TIM hardware, a number of areas in the design and implementation of the software architecture may be improved, and new features may be added, in order to increase its applicability.

As discovered through the evaluation of the software architecture, processing and memory limitations imposed by the current TIM hardware implementation resulted in relatively large latencies being introduced into multiple layers of the software architecture. These latencies cumulatively act to reduce system performance, limiting its applicability to demanding scenarios where real-time performance is required. Utilizing a revision of the TIM hardware possessing a microcontroller based on more recent, high-speed versions

of the ARM processor, such as those in the ARM9 and ARM11 series, would result in substantially improved architecture performance. In addition, these microcontrollers also expose an interface providing direct access to their address bus, data bus, and chip select lines, facilitating the incorporation of vasts amounts of high-speed static random access memory (on the order of hundreds of megabytes) into newer TIM implementations. This would greatly increase the number of concurrent logical modules of which a TIM may be a collaborating member. Increased processing and memory resources would also enable very bandwidth-intensive sensors which regularly transfer large volumes of data, such as embedded cameras, to be associated with TIMs and utilized in a modular sensing system.

Greater sustained packet and message transmission speeds would also noticeably improve the performance of the overall software architecture, particularly in scenarios requiring real-time collaboration between member modules transferring substantial amounts of data, since the round-trip latency for every service call issued would be reduced. Moving channel reservation and switching logic into the firmware of the transceiver would also aid in reducing these latencies. A transceiver based on the 2.4 GHz short-range, frequency-hopping spread spectrum (FHSS) technology *Bluetooth* [72] would be well suited to achieving these goals. In addition, due to the pseudorandom frequency-hopping sequence utilized in switching between channels, data transmissions are indistinguishable from background noise without knowledge of the sequence. This provides an extra layer of data security to supplement its firmware-level encryption.

Throughout the software architecture, *critical sections* are extensively utilized to facilitate the atomic operation of timing-critical operations. Critical sections are implemented through the disabling of interrupts, which has the side effect of preventing the real-time operating system (RTOS) from performing context switches between concurrently executing tasks. However, critical sections are also utilized in a number of areas of the software architecture to provide atomic access to queues and other forms of data to con-

currently executing tasks competing for these resources. With frequent utilization, critical sections often result in greatly increased execution latencies. Although the TNKernel RTOS utilized within the software architecture supports constructs such as *mutexes* and *semaphores* that are designed to allow atomic access to shared data without preventing context switches, they were not utilized in order to avoid the overhead involved in their use on the microcontroller. In the TNKernel RTOS, each use of these constructs requires a switch to kernel code. On hardware possessing greater processing capability, mutexes and semaphores should be utilized to implement task-safe access to shared data.

To keep the implementation of the Java-based virtual machine within the software architecture lightweight, features of complete Java virtual machine implementations deemed not critical to logical module member collaboration, such as threads and automatic garbage collection, were not implemented due to the previously mentioned limited processing capability of the TIMs. However, these features are recognized as part of the standard definition of the Java language, and would likely be seen as a requirement by some system users. On more capable TIM hardware, these features should be provided in order to bring the capabilities of the implemented virtual machine in line with the complete Java virtual machine specification as defined by Sun Microsystems [67]. Alternatively, if the utilized TIM hardware possesses enough memory resources to parse, and possibly compile, logical module behaviour specified strictly in the form of a text-based script, interpreters for well-supported and well-documented scripting languages such as *Lua* [73] and *Python* [74] may be incorporated into the software architecture in place of the Java-based virtual machine. These languages are somewhat less flexible than a complete Java implementation, but the lack of a need for source code compilation may simplify template algorithm implementation, deployment, and debugging. The Lua interpreter is particularly conducive to being embedded within a larger, encompassing application.

The library of template TEDS specifications and processing algorithms stored in the

non-volatile memory local to each TIM within a modular sensing system is updated, without requiring user intervention, in an automatic, peer-to-peer fashion through the use of *join structures* (see Section 6.2.3) issued by *Join* service calls during the formation of logical modules. However, no mechanism is provided by which only the latest version of a particular template TEDS specification and its associated template class is transferred between TIMs before execution. A versioning scheme, possibly in the form of a new TEDS field, should be incorporated into TEDS specifications to ensure that a module with the latest version of a particular specification (and its associated template class if applicable) does not have its stored specification overwritten by older versions during peer-to-peer updates. In the presence of a connection to the administrative interface on a module possessing conflicting TEDS specification versions, the system user may be queried for conformation before the older TEDS specification is overwritten.

In summary, implementing and evaluating the software architecture proposed in this thesis has enabled numerous performance limitations to be exposed that would otherwise be difficult to discover. Addressing the latencies encountered at multiple layers within the software architecture stack as well as improving the speeds of message transmission and bytecode execution will result in a greatly improved adaptive modular sensing system architecture that will prove useful in its many applicable domains.

References

[1] G. E. Moore, "Cramming more components onto integrated circuits," in *Proceedings of the IEEE*, vol. 86, pp. 82–85, January 1998.

[2] T. Henderson, C. Hansen, and B. Bhanu, "A framework for distributed sensing and control," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI '85)*, (Los Angeles, CA, USA), pp. 1106–1109, August 1985.

[3] M. Dekhil and T. C. Henderson, "Instrumented sensor systems," in *Proceedings of the 1996 IEEE/SICE/RSJ International Conference on Multisensor Fusion and Integration for Intelligent Systems*, (Washington, DC, USA), pp. 193–200, IEEE, December 1996.

[4] L. Mottola and G. P. Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *Proceedings of the Second IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS 2006)*, (San Francisco, CA, USA), pp. 150–168, Springer-Verlag, June 2006.

[5] P. Ciciriello, L. Mottola, and G. P. Picco, "Building virtual sensors and actuators over logical neighborhoods," in *Proceedings of the International Workshop on Middleware for Sensor Networks (MidSens 2006)*, (Melbourne, Australia), pp. 19–24, Association for Computing Machinery, November 2006.

[6] K. Lee, "IEEE 1451: A standard in support of smart transducer networking," in *Proceedings of the 17th IEEE Instrumentation and Measurement Technology Conference*, vol. 2, (Baltimore, MD, USA), pp. 525–528, IEEE, May 2000.

[7] National Institute of Standards and Technology, "NIST IEEE-P1451 draft standard home page." [Online]. Available: `http://ieee1451.nist.gov/`, January 2008 [Accessed: April 16, 2008].

[8] National Instruments, "An overview of IEEE 1451.4 transducer electronic data sheets (TEDS)." [Online]. Available: `http://www.ni.com/teds/`, September 2006 [Accessed: April 16, 2008].

[9] Institute of Electrical and Electronics Engineers, "IEEE P1451.7D0.3: Draft standard for a smart transducer interface for sensors and actuators – RFID commu-

nication protocols and transducer electronic data sheet (TEDS) formats." [Online]. Available: http://www.autoid.org/sc31/wg4sg1/07/Nov/SG1_200711_149_IEEE_042_P1451_7D0_03_10_23_2007.doc, November 2007 [Accessed: April 16, 2008].

[10] J. L. Hill and D. E. Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, pp. 12–24, November 2002.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*, vol. 34, (Cambridge, MA, USA), pp. 93–104, ACM, December 2000.

[12] L. E. Holmquist, H.-W. Gellersen, G. Kortuem, A. Schmidt, M. Strohbach, S. Antifakos, F. Michahelles, B. Schiele, M. Beigl, and R. Mazé, "Building intelligent environments with Smart-Its," *IEEE Computer Graphics and Applications*, vol. 24, pp. 56–64, January 2004.

[13] S. Cotterell, R. Mannion, F. Vahid, and H. Hsieh, "eBlocks – an enabling technology for basic sensor based systems," in *Proceedings 2005 Fourth International Symposium on Information Processing in Sensor Networks*, (Los Angeles, CA, USA), pp. 422–427, IEEE, April 2005.

[14] S. Cotterell, K. Downey, and F. Vahid, "Applications and experiments with eBlocks – electronic blocks for basic sensor-based systems," in *Proceedings 2004 First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, (Santa Clara, CA, USA), pp. 7–15, IEEE, October 2004.

[15] T. D. Ngo and H. H. Lund, "Modular artefacts," in *Component-Oriented Approaches to Context-aware Computing (ECOOP 2004)*, (Oslo, Norway), June 2004.

[16] N. Edmonds, D. Stark, and J. Davis, "MASS: Modular architecture for sensor systems," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, (Los Angeles, CA, USA), pp. 393–397, IEEE, April 2005.

[17] A. S. Tanenbaum, *Computer Networks*. Upper Saddle River, NJ, USA: Prentice Hall, 4th ed., 2003.

[18] Bug Labs, "Bug labs: Products." [Online]. Available: http://www.buglabs.net/products/, 2008 [Accessed: April 16, 2008].

[19] M. P. Weller, E. Y.-L. Do, and M. D. Gross, "Posey: Embedding computation in a poseable hub and strut construction kit for undirected play," in *2nd ACM Conference on Tangible and Embedded Interaction*, (Bonn, Germany), February 2008.

[20] NXP Semiconductors, "LPC2141, LPC2142, LPC2144, LPC2146, and LPC2148 device highlight." [Online]. Available: http://www.standardics.nxp.com/products/lpc2000/lpc214x/, 2007 [Accessed: June 30, 2007].

[21] Nordic Semiconductor, "nRF24L01 single chip 2.4GHz transceiver product specification." [Online]. Available: http://www.nordicsemi.com/files/Product/data_sheet/nRF24L01_Product_Specification_v2_0.pdf, July 2007 [Accessed: May 15, 2008].

[22] A. Jain and M. D. Naish, "Building blocks for adaptive modular sensing systems," in *Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics*, (Montréal, QC, Canada), pp. 184–189, IEEE, October 2007.

[23] Sun Microsystems Inc., "Java technology." [Online]. Available: http://java.sun.com/, 2007 [Accessed: March 4, 2007].

[24] R. Barry, "FreeRTOS real-time operating system." [Online]. Available: http://www.freertos.org/, 2007 [Accessed: March 4, 2007].

[25] Y. Tiomkin, "TNKernel real-time kernel." [Online]. Available: http://www.tnkernel.com/, 2006 [Accessed: June 29, 2007].

[26] Microsoft Corporation, "FAT technical reference." [Online]. Available: http://technet.microsoft.com/en-us/library/cc758586.aspx, March 2003 [Accessed: August 13, 2008].

[27] S. Tweedie, "EXT3, journaling filesystem." [Online]. Available: http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000 [Accessed: February 28, 2008].

[28] Microsoft Corporation, "NTFS technical reference." [Online]. Available: http://technet.microsoft.com/en-us/library/cc758691.aspx, March 2003 [Accessed: August 13, 2008].

[29] Electronic Lives Manufacturing, "FAT file system module." [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html, 2007 [Accessed: January 28, 2008].

[30] L. Ysboodt and M. De Nil, "EFSL embedded filesystems library." [Online]. Available: http://efsl.be/, 2005 [Accessed: January 28, 2008].

[31] Microsoft Corporation, "Naming a file." [Online]. Available: http://msdn.microsoft.com/en-us/library/aa365247.aspx, June 2008 [Accessed: July 7, 2008].

[32] K. Kaukonen and R. Thayer, "A stream cipher encryption algorithm: Arcfour." [Online]. Available: http://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt, July 1999 [Accessed: May 13, 2008].

[33] The World Wide Web Consortium, "Extensible markup language (XML)." [Online]. Available: http://www.w3.org/XML/, January 2008 [Accessed: May 13, 2008].

[34] V. Bharghavan, A. Demers, S. Shenker, and L. Zhang, "MACAW: A media access protocol for wireless LANs," in *Proceedings ACM SIGCOMM '94: Conference on Communications Architectures, Protocols and Applications*, (London, UK), pp. 212–225, August 1994.

[35] P. Karn, "MACA — a new channel access method for packet radio," in *Proceedings of the 9th ARRL/CRRL Amateur Radio Computer Networking Conference*, (London, ON, Canada), pp. 134–140, September 1990.

[36] D. L. Mills, "Simple network time protocol (SNTP) version 4 for IPv4, IPv6 and OSI." [Online]. Available: http://www.faqs.org/ftp/rfc/pdf/rfc4330.txt.pdf, January 2006 [Accessed: March 4, 2007].

[37] D. L. Mills, "Network time protocol (version 3) specification, implementation and analysis." [Online]. Available: http://www.faqs.org/ftp/rfc/rfc1305.pdf, March 1992 [Accessed: March 4, 2007].

[38] T. Karygiannis and L. Owens, "Wireless network security: 802.11, Bluetooth and handheld devices (NIST special publication 800-48)." [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-48/NIST_SP_800-48.pdf, November 2002 [Accessed: May 15, 2008].

[39] B. Schneier, "Description of a new variable-length key, 64-bit block cipher (Blowfish)," in *Fast Software Encryption, Cambridge Security Workshop Proceedings*, (Cambridge, UK), pp. 191–204, Springer-Verlag, December 1993.

[40] National Institute of Standards and Technology, "Federal information processing standards publication 197: Announcing the Advanced Encryption Standard (AES)." [Online]. Available: http://www.csrc.nist.gov/publications/fips/fips197/fips-197.pdf, November 2001 [Accessed: May 13, 2008].

[41] The Wi-Fi Alliance, "Wi-Fi Protected Access: Strong, standards-based, interoperable security for todays Wi-Fi networks." [Online]. Available: http://www.wi-fi.org/files/wp_8_WPASecurity_4-29-03.pdf, April 2003 [Accessed: May 13, 2008].

[42] C. E. Strangio, "The RS232 standard: A tutorial with signal names and definitions." [Online]. Available: http://www.camiresearch.com/Data_Com_Basics/RS232_standard.html, 2006 [Accessed: May 17, 2008].

[43] J. Hurwitz, "Sorting out middleware." [Online]. Available: http://www.dbmsmag.com/9801d04.html, January 1998 [Accessed: February 12, 2008].

[44] R. Srinivasan, "RPC: Remote procedure call protocol specification version 2." [Online]. Available: http://www.faqs.org/ftp/rfc/pdf/rfc1831.txt.pdf, August 1995 [Accessed: February 12, 2008].

[45] Carnegie Mellon Software Engineering Institute, "Object request broker." [Online]. Available: http://www.sei.cmu.edu/str/descriptions/orb_body.html, January 2007 [Accessed: February 12, 2008].

[46] Object Management Group, "Common object request broker architecture: Core specification." [Online]. Available: http://www.omg.org/docs/formal/04-03-12.pdf, March 2004 [Accessed: February 12, 2008].

[47] P. E. Chung, Y. Huang, S. Y. D. Liang, J. C. Shih, C.-Y. Wang, and Y.-M. Wang, "DCOM and CORBA side by side, step by step, and layer by layer." [Online]. Available: http://research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html, September 1997 [Accessed: February 12, 2008].

[48] Sun Microsystems Inc., "Java remote method invocation – distributed computing for Java." [Online]. Available: http://java.sun.com/javase/technologies/core/basic/rmi/whitepaper/index.jsp, 2008 [Accessed: February 12, 2008].

[49] K. North, "Understanding multidatabase APIs and ODBC." [Online]. Available: http://www.dbmsmag.com/9403d13.html, June 1994 [Accessed: February 12, 2008].

[50] Sun Microsystems Inc., "JDBC overview." [Online]. Available: http://java.sun.com/products/jdbc/overview.html, 2008 [Accessed: February 12, 2008].

[51] R. Balani, S. Han, R. Kumar, I. Tsigkogiannis, and M. Srivastava, "Multi-level software reconfiguration for sensor networks," in *Proceedings of the 6th ACM and IEEE International Conference on Embedded Software (EMSOFT 2006)*, (Seoul, South Korea), pp. 112–121, Association for Computing Machinery, October 2006.

[52] J. Jeong, "Node-level representation and system support for network programming," tech. rep., University of California, Berkeley, December 2003.

[53] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems (SenSys '04)*, (Baltimore, MD, USA), pp. 81–94, ACM, November 2004.

[54] S. S. Kulkarni and L. Wang, "MNP: Multihop network reprogramming service for sensor networks," in *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops (ICDCS 2005)*, (Columbus, OH, USA), pp. 7–16, IEEE, June 2005.

[55] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," tech. rep., University of California, Los Angeles, 2003.

[56] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki: A lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN 2004)*, (Tampa, FL, USA), pp. 455–462, IEEE, November 2004.

[57] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the Third International Conference on Mobile Systems, Applications, and Services (MobiSys 2005)*, (Seattle, WA, USA), pp. 163–176, USENIX Association, June 2005.

[58] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl, "Virtual machine showdown: Stack versus registers," in *Proceedings of the First ACM/USENIX International Conference on Virual Execution Environments (VEE '05)*, (Chicago, IL, USA), pp. 153–163, ACM, June 2005.

[59] E. Meijer and J. Gough, "Technical overview of the common language runtime," tech. rep., Microsoft, 2000.

[60] P. Levis and D. Culler, "Maté: A tiny virtual machine for sensor networks," in *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, vol. 37, (San Jose, CA, USA), pp. 85–95, ACM, October 2002.

[61] P. Levis, D. Gay, and D. Culler, "Active sensor networks," in *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI '05)*, (Boston, MA, USA), pp. 343–356, May 2005.

[62] P. Stanley-Marbell and L. Iftode, "Scylla: A smart virtual machine for mobile embedded systems," in *Proceedings of the Third IEEE Workshop on Mobile Computing Systems and Applications*, (Los Alamitos, CA, USA), pp. 41–50, IEEE, December 2000.

[63] J. Koshy and R. Pandey, "VM*: Synthesizing scalable runtime environments for sensor networks," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems (SenSys '05)*, (San Diego, California, USA), pp. 243–254, ACM, November 2005.

[64] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, pp. 122–173, March 2005.

[65] Codehaus Foundation, "Groovy: An agile dynamic language for the Java platform." [Online]. Available: `http://groovy.codehaus.org/`, 2006 [Accessed: February 27, 2008].

[66] Python Software Foundation, "The Jython project." [Online]. Available: http://www.jython.org/, 2007 [Accessed: February 27, 2008].

[67] T. Lindholm and F. Yellin, "The Java™ virtual machine specification, second edition." [Online]. Available: http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html, 1999 [Accessed: February 27, 2008].

[68] F. Dunn and I. Parberry, *3D Math Primer for Graphics and Game Development.* Plano, TX, USA: Wordware Publishing, Inc., 2002.

[69] L. Yun, L. Ke-Ping, Z. Wei-Liang, and W. Chong-Gang, "Analyzing the channel access delay of IEEE 802.11 DCF," in *Proceedings of the 2005 IEEE Global Telecommunications Conference (GLOBECOM '05)*, vol. 5, (St. Louis, MO, USA), pp. 2997–3001, IEEE, November 2005.

[70] A. M. Khandker, P. Honeyman, and T. J. Teorey, "Performance of DCE RPC," in *Proceedings of the Second International Workshop on Services in Distributed and Networked Environments*, (Whistler, BC, Canada), pp. 2–10, IEEE, June 1995.

[71] C. Porthouse, "High-performance Java on embedded devices." [Online]. Available: http://www.arm.com/pdfs/JazelleDBX_WhitePaper_2007v1p1.pdf, October 2005 [Accessed: July 5, 2007].

[72] Bluetooth Special Interest Group Inc., "How Bluetooth technology works." [Online]. Available: http://www.bluetooth.com/Bluetooth/Technology/Works/, 2008 [Accessed: July 9, 2008].

[73] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho, "Lua — an extensible extension language," *Software: Practice and Experience*, vol. 26, pp. 635–652, June 1996.

[74] Python Software Foundation, "The Python programming language." [Online]. Available: http://www.python.org/, 2006 [Accessed: March 14, 2007].

Appendix A

Standard Class Library

A.1 Introduction

In this appendix, the standard class library provided for use within the software architecture is depicted. Each class package within the standard class library contains classes that provide a higher-level interface to various native methods defined within the software architecture. Further details may be found in Section 5.4.

A.2 Package java.lang

A.2.1 java.lang.Math

```
// ------------------------------
// java.lang.Math class
// Copyright (c) 2008 Andrew Lyle
// ------------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
```

```
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package java.lang;

public final class Math {
  public static final double E  = 2.7182818284590452354;
  public static final double PI = 3.14159265358979323846;

  private Math() {}
  public static native double sin(double d);
  public static native double cos(double d);
  public static native double tan(double d);
  public static native double asin(double d);
  public static native double acos(double d);
  public static native double atan(double d);
  public static native double exp(double a);
  public static native double log(double a);
  public static native double ceil(double a);
  public static native double floor(double a);
  public static native double rint(double a);
  public static native double atan2(double y, double x);
  public static native double pow(double a, double b);
  public static native int round(float a);
  public static native long round(double a);
  public static native double random();
  public static native int abs(int a);
  public static native long abs(long a);
  public static native float abs(float a);
  public static native double abs(double a);
  public static native int max(int a, int b);
  public static native long max(long a, long b);
  public static native float max(float a, float b);
  public static native double max(double a, double b);
  public static native int min(int a, int b);
  public static native long min(long a, long b);
  public static native float min(float a, float b);
  public static native double min(double a, double b);
```

```
  public static native double sinh(double x);
  public static native double cosh(double x);
  public static native double tanh(double x);
}
```

A.2.2 java.lang.String

```
// -----------------------------
// java.lang.String class
// Copyright (c) 2008 Andrew Lyle
// -----------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package java.lang;

public final class String {
  private String() {}
  public native char charAt(int index);
  public native boolean endsWith(String suffix);
  public native boolean equals(Object anObject);
  public native boolean equalsIgnoreCase(String anotherString);
  public native int length();
  public native boolean startsWith(String prefix);
  public native String substring(int beginIndex, int endIndex);
  public native String toLowerCase();
  public native String toUpperCase();
}
```

A.3 Package amss.system

A.3.1 amss.system.AMSS

```
// ----------------------------
// amss.system.AMSS class
// Copyright (c) 2008 Andrew Lyle
// ----------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package amss.system;

public final class AMSS {
  // sleep constants
  public static final long TICK_1MS = 1;
  public static final long TICK_1S  = 1000 * TICK_1MS;
  public static final long TICK_1M  = 60 * TICK_1S;

  // clock constants
  public static final long TIME_1US = 1;
  public static final long TIME_1MS = 1000 * TIME_1US;
  public static final long TIME_1S  = 1000 * TIME_1MS;
  public static final long TIME_1M  = 60 * TIME_1S;

  private AMSS() {}
  public static native void   gc(Object obj);
  public static native void   enterAtomic();
  public static native void   leaveAtomic();
  public static native long   getClock();
  public static native void   sleep(long millis);
```

```
  public static native int    readAsync();
  public static native int    read();
  public static native String readString(int maxlen);
  public static native int    readInt(int base);
  public static native long   readLong(int base);
  public static native double readDouble(int base);
  public static native int    intValue(String s, int base);
  public static native float  floatValue(String s, int base);

  public static native void print(String s);
  public static native void print(boolean b);
  public static native void print(char c);
  public static native void print(int i);
  public static native void print(long l);
  public static native void print(float f);
  public static native void print(double d);
  public static native void println(String s);
  public static native void println(boolean b);
  public static native void println(char c);
  public static native void println(int i);
  public static native void println(long l);
  public static native void println(float f);
  public static native void println(double d);
  public static native void println();
}
```

A.3.2 amss.system.Message

```
// -----------------------------
// amss.system.Message class
// Copyright (c) 2008 Andrew Lyle
// -----------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
```

```
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package amss.system;

public final class Message {
  // message type constants
  public static final long NO_ADDRESS  = 0;
  public static final long NO_DEADLINE = 0x7fffffffffffffffL;
  public static final int  CALLAT      = 1;
  public static final int  CALLBY      = 2;
  public static final int  RETURN      = 3;

  // service function constants
  public static final int GET        = 1;
  public static final int SET        = 2;
  public static final int APPEND     = 3;
  public static final int RESET      = 4;
  public static final int GETTEDS    = 5;
  public static final int GETPOSE    = 6;
  public static final int UPDATEPOSE = 7;
  public static final int LOCK       = 8;
  public static final int UNLOCK     = 9;
  public static final int JOIN       = 10;

  // parameter types
  public static final int NO_PARAM = 0;
  public static final int INT8     = (1<<0);
  public static final int INT16    = (1<<1);
  public static final int INT32    = (1<<2);
  public static final int INT64    = (1<<3);
  public static final int UINT8    = (1<<4);
  public static final int UINT16   = (1<<5);
  public static final int UINT32   = (1<<6);
  public static final int UINT64   = (1<<7);
  public static final int FLOAT    = (1<<8);
  public static final int DOUBLE   = (1<<9);
  public static final int STATUS   = (1<<10);
  public static final int STRING   = (1<<11);
  public static final int OBJECT   = (1<<12);

  // status constants
```

```
public static final int SUCCESS = 1;
public static final int ERROR = 2;
public static final int MISSED_DEADLINE = 3;
public static final int INVALID_PARAMETER = 4;
public static final int LOCKED = 5;
public static final int NOT_ALLOWED = 6;

// core methods
private Message() {}
public static native Message create(long src, long dst,
  long deadline, long timestamp, int type, int srvfunc,
  int partype, int parw, int parh);
public static native Message clone(long src, long dst,
  long deadline, long timestamp, int type, int srvfunc,
  Message msg);
public static native Message enclose(long src, long dst,
  long deadline, long timestamp, int type, int srvfunc,
  Message msg);
public static native Message createReturn(Message mcall,
  int partype, int parw, int parh);
public static native Message createReturnStatus(Message mcall,
  int status);
public static native boolean enqueueMessageOut(Message m);

// property methods
public native long source();
public native long destination();
public native long deadline();
public native long timestamp();
public native int  type();
public native int  serviceFunction();
public native int  serviceId();
public native int  parameterType();
public native int  parameterWidth();
public native int  parameterHeight();

// get/set methods
public native char    getInt8(int y, int x);
public native short   getInt16(int y, int x);
public native int     getInt32(int y, int x);
public native long    getInt64(int y, int x);
public native char    getUint8(int y, int x);
public native short   getUint16(int y, int x);
```

```
  public native int     getUint32(int y, int x);
  public native long    getUint64(int y, int x);
  public native float   getFloat(int y, int x);
  public native double  getDouble(int y, int x);
  public native int     getStatus(int y, int x);
  public native String  getString(int y);
  public native Message getMessage();
  public native Object  getObject();
  public native void    setInt8(int y, int x, char val);
  public native void    setInt16(int y, int x, short val);
  public native void    setInt32(int y, int x, int val);
  public native void    setInt64(int y, int x, long val);
  public native void    setUint8(int y, int x, char val);
  public native void    setUint16(int y, int x, short val);
  public native void    setUint32(int y, int x, int val);
  public native void    setUint64(int y, int x, long val);
  public native void    setFloat(int y, int x, float val);
  public native void    setDouble(int y, int x, double val);
  public native void    setStatus(int y, int x, int val);
  public native void    setString(int y, String val);
}
```

A.3.3 amss.system.Module

```
// ------------------------------
// amss.system.Module class
// Copyright (c) 2008 Andrew Lyle
// ------------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

```
package amss.system;

public final class Module {
  // core functions
  private Module() {}
  public static native Pose getLocalPose();

  // handler functions
  public native boolean isRunning();
  public native boolean isPrimary();
  public native boolean isLocked(Message m);
  public native Message nextServiceCall();
  public native void    secondaryHandler(Message call);
  public native void    statusCheck();
  public native int     roleCount();
  public native int     roleEnvironmentCount(int role);
  public native Message serviceCall(Message m, int role,
    int envindex);
  public native int     serviceCallAsync(Message m, int role,
    int envindex);
  public native Message getReturn(int serviceid);

  // logical teds get/set functions
  public native char    getTedsChar(String propname);
  public native int     getTedsInt(String propname, int base);
  public native float   getTedsFloat(String propname, int base);
  public native String  getTedsString(String propname);
  public native boolean setTedsChar(String propname, char val);
  public native boolean setTedsInt(String propname, int val);
  public native boolean setTedsFloat(String propname, float val);
  public native boolean setTedsString(String propname, String val);
}
```

A.3.4 amss.system.Pose

```
// ------------------------------
// amss.system.Pose class
// Copyright (c) 2008 Andrew Lyle
// ------------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
```

```
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package amss.system;

public final class Pose {
  private Pose() {}
  public native Vector3D getTransducerPosition();
  public native Vector3D getTransducerFaceNormal();
  public native Vector3D getTransducerFaceNorth();
  public native Vector3D getTransducerFaceEast();
  public native Vector3D getTransducerSeparation(Pose pose);
}
```

A.3.5 amss.system.Vector3D

```
// ------------------------------
// amss.system.Vector3D class
// Copyright (c) 2008 Andrew Lyle
// ------------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.
```

```
package amss.system;

public final class Vector3D {
  private Vector3D() {}
  public native float    getI();
  public native float    getJ();
  public native float    getK();
  public native float    magnitude();
  public native Vector3D normalize();
  public native Vector3D add(Vector3D v);
  public native Vector3D subtract(Vector3D v);
  public native float    dot(Vector3D v);
  public native Vector3D cross(Vector3D v);
  public native float    angle(Vector3D v);
  public native boolean  parallel(Vector3D v);
  public native boolean  antiparallel(Vector3D v);
  public native boolean  perpendicular(Vector3D v);
}
```

Appendix B

Architecture Evaluation Data

B.1 Introduction

In this appendix, the module TEDS, template TEDS, and template algorithm classes utilized for the purposes of evaluating the functionality and performance of the software architecture as described in Chapter 7 are depicted.

B.2 Module TEDS

B.2.1 accel.mod

```
# Accelerometer Module TEDS File
# ------------------------------

# AMSS TEDS
# ---------
ModuleAddress         2000000000000001
ModuleType            1   # Sensor
ModuleClass           10  # Acceleration
ModuleDataType        100 # Float
ModuleDataTypeWidth   3
ModuleDataTypeHeight  1
```

```
PrimaryHandlerName   accel

# Basic TEDS
# ----------
ManufacturerID       1
ModelNumber          1
VersionLetter        A
VersionNumber        1
SerialNumber         1

# Standard Template TEDS
# ----------------------
TemplateID           30     # Voltage Template
ElecSigType          0      # Voltage Sensor
MapMeth              0      # Linear
SelectExcitation     1      # Include Excitation
ExciteAmplNom        3.3    # Volts
ExciteAmplMin        3.3    # Volts
ExciteAmplMax        3.3    # Volts
ExciteType           0      # DC
ExciteCurrentDraw    0.0006 # Amps
CalDate              1
CalInitials          NUL
CalPeriod            365    # Days
MeasID               1
```

B.2.2 lcd.mod

```
# LCD Module TEDS File
# --------------------

# AMSS TEDS
# ---------
ModuleAddress        1000000000000005 # and 2000000000000005
ModuleType           2   # Actuator
ModuleClass          4   # Text
ModuleDataType       800 # String
ModuleDataTypeWidth  16
ModuleDataTypeHeight 4
PrimaryHandlerName   lcd
```

```
# Basic TEDS
# ----------
ManufacturerID       1
ModelNumber          1
VersionLetter        A
VersionNumber        1
SerialNumber         1

# Standard Template TEDS
# ----------------------
MapMeth              0      # Linear
SelectExcitation     1      # Include Excitation
ExciteAmplNom        3.3    # Volts
ExciteAmplMin        3.3    # Volts
ExciteAmplMax        3.3    # Volts
ExciteType           0      # DC
ExciteCurrentDraw    0.003  # Amps
CalDate              1
CalInitials          NUL
CalPeriod            365    # Days
MeasID               1
```

B.2.3 ldr.mod

```
# LDR Module TEDS File
# --------------------

# AMSS TEDS
# ---------
ModuleAddress        3000000000000006
ModuleType           1   # Sensor
ModuleClass          8   # Voltage
ModuleDataType       100 # Float
ModuleDataTypeWidth  1
ModuleDataTypeHeight 1
PrimaryHandlerName   ldr

# Basic TEDS
# ----------
ManufacturerID       1
ModelNumber          1
```

```
VersionLetter        A
VersionNumber        1
SerialNumber         1

# Standard Template TEDS
# ----------------------
TemplateID           30      # Voltage Template
ElecSigType          0       # Voltage Sensor
MapMeth              0       # Linear
SelectExcitation     1       # Include Excitation
ExciteAmplNom        3.3     # Volts
ExciteAmplMin        3.3     # Volts
ExciteAmplMax        3.3     # Volts
ExciteType           0       # DC
ExciteCurrentDraw    0.00033 # Amps
CalDate              1
CalInitials          NUL
CalPeriod            365     # Days
MeasID               1
```

B.2.4 servo.mod

```
# Servo Module TEDS File
# ----------------------

# AMSS TEDS
# ---------
ModuleAddress        1000000000000007
ModuleType           2   # Actuator
ModuleClass          20  # Rotation
ModuleDataType       100 # Float
ModuleDataTypeWidth  1
ModuleDataTypeHeight 1
PrimaryHandlerName   servo

# Basic TEDS
# ----------
ManufacturerID       1
ModelNumber          1
VersionLetter        A
VersionNumber        1
```

```
SerialNumber          1

# Standard Template TEDS
# ----------------------
MapMeth               0   # Linear
SelectExcitation      1   # Include Excitation
ExciteAmplNom         3.3 # Volts
ExciteAmplMin         3.3 # Volts
ExciteAmplMax         3.3 # Volts
ExciteType            0   # DC
ExciteCurrentDraw     1.0 # Amps
CalDate               1
CalInitials           NUL
CalPeriod             365 # Days
MeasID                1
```

B.3 Template TEDS

B.3.1 LCDMerge.mod

```
# LCD Merge Template TEDS File
# ----------------------------

# AMSS TEDS
# ---------
ModuleType                2   # Actuator
ModuleClass               4   # Text
ModuleDataType            800 # String
ModuleDataTypeWidth       1
ModuleDataTypeHeight      1

# Basic TEDS
# ----------
ManufacturerID            1
ModelNumber               1
VersionLetter             A
VersionNumber             1
SerialNumber              1

# Standard Template TEDS
```

```
# ----------------------
MapMeth                   0     # Linear
SelectExcitation          1     # Include Excitation
ExciteAmplNom             3.3   # Volts
ExciteAmplMin             3.3   # Volts
ExciteAmplMax             3.3   # Volts
ExciteType                0     # DC
ExciteCurrentDraw         0.006 # Amps
CalDate                   1
CalInitials               NUL
CalPeriod                 365   # Days
MeasID                    1

# Roles
# -----
Role                      1
RoleAssignmentLimit       >=1
RoleConnectionType        3   # Local/Physical
RoleModuleType            2   # Actuator
RoleModuleClass           4   # Text
RoleModuleDataType        800 # String
RoleModuleDataTypeWidth   >=1
RoleModuleDataTypeHeight  >=1
```

B.3.2 ServoCon.mod

```
# Servo Control Template TEDS File
# --------------------------------

# AMSS TEDS
# ---------
ModuleType                2   # Actuator
ModuleClass               20  # Rotation
ModuleDataType            100 # Float
ModuleDataTypeWidth       1
ModuleDataTypeHeight      1

# Basic TEDS
# ----------
ManufacturerID            1
ModelNumber               1
```

```
VersionLetter            A
VersionNumber            1
SerialNumber             1

# Standard Template TEDS
# ----------------------
MapMeth                  0   # Linear
SelectExcitation         1   # Include Excitation
ExciteAmplNom            3.3 # Volts
ExciteAmplMin            3.3 # Volts
ExciteAmplMax            3.3 # Volts
ExciteType               0   # DC
ExciteCurrentDraw        1.0 # Amps
CalDate                  1
CalInitials              NUL
CalPeriod                365 # Days
MeasID                   1

# Roles
# -----
Role                     1
RoleAssignmentLimit      >=1
RoleConnectionType       7   # Any
RoleModuleType           1   # Sensor
RoleModuleClass          18  # Acceleration/Voltage
RoleModuleDataType       100 # Float
RoleModuleDataTypeWidth  >=1
RoleModuleDataTypeHeight ==1

Role                     2
RoleAssignmentLimit      >=1
RoleConnectionType       7   # Any
RoleModuleType           2   # Actuator
RoleModuleClass          20  # Rotation
RoleModuleDataType       100 # Float
RoleModuleDataTypeWidth  ==1
RoleModuleDataTypeHeight ==1
```

B.4 Template Algorithm Classes

B.4.1 amss.algo.LCDMerge

```
// ----------------------------
// amss.algo.LCDMerge class
// Copyright (c) 2008 Andrew Lyle
// ----------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package amss.algo;
import amss.system.*;

public final class LCDMerge {
  public static void main(Module mod) {
    while(mod.isRunning()) {
      Message mcall = mod.nextServiceCall();

      if(mcall != null && !primaryHandler(mod, mcall))
        mod.secondaryHandler(mcall);

      mod.statusCheck();
    }
  }

  public static boolean primaryHandler(Module mod, Message mcall) {
    boolean handled = true;
    Message mret = null;
    int srvfunc = mcall.serviceFunction();
```

```
    if(srvfunc == Message.GET)
      mret = Message.createReturnStatus(mcall, Message.NOT_ALLOWED);
    else if(srvfunc == Message.SET) {
      if(mod.isLocked(mcall))
        mret = Message.createReturnStatus(mcall, Message.LOCKED);
      else
        mret = lcdMergeSet(mod, mcall);
    }
    else if(srvfunc == Message.APPEND) {
      if(mod.isLocked(mcall))
        mret = Message.createReturnStatus(mcall, Message.LOCKED);
      else
        mret = Message.createReturnStatus(mcall, Message.NOT_ALLOWED);
    }
    else if(srvfunc == Message.RESET) {
      if(mod.isLocked(mcall))
        mret = Message.createReturnStatus(mcall, Message.LOCKED);
      else
        mret = Message.createReturnStatus(mcall, Message.NOT_ALLOWED);
    }
    else
      handled = false;

    if(handled) {
      AMSS.gc(mcall);

      if(mret != null)
        Message.enqueueMessageOut(mret);
    }

    return handled;
  }

  public static Message lcdMergeSet(Module mod, Message mcall) {
    if(mod.roleEnvironmentCount(1) >= 2) {
      // get pose for first LCD
      Message msg1 = Message.create(Message.NO_ADDRESS,
        Message.NO_ADDRESS, Message.NO_DEADLINE, AMSS.getClock(),
        Message.CALLBY, Message.GETPOSE, Message.NO_PARAM, 0, 0);
      Message mret1 = mod.serviceCall(msg1, 1, 0);
      Pose mpose1 = null;

      if(mret1 != null)
```

```
  mpose1 = (Pose)mret1.getObject();

// get pose for second LCD
Message msg2 = Message.create(Message.NO_ADDRESS,
  Message.NO_ADDRESS, Message.NO_DEADLINE, AMSS.getClock(),
  Message.CALLBY, Message.GETPOSE, Message.NO_PARAM, 0, 0);
Message mret2 = mod.serviceCall(msg2, 1, 1);
Pose mpose2 = null;

if(mret2 != null)
  mpose2 = (Pose)mret2.getObject();

// get pose vectors
Vector3D vn1 = mpose1.getTransducerFaceNorth();
Vector3D ve1 = mpose1.getTransducerFaceEast();
Vector3D vn2 = mpose2.getTransducerFaceNorth();
Vector3D ve2 = mpose2.getTransducerFaceEast();

if(vn1 != null && vn2 != null && ve1 != null && ve2 != null &&
   vn1.parallel(vn2) && ve1.parallel(ve2)) {
  Vector3D vs1 = mpose1.getTransducerSeparation(mpose2);

  if(vs1 == null)
    AMSS.print("LCDMerge: Could not determine LCD separation.");
  else {
    boolean stacked = false;
    boolean firstIsPrimary = false;
    boolean inited = true;

    // determine geometry
    if(vs1.parallel(vn1)) {
      stacked = true;
      firstIsPrimary = true;
    }
    else if(vs1.antiparallel(vn1)) {
      stacked = true;
      firstIsPrimary = false;
    }
    else if(vs1.antiparallel(ve1)) {
      stacked = false;
      firstIsPrimary = true;
    }
    else if(vs1.parallel(ve1)) {
```

```
    stacked = false;
    firstIsPrimary = false;
  }
  else
    inited = false;

  if(inited) {
    // print debug messages
    AMSS.print("LCDMerge: Vertical/stacked configuration? ");
    AMSS.println(stacked);
    AMSS.print("LCDMerge: First LCD module is primary? ");
    AMSS.println(firstIsPrimary);

    // create messages to send to LCD modules based
    // on composite geometry
    Message mstrpri = Message.create(Message.NO_ADDRESS,
      Message.NO_ADDRESS, Message.NO_DEADLINE, AMSS.getClock(),
      Message.CALLBY, Message.SET, Message.STRING,
      mcall.parameterWidth(), mcall.parameterHeight());
    Message mstrsec = Message.create(Message.NO_ADDRESS,
      Message.NO_ADDRESS, Message.NO_DEADLINE, AMSS.getClock(),
      Message.CALLBY, Message.SET, Message.STRING,
      mcall.parameterWidth(), mcall.parameterHeight());

    if(mstrpri != null && mstrsec != null) {
      if(stacked) {
        for(int i = 0; i < 64 &&
        i < mcall.parameterWidth(); i++)
          mstrpri.setInt8(0, i, mcall.getInt8(0, i));

        for(int i = 64; i < 128 &&
        i < mcall.parameterWidth(); i++)
          mstrsec.setInt8(0, i - 64, mcall.getInt8(0, i));
      }
      else { // horizontal configuration
        boolean primarymod = true;
        int indexpri = 0;
        int indexsec = 0;

        for(int i = 0; i < mcall.parameterWidth(); i++) {
          if(primarymod)
            mstrpri.setInt8(0, indexpri++, mcall.getInt8(0, i));
          else
```

```
            mstrsec.setInt8(0, indexsec++, mcall.getInt8(0, i));

          if((i + 1) % 16 == 0)
            primarymod = !primarymod;
        }
      }

      AMSS.gc(mod.serviceCall(firstIsPrimary ? mstrpri : mstrsec,
        1, 0));
      AMSS.gc(mod.serviceCall(firstIsPrimary ? mstrsec : mstrpri,
        1, 1));
    }
  }
  else // print debug error message
    AMSS.println("LCDMerge: Could not determine geometry.");
}

AMSS.gc(vs1);
}
else {
  if(vn1 != null && vn2 != null && (vn1.getJ() != vn2.getJ() ||
     vn1.getJ() != vn2.getJ() || vn1.getK() != vn2.getK())) {
    AMSS.println("LCDMerge: Face north vectors not equal.");
    AMSS.print(vn1.getI()); AMSS.print(",");
    AMSS.print(vn1.getJ()); AMSS.print(",");
    AMSS.print(vn1.getK()); AMSS.print(" != ");
    AMSS.print(vn2.getI()); AMSS.print(",");
    AMSS.print(vn2.getJ()); AMSS.print(",");
    AMSS.println(vn2.getK());
  }

  if(ve1 != null && ve2 != null && (ve1.getJ() != ve2.getJ() ||
     ve1.getJ() != ve2.getJ() || ve1.getK() != ve2.getK())) {
    AMSS.println("LCDMerge: Face east vectors not equal.");
    AMSS.print(ve1.getI()); AMSS.print(",");
    AMSS.print(ve1.getJ()); AMSS.print(",");
    AMSS.print(ve1.getK()); AMSS.print(" != ");
    AMSS.print(ve2.getI()); AMSS.print(",");
    AMSS.print(ve2.getJ()); AMSS.print(",");
    AMSS.println(ve2.getK());
  }
}
```

```
      AMSS.gc(vn1);
      AMSS.gc(ve1);
      AMSS.gc(vn2);
      AMSS.gc(ve2);
    }

    return null;
  }
}
```

B.4.2 amss.algo.ServoCon

```
// ------------------------------
// amss.algo.ServoCon class
// Copyright (c) 2008 Andrew Lyle
// ------------------------------
// This program is free software: you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation, either version 3 of the License, or
// (at your option) any later version.
//
// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// GNU General Public License for more details.
//
// You should have received a copy of the GNU General Public License
// along with this program.  If not, see <http://www.gnu.org/licenses/>.

package amss.algo;
import amss.system.*;

public final class ServoCon {
  public static void main(Module mod) {
    while(mod.isRunning()) {
      Message mcall = mod.nextServiceCall();

      if(mcall != null && !primaryHandler(mod, mcall))
        mod.secondaryHandler(mcall);

      mod.statusCheck();
```

```
    if(mod.isPrimary())
      update(mod);
  }
}

public static boolean primaryHandler(Module mod, Message mcall) {
  boolean handled = true;
  Message mret = null;
  int srvfunc = mcall.serviceFunction();

  if(srvfunc == Message.GET)
    AMSS.println("ServoCon: Received Get service call.");
  else if(srvfunc == Message.SET)
    AMSS.println("ServoCon: Received Set service call.");
  else if(srvfunc == Message.APPEND)
    AMSS.println("ServoCon: Received Append service call.");
  else if(srvfunc == Message.RESET)
    AMSS.println("ServoCon: Received Reset service call.");
  else
    handled = false;

  if(handled) {
    AMSS.gc(mcall);

    if(mret != null)
      Message.enqueueMessageOut(mret);
  }

  return handled;
}

public static void update(Module mod) {
  int envcountrole1 = mod.roleEnvironmentCount(1);
  int envcountrole2 = mod.roleEnvironmentCount(2);
  float servovoltage = 0;

  // acquire all voltages
  for(int i = 0; i < envcountrole1; i++) {
    Message msg = Message.create(Message.NO_ADDRESS,
      Message.NO_ADDRESS, Message.NO_DEADLINE, AMSS.getClock(),
      Message.CALLBY, Message.GET, Message.NO_PARAM, 0, 0);
    Message mret = mod.serviceCall(msg, 1, i);
```

```
      servovoltage += mret.getFloat(0, 0);

      if(mret != null)
        AMSS.gc(mret);
    }

    // average and limit to values at most equal to 1
    servovoltage = Math.min(servovoltage / (float)envcountrole1, 1);

    // set servo position
    for(int i = 0; i < envcountrole2; i++) {
      Message mcall = Message.create(Message.NO_ADDRESS,
        Message.NO_ADDRESS, Message.NO_DEADLINE, AMSS.getClock(),
        Message.CALLBY, Message.SET, Message.FLOAT, 1, 1);

      if(mcall != null) {
        mcall.setFloat(0, 0, servovoltage);
        AMSS.gc(mod.serviceCall(mcall, 2, i));
      }
    }

    // delay due to latency considerations
    AMSS.sleep(AMSS.TICK_1S / 2);
  }
}
```