University of Glasgow | Department of Computing Science

IceOS: The Insense-Specific Operating System

Paul Harvey

April 23, 2010

Abstract

This dissertation details the creation of InceOS, a language-specific operating system used to animate Insense, a new component-based language for programming networks of computing devices. A detailed justification for InceOS is made by comparison with other language animation systems, as well as with the existing animation system, explicitly highlighting the inherent flaws and inefficiencies present. The new virtual machine underlying InceOS is presented, with which an Insense application interacts. The design, implementation, and operational considerations of InceOS are also presented. The efficacy of InceOS is quantified through an in-depth performance analysis and comparison with the existing animation environment.

"*Great things are done by a series of small things brought together.*"
– Vincent Van Gogh

Acknowledgements

To Professor Joe Sventek, for his consistent help and guidance throughout all stages of this project.

To Professor Alan Dearle, during the design of the interface the OS would provide.

To Dr Jonathan Lewis, for his input during the design stages, and persistent tolerance with questions as the how the runtime operates, even while on holiday.

To Phillipa, for letting me talk my jargon to her and making me dinner.

To my family, for reminding me of the importance of the button.

Contents

1 Introduction **1**

1.1 Statement of Research Problem 2

1.1.1 Motes 3

1.1.2 Scheduling 4

1.1.3 Buffering 4

1.1.4 Code Footprint 5

1.1.5 Power Consumption and Throughput 5

1.2 Solution 6

1.2.1 Scheduling 6

1.2.2 Buffering 6

1.3 Code Footprint 6

1.4 Power Consumption and Throughput 7

1.5 Document Outline 7

2 Literature Survey **9**

2.1 Microkernel 9

2.2 Abstraction Layers 10

2.3 Embedded OS 12

2.4 Language Specific Operating Systems and Runtimes 14

2.5 Summary 16

3 Insense Language Semantics **18**

3.1 Fundamental Concepts 18

3.1.1 Components 18

3.1.2 Channels 19

4 Components **22**

4.1 Design and Configuration 22

4.1.1 This Pointer 22

4.1.2 Component Create 23

4.1.3 Component Exit 24

4.1.4 Component Yield 25

4.1.5 Component Stop 25

5 Scheduler **27**

5.1 Component Control Block 27

5.1.1 Component States 29

5.2 Scheduling Algorithm 30

5.2.1 Sleep Vs Idle 30

5.2.2 Blocking 31

5.2.3 Unblocking 31

5.3 Pre-emption 31

5.4 Architectural Specific Considerations 32

5.4.1 Context Switching 32

5.4.2 Stacks 34

5.4.3 Interrupts and Pre-Emption 34

5.5 Component Death 34

6 Channels **36**

6.1 A Channel . . . 36

6.2 Configuration . . . 38

6.3 System Calls . . . 38

6.3.1 Channel Create . . . 39

6.3.2 Channel Destroy . . . 40

6.3.3 Channel Bind . . . 40

6.3.4 Channel Unbind . . . 40

6.3.5 Channel Duplicate . . . 41

6.3.6 Channel Adopt . . . 41

6.3.7 Channel Receive . . . 42

6.3.8 Channel Select . . . 43

6.3.9 Channel Send . . . 45

6.3.10 Multicast Send . . . 46

6.3.11 Channel Errors . . . 47

7 System Components **48**

7.1 Timers . . . 48

7.1.1 Timer Component . . . 48

7.1.2 Rtimer Component . . . 51

7.2 Sensors . . . 51

7.2.1 Button Sensor . . . 52

7.2.2 Pre-InceOS System . . . 52

7.3 Radio . . . 52

7.3.1 Radio Send . . . 53

7.3.2 Radio Receive . . . 54

7.3.3 Issues . . . 54

7.3.4 Pre-InceOS System . . . 55

7.4 Debug . . . 55
7.4.1 Print Component . . . 55
7.4.2 Led Component . . . 56
7.5 Dynamic Memory . . . 56
7.6 Name Server . . . 57

8 Evaluation **58**
8.1 Experimental Set-up . . . 58
8.2 Size . . . 59
8.2.1 Flash Occupancy . . . 59
8.2.2 RAM Occupancy . . . 61
8.3 Timing . . . 61
8.3.1 System Call Timings . . . 62
Component Create . . . 62
Component Exit . . . 63
Component Stop . . . 64
Component Yield and Pre-Emption . . . 65
Channel Create . . . 66
Channel Destruction . . . 67
Channel Duplicate . . . 67
Channel Adopt . . . 67
Channel Bind and Unbind . . . 68
Channel Send . . . 69
Channel Select . . . 73
8.3.2 Radio . . . 75
8.3.3 Timer Delay . . . 76
8.4 Throughput . . . 77
8.5 Power Consumption . . . 78

9 Future Work **79**

9.1 Improvements and Optimisations 79

9.2 Future Directions 80

10 Conclusion **81**

A Examples **85**

A.1 Insense Language Version 85

A.2 Compiled C Implementation 86

B IncеOS Virtual Machine Specification **87**

Chapter 1

Introduction

In a collaborative project between Glasgow and St Andrews, a new programming language (Insense [8, 31]) was designed and built to ease the programming of networks of battery-powered computers that interact via short-range radio communications. Based upon the π-calculus [26], it supports concurrent programming at the language level, and has been integrated with model checking systems.

The Insense language is an example of a domain-specific language [24]. A domain specific language (DSL) is used to address the specific problems of a single application domain. Examples of domain specific languages include spreadsheet formulas, SQL, and YACC grammars. Such languages are to be contrasted with C or Java, which are designed to be general purpose.

A DSL will usually offer more expression in a certain application domain in exchange for less generality. This has the advantage of reducing, or removing, the amount of domain specific knowledge that is required by the developer, consequently opening up the domain to more developers. This is precisely the situation with Insense. For example, the interactions with the radio hardware module, which can require the developer to understand hardware interactions in embedded programming, have been abstracted by the language into interactions with a well defined software component. Furthermore, concurrency is implicit in Insense, requiring no manipulation of threads by the developer.

Applications that are written in DSLs are animated by an underlying system which can usually be placed into one of the following three equivalence classes: microkernels, abstraction layers, or language-specific operating systems (OSs) and runtimes.

Microkernels are small OSs that represent the core aspects of an OS, such as boot loaders and drivers. From a microkernel, a customised OS can be built to animate the DSL application, as opposed to creating an entirely new system. This has the advantage of enabling a researcher to focus on the more "interesting" aspects of OS and language research. See Section 2.1.

Abstraction layers are intermediate layers that require the DSL application to be translated into an intermediate form that can be executed by a pre-existing platform or further translated to work with other platforms. This removes the need to create a customised OS or specific runtime. See Section 2.2.

DSL applications may also be animated by a newly created and customised OS or runtime. Such a system will entirely address the needs of the DSL and consequently will be more efficient than the above systems. See Section 2.3.

The implementation of Insense currently requires that a runtime library be supplied to support the Insense concurrency and communication semantics; this library, in turn, then orchestrates concurrency and communication functionality within the Contiki [11] operating system which controls the sensor node. Between the Insense runtime and Contiki there is substantial duplication, as scheduling is taking place both within the OS and within the library. Additionally, messaging is less than optimal, with buffering taking place in both the library and the OS. This duplication, besides leading to less efficient operation, consumes much of the RAM and FLASH in each node, with the result that it is difficult to construct anything but the most simple applications for use on the nodes.

1.1 Statement of Research Problem

The Insense language is currently compiled into an intermediate form (ANSI C) which then, with the help of an Insense specific runtime, is animated by the Contiki operating system, as shown in Figure 1.1. To be precise, the Insense language program is compiled into an application for Contiki. The Insense program in C is then compiled, together with the runtime and Contiki, into a binary image; this binary image is then loaded onto the hardware. The combination of the runtime and Contiki will henceforth be referred to as the pre-InceOS system.

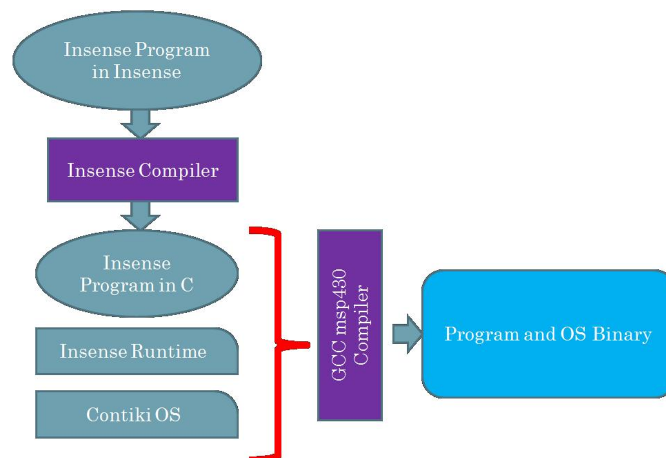


Figure 1.1: Compilation Process for an Insense Program

The fundamental problem faced by having Contiki animate Insense applications is that the Insense language has a different set of requirements to the services that are being provided by Contiki. For example, within Insense, communication between components is made explicit by using a channel mechanism to facilitate message passing. Within Contiki there is no such notion. In order to provide this service to an Insense application, function calls must be made into the runtime which then manipulates the features provided by Contiki in order to facilitate the channel mechanism. As a consequence of this constant use of indirection, performance is degraded. Also, due to the space occupied by the runtime and superfluous features of Contiki that are not required by Insense,

an Insense application's complexity is constrained both by the lack of available space and reduced amount of execution time available to it.

The following is an overview of the embedded hardware environment, followed by a breakdown of the specific problems that occur when Insense applications are animated by the pre-InceOS system.

1.1.1 Motes

Before discussing the main problem addressed by this research, it is useful to have an understanding of embedded wireless sensor networks; the following discussion is adapted from the Xenocontiki technical report [19] .

A wireless sensor network is a collection of low cost sensor nodes which interact using radio communications. Typically nodes are able to sense features of the environment around them such as temperature, humidity, vibration or light and then convey these readings to a central node. This central node may present its data to a human user in some way, store this data as a record in a database or respond to this data by affecting a change in the monitored environment.

Each node in the network, also known as a mote, will typically contain a low power CPU (between 2 and 8 MHz) accompanied by a low power radio. The radio's range varies between ten and a few hundred meters, and operates at a speed of around 250Kbps to communicate with other nodes. RAM capacities typically vary from 8KB to 32KB. A node may also come with extra components, like the sensors mentioned previously, depending on the role it plays within the network; some may be simply sensor nodes whereas others may be relays, which would not require sensors. An important component of a node is its power source, usually a battery. Despite a combination of low power hardware and considerate programming, a node would exhaust its battery after a few days of continuous use of all components; it is, therefore, commonplace to ensure that a node will enter a low-power *sleep* state while there is no work to be done. This technique can extend the life of a mote battery up to 99%, resulting in a lifetime increase from a few hours to years.

The nodes themselves are often small in size, as seen in Figure 1.2, with the largest volume of space taken up by the casing for the battery, usually supporting two AA batteries.

Figure 1.2: Mote beside an American 25c coin. Intel Research, Berkeley [19]

Although each individual mote has a limited amount of computational capacity, when networked together in tens or even hundreds, they are capable of quite sophisticated activities.

1.1.2 Scheduling

Currently, the Insense runtime has a scheduler that schedules the different components that make up the Insense application. Components encapsulate the Insense equivalent of a UNIX process and a Java object combined (see Chapter 4), and are the schedulable entities in Insense. Components are compiled into Contiki protothreads (see Section 2.3), which are scheduled by the scheduler contained within the Contiki OS. As a result, there are two independent schedulers which may work at cross purposes. This leads to inefficient use of the less-than-ample resources available on a mote (see Section 1.1.1). There will always be a delay between Insense scheduling a component and Contiki scheduling a component. This delay will vary, depending on what other tasks Contiki must perform.

1.1.3 Buffering

Within Insense, components are able to communicate with other components via *channels*. Channels are unidirectional; if components A and B wish to both send and receive messages, then two channels are required: $A \rightarrow B$ and $B \rightarrow A$. Within the pre-InceOS implementation, a single high-level channel is represented as two low-level *half channels*, each with an associated message buffer. In order to send a message from A to B, the message must first be created in a buffer within component A, then copied into A's half channel, then copied into B's half channel, then finally into a buffer in component B. This situation represents the worst case, however even the best case still requires three of these buffers. Channels may be composed to provide either 1:N semantics for the sender, or M:1 semantics for the receiver, as seen in Figure 1.3. These semantics, combined with half channel buffering, consume a large amount of space at runtime, which limits the size of both N and M, as well as the available space for other Insense features, such as components.

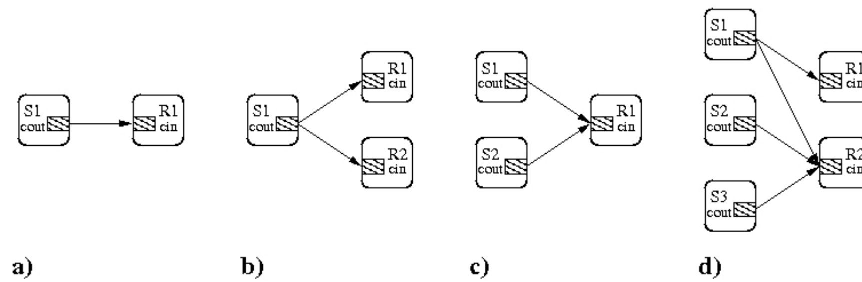


Figure 1.3: Channel Connection Semantics [31]

1.1.4 Code Footprint

As previously discussed, Insense applications are animated by a combination of an Insense runtime and the Contiki operating system, which takes up more physical space than is desired for resource constrained motes. Here the runtime acts as pure overhead to accommodate the Insense applications within Contiki. As an example, the following table shows the space consumption for the t-mote sky hardware [6], the currently-used hardware platform, with a single, simple hello world application.

Modules	Size(bytes)
Contiki	19236
Insense Runtime	7598
Insense App in Contiki	938

This translates to the runtime alone consuming 15.46% of the total flash space (48KB) on the t-mote. It is worth noting that both the runtime and Contiki use conditional compilation when including their modules, and this example does not include many features from either the runtime or Contiki, such as the radio.

1.1.5 Power Consumption and Throughput

Again, due to the amount of duplication present in the current system, more power is being consumed than is required for just an Insense application. For example, with two *independent* schedulers executing within a single system, power is being wasted in the second scheduler. At present, the Insense scheduler will decide that a component should execute and sends a *tick* to the relevant component. The Insense runtime then informs the Contiki scheduler that the relevant Contiki protothread is eligible to be executed. This protothread must then wait until control is handed to it to begin executing.

In the same way that power is being consumed by two schedulers, the same can be said of the throughput of the system. Instead of processor cycles being spent advancing the Insense application, they are being spent in the second scheduler or running some other element required by the abstraction.

1.2 Solution

The main goal of the Insense language is to open up embedded wireless networks to non-specialist programmers by removing the specialist knowledge that is required to program them. With the present situation, the pre-InceOS system does enable this, although in a restricted fashion. As a result of the limitations described above, a developer is again left constrained in a different manner.

Of the issues enumerated above, currently the code footprint is the most pressing. The situation is such that Insense applications are restricted to being relatively simple, and only so many of them may exist at one time. It is also the case that this limits the growth of Insense, as the underlying animation system has no room to grow. The lifetimes of the motes are being restricted due the duplication present, even if the application is not power hungry. The same argument applies to the throughput of the system; even though there are few components, they are not able to achieve their maximum potential.

The chosen solution is the creation of an operating system which has been customised to provide exactly the functionality that is required by the Insense language: InceOS.

1.2.1 Scheduling

As there would only be one operating system, compared to the current combination of OS and runtime, there would only be one scheduler. This would remove the current clash between the two schedulers and enable more processor cycles to be spent advancing the Insense program.

In the current system, some of the OS services are provided by the runtime and others by the runtime acting as a wrapper around elements of Contiki. In each case, the scheduling of the particular service is dependant on both schedulers reaching a consensus. With a single OS this issue is removed for free, even without any *inspired* implementation.

With both the system and user components being controlled by one single scheduler, not only will the system throughput improve but so will the system's responsiveness.

1.2.2 Buffering

As discussed in Chapter 6, data transfers will not require extra buffering, in fact no buffering at all. Data will be directly moved from component to component without the need for internal buffering. This makes more space available for other uses, such as more components or more complex applications.

1.3 Code Footprint

Inherent in the design of a system being created for a single purpose is the saving of space by comparison with a general purpose system. For InceOS, in particular, this will include the removal of the superfluous features of Contiki and interactions that the runtime must make with Contiki,

instead leaving behind a specifically targeted system, which is solely dedicated to the needs of Insense. As before, this will reclaim currently used space for the use of Insense applications or expansion of the services provided to an Insense application.

1.4 Power Consumption and Throughput

The proposed system will remove the extra cycles that are being consumed by the current duplication, as well as the power they consume. With a scheduler that has better control of the activity in the system, the removal of the extra buffering, and the removal of the duplication in general, the end result leaves a system that, even without attention to detail, will makes savings in terms of both power consumption and throughput.

Also, to improve the throughput of the system, and overcome a limitation of Contiki, the new OS will be pre-emptive. This is helpful as no locks are required in the system, meaning that components will not be required to explicitly relinquish control of the processor. This removes the need for such logic to be present in the C implementation of an Insense application.

The resulting system will have a longer lifetime, offer more execution time to its applications, and in general make Insense as a language a more attractive and viable option to wireless sensor network developers, both novice and expert alike.

1.5 Document Outline

The remainder of the document is split in to the following sections:

- **Literature Survey: Chapter 2**
  Review and critique of related works
- **Insense: Chapter 3**
  Overview of the language semantics of Insense
- **Component: Chapter 4**
  Design and implementation of Insense components in InceOS
- **Scheduler: Chapter 5**
  Design and implementation of the InceOS scheduler
- **Channel: Chapter 6**
  Design and implementation of Insense channels in InceOS
- **System Components: Chapter 7**
  Design and implementation of the services offered by InceOS to its applications
- **Evaluation: Chapter 8**
  Comparison of the pre-InceOS animation system for Insense against InceOS
- **Future Work: Chapter 9**
  Future areas for research and extension of InceOS

- **Conclusion: Chapter 10**
- **Example Transformation: Appendix A**
  Example of an application written in Insense and the equivalent representation for InceOS
- **InceOS Virtual Machine Specification: Appendix B**

Chapter 2

Literature Survey

The related work can be clustered into the following equivalence classes.

2.1 Microkernel

One approach to writing a new operating system, for a new language or any other purpose, is to use a pre-existing system to provide the core elements of an operating system. This includes boot loader code, drivers and kernel dynamic memory allocation. These elements can be considered to be the lowest common denominators in OS development and are often borrowed by new systems from previously existing ones [13], leaving room for exploration into the more "interesting" areas of OS research, such as scheduling. Borrowing can be done, for example, in the case of a radio driver because the radio hardware remains the same, thus the instructions needed to drive the hardware are the same, regardless of the OS in which these instructions are executed.

The OSKit [13] is such a system. The OSKit provides a set of well documented interfaces, boot-loader, minimal POSIX environment, and library code to help in building a basic operating system. Unlike the systems in Section 2.2, the OSKit makes no attempt to be portable, instead purposefully exposing the underlying hardware, with the assumption that this will be of use to the user. For example, OSKit provides functions to directly manipulate segment registers on x86 hardware. However, it is also possible to build platform-agnostic components on top of this. A useful feature of OSKit is that if the user's OS does not provide a custom implementation for some component in the OS, say the interrupt vector table, then the OSKit will use its default implementation. There have been a number of systems implemented for languages using the OSKit: the SR language, Java, and Standard ML [13]. OSKit was designed for desktop and server computers; none of the publicly-available information[1] indicates that it has been ported to embedded systems, in general, or sensor nodes, in particular.

This does offer a benefit to the research community by removing certain implementation stages required for OS development in research. However, as with many things, the advantage gained depends on the goal of the researcher or developer. If exploring general concepts which do not rely on any performance characteristics, such as generic algorithm or policy development, then this

[1] http://www.cs.utah.edu/flux/oskit/index.html, accessed 19/03/2010

system would be quite beneficial. If the work is more focused on performance, the generic modules would not be sufficient. Particularly for any work related to sensor motes, the generic modules would not be suitable. As explained in Section 1.1.1, the hardware and power environments are distinctly different from those offered by an x86 platform, and so any such representation would not prove useful for more than a basic exploration of concepts for a sensor device.

Another microkernel system is Choices [5]. Choices is an object-oriented operating system that has been built in such a way that it can be extended, customised or to have parts of itself replaced by other objects. This is achieved via the objects that are present in the kernel. Abstract interfaces are central to Choices, and the objects within the core kernel will implement the kernel interfaces. To replace a kernel object, a new object must simply implement the interface. An interesting notion from this system is that hardware interrupts are handled by treating them as software *raise* statements on exception objects. Like the OSKit, Choices enables a developer to customise its operation, although it is primarily focused around the modification of itself, and not to allow itself to be completely decomposed into a new system, like the OSKit.

In terms of embedded system development, Choices is not suitable, again due to the hardware environment of a mote. Embedded systems are usually finely tuned to the requirements which are made of them, unlike Choices which is attempting to be configurable. As for the OSKit, Choices would be an interesting platform for high level exploration of OS concepts and policies but not for an actual deployment implementation. Also, while it is true that the object-orientated paradigm is a powerful one, it may not be the best for embedded systems. As an example, the main purpose for the development of Insense was to ease the task of development for wireless sensor networks, and one the ways in which this is done is by using an object, or component, based structure in the language. However, the implementation of the current system that animates Insense uses a mixture of the imperative and event driven programming paradigms as they are more suited to the embedded environment.

In general, the idea of a microkernel has both advantages and disadvantages, depending upon the intention of the user of the system. If being used as an exploration tool of high level concepts and policies, then a microkernel provides an opportunity to rapidly develop an OS, although it is still true that the developer must still construct a system from these components. However, in terms of performance or development of an actual system, a microkernel is not the best method for achieving this. It is also the case that for embedded development, previous systems will have been developed for the targeted platform, consequently enabling a developer to *borrow* specific elements, such as device drivers, and reuse them. However, this often depends upon the implementation being open source and on the associated documentation.

2.2 Abstraction Layers

Another approach to the problem of implementing an operating system for a new language is to target the language's compiler at an abstract layer and not at a particular hardware architecture. This style can be considered as targeting an architecture in its own right, but this *virtual* architecture has the advantage that its implementation can be on any hardware, thus enabling the top level language to be portable even though it is only targeted at one architecture. It is also the case with this approach that different languages are able to be executed and work together because of the shared middle ground. By taking this approach the need to implement a new OS is not required, as any

system that implements the virtual architecture will suffice. However, this implementation will more easily accommodate some approaches than others, as will be explained below.

Java has a popular implementation of a virtual machine (VM), the Java Virtual Machine (JVM) [25]. The Java language is compiled into a series of instructions that resemble assembler style op-codes, known as bytecodes, and it is these instructions that are executed by the stack based JVM. The bytecodes will fit into a byte (8 bits), resulting in 255 possible instructions to the JVM. While the JVM does provide an abstract machine, it is one that is specifically targeted to the Java language. For Java, this is a great advantage in terms of efficiency, although for languages other than Java this presents a number of issues [16]. For example, Java only allows passing parameters by value. Should any language that requires parameter passing by reference wish to run on the JVM, then it will be required to *box* and *unbox* its reference parameters.

Java presents a concept which fits quite naturally to the envisaged idea of Insense. Mainly, the idea of a single language which can be distributed across many different platforms. This said the actual implementation is not quite so aligned. The need to have an intermediate execution layer which is "on-the-fly" compiling (or JITing) the instructions, explained below, does not fit well with the hardware constraints of a mote, either in space or computational power.

The idea of an intermediate layer for different language runtime integration is not a new one. In 1989, the Portable Common Runtime [33] (PCR) was implemented at Xerox PARC with the goal of close-coupled interoperation between different programming languages. One possible scenario for this system would be the creation of a single database management system which has been written in different languages. The PCR addresses the issues of language interoperation, such as garbage collection, thread implementation, symbol binding, and a shared address space, via a *common runtime layer*. The runtime operates on top of an existing OS, such as UNIX or Mach, and solves some of the issues by mapping functionality onto the features provided by the underlying OS. For example, when being animated by UNIX, the threads that are presented by the runtime map onto the thread implementation that is present in UNIX. If threads are not available, then coroutines are used. These are similar to protothreads in Contiki, see Section 2.3. I/O requests are mapped onto processes in the underlying OS and then carried out as normal. In order to use the runtime, languages will still require their own runtimes to operate on top of the PCR, and as a result of this garbage collection has been implemented within the PCR. However, this has the downside of needing to cope with languages with arbitrary pointers, such as C, consequently only allowing a conservative garbage collector. Symbol binding is one of the most challenging aspects of the system. In order to enable the different language applications to interoperate, the symbol tables from each of the object files from each language must first be parsed and saved within the PCR. This is to resolve any static links and maintain a master symbol table. The second stage requires runtime maintenance of the PCR symbol table and resolution of symbolic bindings between the different languages.

The PCR does not naturally align itself with the goals of Insense. This is mainly due to the fact that Insense is trying to present a single high level language, not a single language which can interact with others. Also, the space requirements for such a system may be acceptable for a larger desktop computer, but would simply be infeasible for a mote. In fact, this system is similar in style to the pre-InceOS animation system, consequently any problems or issues currently observed would be replicated with a new system of this type.

Another approach taken to the abstract layer is observed in Microsoft's .Net Common Language Runtime (CLR) and Common Intermediate Language (CIL) [18]. Here, a language is again tar-

geted at an intermediate language instead of at hardware. The main difference between .Net and Java is that the CIL was designed with the purpose of being targeted by other languages, whereas bytecodes were designed for Java. The VM target that is presented by the CIL is much more complex than Java bytecodes; there is a generic specification of types before reaching the .Net instructions. The runtime system is also stack based, however in the general case it is less efficient. For example, the CIL presents a generic *add* instruction that must have its operands' type checked at runtime which presents runtime overhead, whereas in Java, the intermediate bytecodes have specific instructions for integer add, floating point add, etc. This is assuming that the .Net instructions are being interpreted with just-in-time compilation, where they are compiled into native machine code at runtime. It is also possible to compile the intermediate language directly into machine code before runtime. These compilation and interpretation strategies are similar in Java.

This hybrid style of Java and the PCR would not present an optimal system for an embedded device. The general idea of compiling a language to an intermediate form is good, as it leads to a portable language. Also, the notion that this CIL could then be statically compiled into a native binary is also beneficial, presenting similar benefits to the Java style. Compilation into the CIL would be "overkill" due to the fact that only Insense would be using the system, meaning types would be standardised and the Insense compiler could be used to ensure type checking. However, the runtime checks that would have to be performed are not desirable, due both the the hardware constraints and the desire to have a system which will maximise throughput and efficiency.

As an aside, it is also possible to reach the other extreme of language implementation with direct hardware execution of the intermediate language. Unlike abstraction layers, it is possible to execute the bytecodes or .Net instructions directly on hardware, as in Jazelle [28]. By extending the instruction set of an Arm processor, it is possible to achieve direct hardware execution. Consequently, the need for an intermediate runtime is removed along with the overhead in space and time that it requires. However, this means that the overhead has now been switched to *more* physical hardware. It is also the case that the set of intermediate instructions (bytecode or .Net) cannot be extended, lest the new instructions must be handled in software. In such a situation the advantage that the hardware offers would start to be undermined. In this situation it would also be possible to reimplement the hardware, but this would be costly and time consuming, not to mention potentially making previous hardware obsolete.

In general, the notion of having an abstraction layer between the source language and the target platform is desirable, primarily due to the decoupling between the compilers and the target platform. Some of the implementation strategies discussed above are not optimal for an embedded platform, and the direct hardware implementation can not be extended easily.

2.3 Embedded OS

When writing software for embedded devices, the most important constraints are those presented by the hardware: the constrained battery, memory, communication, and processing power available. Operating systems for such devices have a number of common traits. Small code size is required and is usually achieved by a streamlined coding style, usually written in C or assembler. This is combined with compilation of only the essential components required by the application; for example, if the system does not use threads, then the thread modules will not be loaded. For this purpose, modularisation is also another common feature of such systems. In order to preserve

power, these systems will usually have one or more *sleep* mode, that force the mote into a low power state when a normal desktop machine would simply run the idle thread. Considering the remote deployment of some sensor networks, the ability to dynamically reprogram the motes during runtime is often included. The concurrency models used in such systems vary, with some using an event-based methodology and others using some form of threaded implementation.

Contiki [11] is a light-weight embedded operating system, and exhibits all of the above features. The organisation of the source tree is very modular, which aids in the optional compilation techniques discussed above, as the platform-independent sections are explicitly split from the platform-dependent sections. This makes porting from a software build point of view straightforward. Contiki by default uses an event-based model. Due to hardware constraints presented by the motes, events make many things straightforward. For example, heavyweight threads require their own stack, which could lead to the mote running out of memory for deep function call trees or even multiple threads. Also, this means that there is no need for locking mechanisms in Contiki, as two event handlers cannot run at once. However, due to this state machine model, it can be difficult to express programs. Furthermore, the style of programming can present a significant learning curve to programmers; for example, the need to explicitly yield a user *protothread*. A protothread is an event-based representation of a thread. Contiki also supports a pre-emptive multithreaded linkable library. Protothreads are scheduled via a combination of event posting to the protothreads and the scheduler "calling back" to the protothread, and in this case each thread must be explicitly yielded. Protothreads do not posses their own stack, instead a single stack is shared between all protothreads. As a consequence, the local state of a function will not persist after a protothread is unscheduled.

As Contiki is used in the pre-InceOS system, a more detailed analysis of which can be found progressively throughout the rest of this document.

TinyOs [20, 15] is another embedded operating system which is event-based. It differs from Contiki most notably in the sense that it is written in a C dialect, known as *nesC* [15]. TinyOs has a more explicit interaction with the hardware via an abstraction layer, know as the hardware protection layer. This layer serves as the barrier between the platform independent and hardware specific sections of the OS. In general the layout of TinyOs's directory structure is more complex than that of Contiki, thus making the learning curve somewhat steep for a new developer. This said, [20] does note the extremely short execution times for essential OS operations, such as context switching and memory operations. TinyOs also supports "over-the-air" reprogramming via the Deluge extension[2]. Scheduling is achieved via event posting to the scheduler, and once the scheduler routine is invoked the relevant task (locus of control) is executed.

TinyOs is a well used and proven embedded OS. Ease of development aside, it present a similar situation as Contiki in that the semantics which it offers to its applications are not the same as those required by Insense programs; in point of fact, they are less suitable than those presented by Contiki. The event driven style does not fit well with the Insense semantics, as Insense itself uses a quasi event system: the blocking rendezvous model, Section 3.1.2. Consequently, the new system should facilitate the events used by Insense, not simply include these events into those events present in an existing system.

Descartes [23] is a runtime system specifically designed for embedded networks that supports Dynamic C, a language similar to the SR language supported by the SR Portable Runtime System discussed in Section 2.4. The goal of this system is to give Dynamic C programs the ability to

[2] http://www.tinyos.net/scoop/section/Releases, Accessed 25/11/2009

perform remote procedure calls, run on embedded platforms, and to enable Dynamic C programs to use message passing as well as have quasi-dynamic creation of processes. Descartes also aims to give programs the ability to perform interprocess communication (IPC) between motes. In this parlance, a process refers to a thread of control that must be explicitly yielded. The coding style used is similar to that of Contiki, in that IPC is achieved through messages (events) and threads must be explicitly yielded. Remote Procedure calls are also possible, by representing the remote call instance as a process. Memory is partitioned into two logical categories: *xmem* and *rootmem*. Both of these are allocated from the heap, the difference being that rootmem is statically allocated, which requires programmer intervention during development to determine the number of different data structures required in the program, and xmem is dynamically allocatable. Descartes also allows messages to either be in ASCII format or binary format. ASCII is usually used for debugging purposes, and binary for deployment. In general, Descartes applications are not as efficient as Dynamic C programs, in terms of both speed and memory use [23]. However, the advantage is a resulting system that is less complicated and easier to debug. As this is an event-based model in all but name, it will suffer from the same issues discussed for Contiki, particularly in terms of the way in which processes are used, which are essentially just protothreads.

The core requirements of embedded systems, such as their small sizes and efficient implementations, are not only desirable for embedded development, but essential. All of the above systems use the event-based model to drive their systems, and this is one of the primary issues of having Insense animated by Contiki. Specifically, Insense is itself generating some explicit events, messages, but only during communication between components. The use of events in the underlying system makes it difficult to correctly capture the required semantics, and possibly introduces duplication in the events that are required.

2.4 Language Specific Operating Systems and Runtimes

In the following systems there are a number of similarities, although each system has its own style and features to support the language with which it is associated. All of the following languages are similar to Insense in that they are component-based with message passing. Components represent the primary language construct, and can be thought of as a cross between a Java object and a Unix process. In order for components to communicate, channels are used. Messages are passed along these channels, where these messages can take the form of primitive types (integers, characters) or more high level constructs (channels, components, records). Components usually represent a single thread of execution without any shared state, and as a result locking mechanisms are usually not required in the language. This eliminates the explicit blocking of components that is associated with locking, however the locking then becomes implicit by requiring a component to block in certain situations, Section 3.1.2.

An (unnamed) runtime [4] has been written by Luc Bläser in Oberon, to animate the Component Language [3]. As well as creating a runtime to animate the language, another goal was to create a system that was optimised towards a newer parallel programming model as opposed to the existing sequential model. Due to the constrained nature of the class of programming language that the system is intended to support, it is possible to support *millions* of parallel processes and have fast and predictable program execution speeds. This speed-up comes from the fact that components have small dynamic stacks that can be context-switched by saving and restoring only three registers. Switching can either be pre-emptive, due to a time-out, or explicit, by yielding. Scheduling in this

system is implemented as a simple FIFO scheduler, and all program structures are allocated on the heap. The compiler supports dynamic memory allocation by providing information on structure sizes that have been gathered from static analysis during compile time. Within the Component Language, components are able to be shared resources and require locking mechanisms (semaphores). Communication is achieved by message passing along channels. The two parties involved in communication take on the roles of client (sender) and server (receiver). Message passing cannot be performed in an arbitrary way, and must adhere to a protocol that has been specified in EBNF. For comparison, in Insense this is handled by typed channels.

This (unnamed) runtime provides a solid model that could be adapted for an embedded system. The small stack size and simple context switch, along with the associated performance, would be very advantageous on a mote. However, the EBNF specifications would need to be simplified to typed channels which could become a compiler issue, rather than a runtime issue, and the restriction added that components cannot be shared.

The Active Oberon System [27] is an OS that is mostly written in Oberon and animates the Active Oberon language [17]. It is implemented in two halves: the lower level is implemented in Oberon, and the upper in Active Oberon; the reason for this split is not specified, although it is assumed so as to provide an easier integration with Active Oberon applications. The goal of this system is to support *active object multitasking*. To continue the Java analogy, an active object is the same as a Java object with methods, state, and the object itself acts as a locking mechanism. However the main difference is that each active object has an implicitly associated thread that executes the associated "behaviour" of that object. It is important to note that once such a thread terminates, the object remains and reverts back to what is conceptually a Java object. The entire system works within a single address space and relies on the compiler to catch high level language errors, as there is no explicit error handling in the OS.

The Active Oberon language draws many parallels with Insense. However, the main difference, shared state, does not mix well. In terms of structure, the notion that a component lives on after its behaviour stops is not acceptable. More generally, this system is not designed for embedded systems and as a result has a large code footprint. Insense is not an extension of another language, even though parallels may be drawn with languages such as those in this section.

Inferno [9] is an operating system that has been designed for creating and supporting distributed services. Unlike the other systems described in this section, the associated language (Limbo) has been specifically designed for Inferno as a way to author Inferno applications. Inferno can function as a standalone operating system or as a user application on many different main stream OS's. It has been ported to multiple architectures and can run within 1MB of space. It uses a virtual machine called Dis to provide the same interface to applications, independent of the platform upon which they are executing. This virtual machine defines its own byte-codes. As with the languages for the previous two systems, the Limbo language is arranged into components that communicate typed data along channels, however pointers are allowed. Beneath the Dis VM is the Inferno kernel. Inside the kernel is where the scheduling, memory management, interrupt handlers and devices drivers are located. The Inferno system is a privately owned framework from the Vita Nuova Holdings Ltd (previously Lucent Technologies), and as such, the implementation details of the lower kernel are not documented, even though the OS's source has been released.

Inferno is primarily a distributed OS, which relies on the styx [9] protocol to facilitate the RPC calls that are used to animate message passing. This particular style of message passing implementation it not compatible with the Insense view. The runtime requirement of 1MB of RAM is also a difficult

requirement to satisfy on many motes.

The Portable Runtime System [2] describes a portable runtime system for the SR language. As the title suggests, the main goal of this system is to enable the SR language to be portable. To achieve this, the system uses a component based. Within SR there is a notion of a virtual machine, although it refers to an isolated address space on a machine. The runtime itself serves as an abstraction layer over existing systems, and may not function as a standalone system. As SR is a concurrent language, the main function of the runtime is to provide a threading interface that maps a static interface, which is presented to the SR language, to an actual thread implementation in the underlying OS; for example, pthreads in Linux. The runtime achieves this via C macros. As the actual threads are the property of the underlying OS, it is that OS's responsibility to schedule them, however to protect critical regions the runtime presents a macro to the SR language that enables the use of underlying locks. If pre-emptive scheduling is not used, then these macros are empty. Thread creation is also hidden behind a macro, and the parameters that are passed resemble those for creating a pthread. The main difference is that the runtime does not pass the actual function to be run in the thread, but rather an intermediate function that calls it. This is done in order to guarantee that an arbitrary number of parameters may be passed to the threaded function, and to bind the thread to a runtime structure. An interesting point is that this runtime has been ported to an OS (Fluke [14])that has been implemented with the OSKit, discussed in Section 2.1.

The main advantage that this equivalence class offers, is that each system is targeted directly to the needs of the language that they animate and, as highlighted below, this is a desirable quality. However, such systems require a great deal of time and effort as they must be designed and built from scratch.

2.5 Summary

It can be seen that there is little work that is directly applicable to the creation of a language-specific operating system for embedded devices. There are examples of OSs for embedded devices, but none that have been designed with the goal of animating a single language. There have been operating systems that have been designed for single languages; however these have not been directly targeted at embedded systems.

This current *knowledge gap* with respect to language-specific operating systems in general, and for motes in particular, justifies this work.

InceOS is by definition a language-specific operating system, and fits into the fourth category discussed above; Section 2.4 details the other members of this class. This class was chosen due to the goals of the project, specifically the desire to have a system which is as efficient, as small and as targeted to Insense as possible. The other equivalence classes were simply not able to offer this level of precision or customisation. Also, it can be seen that the programming models observed in the fourth equivalence class are component based, using message passing to communicate. Even though the precise details of these systems are not identical to Insense, there is enough to draw a parallel. InceOS draws on a feature of another class: mainly the abstraction layers present in Java. Consequently, the Insense language is targeted at the Insense virtual machine which can be found in Appendix B. This is not a virtual machine in the same sense as the JVM, but rather a description of the system call prototypes and data structures that are available to Insense programs, and which are supported by InceOS. It is also the case that the spirit of the third equivalence class will be

incorporated by InceOS, mainly a very small and efficient implementation that offers exactly the services and semantics that are required to animate Insense programs and no more.

Chapter 3

Insense Language Semantics

The following chapter provides an overview of the Insense language. This is necessary to provide the rationale behind the design and implementation decisions found within the subsequent chapters, such as the system calls and intra-OS interaction.

3.1 Fundamental Concepts

The two fundamental concepts in the Insense language are *channels* and *components*.

3.1.1 Components

Components represent the fundamental schedulable entities within Insense. A component consists of zero or more state fields; a single thread of execution known as a behaviour, which may access the component's state; one default, plus zero or more user-specified constructor functions that are used to initialise a component; and also zero of more functions. A component's behaviour consists of an infinite loop, within which user-specified code is executed. This code may interact with the state associated with the component or the functions associated with the component.

A component may be created, self terminate, be terminated by another component, or explicitly yield control of the processor during execution of its behaviour loop.

Components are similar to Java objects, with reference to the state fields or functions that are associated with that object. However, a very important difference is that Insense components execute in isolation of other components - i.e. there is no shared state. One component may not access another component's state, or invoke it's functions. This model of computation draws certain parallels with the *actor* model of computation [7]. Components are able to communicate with each other; however, this must be done through channels.

3.1.2 Channels

Within Insense, channels are used to represent one half of a binding between components, and enable unicast communication. Channels have an associated direction (*IN* or *OUT*) which is specified during a channel's creation. In order for one component to communicate with another, two channels will be required: one OUT and one IN. An OUT channel is always associated with a component that wishes to send data, and an IN channel is associated with a component that wishes to receive data.

Within Insense, channels are considered to be strongly typed in relation to the types of the data that they transfer. These types can consist of standard types, such as integer (`int`) and character (`char`), as well as application defined types. Insense also support a generic type known as the `any` type. It is possible for any data type to be cast to this `any` type.

Before any data may be sent between components over a channel, the IN channel of a receiving component must first be bound to the OUT channel of a sending component. Either type of channel may be bound to a number of other channels, provided that they are of the opposite polarity and associated with identical data types. The different connection patterns that channels may form can be seen in Figure 3.1.

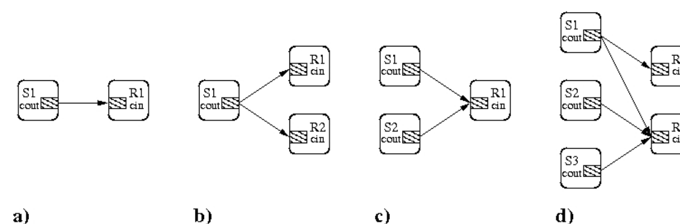


Figure 3.1: Channel Connection Patterns [8]

After a successful binding, a component may execute a receive request on an IN channel or a send request on an OUT channel. In the case of only one connection, a) in Figure3.1, S1 may only push data across to R1 if and only if R1 is in a blocked state and therefore ready to accept data, otherwise S1 will enter a blocked state. In such a case it is up to R1, at some later point, to execute a receive on channel *cin*.

Data may be pushed to a receiving component or pulled from a sending component depending on the conditions listed in Table 3.1.

Sender State	Receiver State	Action
Sending	Runnable	Sender will block until the Receiver executes a *receive* on the channel bound to the Sender's channel
Sending	Blocked	Sender will successfully send the data to the Receiver and continues executing. Receiver subsequently becomes unblocked and acts on the data
Runnable	**Receiving**	Receiver will block until the Sender executes a *send* on the channel bound to the Receiver's channel
Blocked	**Receiving**	Receiver will successfully receive the data from the Sender and continues executing. Sender subsequently becomes unblocked.

Table 3.1: Insense Channel Send and Receive Rules

To highlight the rules, the example discussed above of a single binding is shown in Figure 3.2. The situation shown is where component B wishes to send some data to component A. Firstly A is running, (a), and is pre-empted allowing B to begin execution, (b). B then enters a blocked state while attempting to send to A because A is not currently blocked executing a receive request, (c). At some later time, A then begins executing and executes a receive request, this makes B eligible to execute again and transfers the data to A, (d).

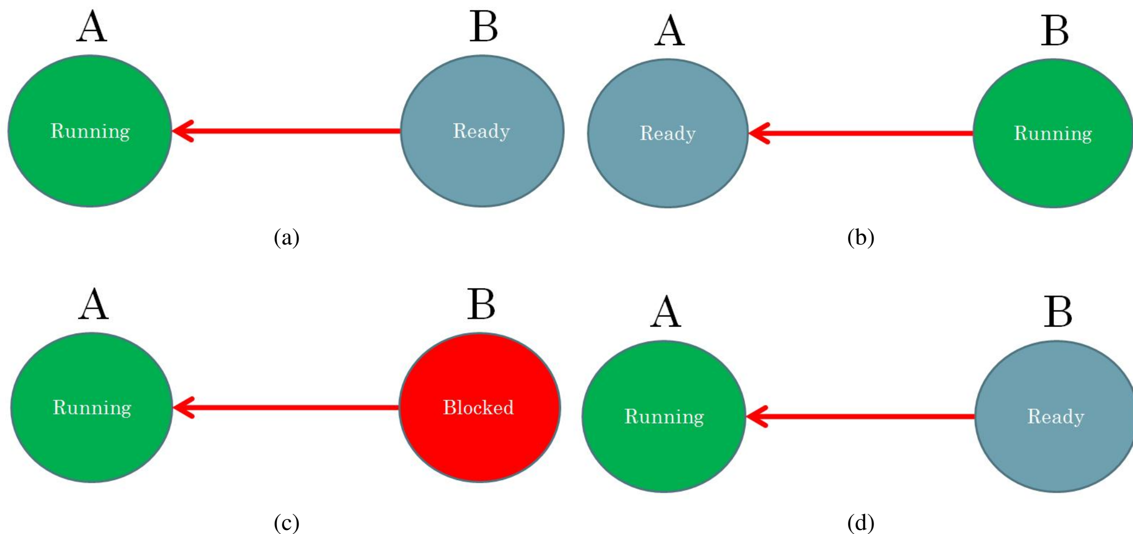


Figure 3.2: Channel Rendezvous Mechanism

As shown in b) and c) of Figure 3.1, an individual channel may have one or more connections; these situations are not reflected in the rules of Table 3.1. The language requires that when sending or receiving data along a channel which is bound to more than one other channel, one channel will be non-deterministically chosen and acted upon. At this point, the situation reverts back to the single binding case.

The final action that Insense supports is the ability to chose between a number of channels upon which to execute a receive: the *select* statement.

```
select {
        receive x from chan1 when p > 7: p := x
        receive y from chan2 when p < 7: p := y
        receive z from chan3: p := z
        default: p := 1
}
```

Listing 3.1: Insense Example Select Statement

Listing 3.1 shows an example of a select statement in an Insense program. Associated with each receive call on a channel is an optional condition. A channel is eligible for consideration in the select statement if the associated condition evaluates to true; in addition, if the condition is omitted, the channel is also eligible. If more than one channel is eligible, then a non-deterministic choice is made among the eligible channels. If no channels are eligible and a default clause has been specified, than the default action is executed; if no channels are eligible and the default clause has been omitted, it is an error.

In summary, components may perform the following actions upon channels: create, destroy, bind, unbind, send data over a bound channel, receive data over a bound channel, and perform a select over a one or more bound IN channels.

Chapter 4

Components

The following chapter discusses the design and implementation of the component abstraction within the OS and how it is associated with the component concept in the Insense language.

4.1 Design and Configuration

As stated in Section 3.1.1, the component abstraction must support a number of operations; in order to meet these requirements, the C language system calls in Listing 4.1 are provided by the OS.

```
void *component_create(void (*constructor)(), void (*behaviour)()
                       int struct_size, int stack_size,
                       int argc, void *argv[])
void component_exit(void)
void component_yield(void)
void component_stop(void *this_ptr)
```

Listing 4.1: Component System Calls

The component system calls are used to create a component, destroy the invoking component, pause a component's execution, and destroy the component referenced by the `this_ptr` variable, respectively.

4.1.1 This Pointer

Each component has its own state. In order to access this state within the C language representation of the Insense application, each component is provided with a *this* pointer. As explained below, it is the duty of the OS to allocate the *this* structure for a component, however it is the duty of the Insense compiler to define the layout of the *this* structure for a component. As shown in Listings

4.2 and 4.3, the user component must cast the value returned by the OS to a pointer to the type of the structure which the compiler has defined for the component.

Listing 4.3 shows the requirement that the first field of any *this* pointer structure must be an integer. This *stopped* field is used for every component animated by InceOS. Specifically it is used as the loop condition in the behaviour of each component. This enables the `component_stop` call to function correctly, as described below.

Before a discussion of the implementation of the system calls, it is worth noting that in order to produce such a streamlined and targeted system, each of the main elements within InceOS (component, channel, scheduler) interact closely with each other, leading to a tightly coupled system. Consequently, it may be difficult to gain an understanding of the operation of the whole system at first glance.

4.1.2 Component Create

This is the most involved of the component system calls. As implied, this system call is tasked with creating a new component, however this is not quite as simple as discussed in Section 3.1.1.

The first point to mention is the `struct_size` argument from `component_create()`, Listing 4.1. In Section 3.1.1 it was established that a component had an implicit associated state which was analogous to the state within a Java object. Within InceOS, this state is represented by a C `struct` associated with each component. This argument specifies to the system call the size that is required by this structure, in turn the system call will allocate that amount of memory and return it to the caller as a generic pointer (`void*`). It is then the duty of the caller to cast this pointer to the type of the structure, the size of which was specified, as explained in Section 4.1.1.

```
/*Declare this component's This Pointer*/
struct this_ptr{int stopped,int var1;char var2;};

/*Storage for the This Pointer*/
struct this_ptr *this;

/*Create component and return a reference to its 'this pointer'*/
this = (this_ptr*)component_create(&constructor_function,
                                   &behaviour_function,
/*Size of struct to be allocated*/ sizeof(struct this_ptr),
                                   STACK_SIZE,
                                   num_arguments,
                                   list_arguments);

/*May now use and update internal state*/
this->var1 = 6;
this->var2 = 'a';
```

Listing 4.2: This Pointer Allocation and Usage

A component has one default constructor, and may have zero or more user-specified constructors associated with it and, if specified, it is required that a constructor be invoked during a component's creation. `component_create` achieves this via the `constructor` function pointer and `argc, argv` arguments. During the creation of a component this function will be invoked with these arguments: `constructor(this_ptr, argc, argv)`. Here the arguments `argc`, and `argv` refer to number of arguments and the arguments themselves which are being passed to the constructor respectively. This is the same paradigm as is used to pass arguments to the main function of a C language program. The `this_ptr` argument is the *this* pointer, as discussed above. As the pointer is allocated by the system call, it is the OS's responsibility to pass this argument, however is the responsibility of the constructor to cast this pointer to the appropriate type. In the Insense language there is always at least one constructor, however this can be an empty constructor which is nothing more than an empty function. In this situation, the `NULL` value is passed in place of a pointer to the constructor function in the create call. This is the responsibility of the Insense compiler.

This style has two advantages. Firstly, the OS simply needs to pass parameters back to an external function, thus not requiring any intervention or interaction with elements not within its control. Secondly, this style still allows many different constructors via one mechanism, although it relies on the Insense compiler to provide the appropriate constructor for the component as a parameter to this system call.

The final parameter is `stack_size` which is used to specify the size of the stack that this component will require. As mentioned previously in Section 3.1.1, each component has an associated thread of execution which is known as it's behaviour, and accordingly each component must have its own stack. This is required because, unlike Contiki, InceOS offers pre-emption, and as a component may be pre-empted at any point in its execution, it must be possible to preserve and restore the execution context of that component at any point. The value of `stack_size` is determined by compile-time analysis by the Insense compiler, and contains the largest stack size that is required to successfully run that component. InceOS will also require a certain amount of extra space to be allocated on the stack, however this will be discussed in Chapter 5.

The final actions when creating a component are firstly, to initialise the *stopped* field of the *this* structure to enable the behaviour loop to execute at least once. Secondly, the component is added to the scheduler queue, so that it may be scheduled. Finally, a reference to the *this* pointer is returned to the caller.

As in all systems, there must be a first component created and handed to the scheduler to execute when system initialization is completed. In InceOS it is known as primordial main, and is responsible for the initial creation of application components and binding of their channels. In Appendix A the bindings present in this system element are generated from the connections which are present at the end of the Insense program. A similar approach has been taken in the pre-InceOS system.

4.1.3 Component Exit

It must also be possible to destroy a component once it has been created and `component_exit()` is used to do this. The actual act of destroying a component's internal representation is delegated to the scheduler and will be discussed in Chapter 5. It is the responsibility of `component_exit()` to update the status of the component to ensure that the scheduler will remove all traces of it.

It should be noted that unlike the runtime, InceOS will not automatically return any dynamically allocated state, which was allocated by the application, to the heap. It is the responsibility of the Insense compiler to generate code that will handle the return of allocated space before calling this function. This is due to the fact that InceOS does not use a reference counted memory system, see Section 7.5.

4.1.4 Component Yield

It is the responsibility of `component_yield()` to pause the execution of the currently executing component and cause it to be rescheduled. Within InceOS, a call to yield leaves the component eligible to run, and simply requests that the scheduler add it to the end of the appropriate run queue. Again, the specifics of the implementation are delegated to the scheduler, Chapter 5.

4.1.5 Component Stop

As stated in Section 4.1.1, each *this* pointer must have an integer variable as it's first field, and it is this variable that acts as the condition of the loop within the behaviour of each component, as shown in Listing 4.3. `component_stop()` enables any component to stop another so long as it has a reference to the other component's *this* pointer. This is achieved within the system call by simply setting the stopped field within the other component's *this* pointer from 0 to 1. As a result, the next time the component to be stopped tests the condition in the loop, it will fail and the component will execute the `component_exit()` system call and terminate itself.

```
void behaviour(void *this)
{
        MY_STRUCT *foo = (MY_STRUCT*)this;

        while(!foo->stopped)
        {
                /*Do work*/
        }

        //Placed by the Insense compiler,
        // a behaviour must never return
        component_exit();
}
```

Listing 4.3: Standard Layout of a Component's Behaviour

As with `component_exit()`, it is the responsibility of the Insense compiler to handle the return of any dynamically allocated memory by generating the appropriate code before exiting the component. Within the Insense language, memory allocation is very similar to the style used in Java. Space is allocated for data via the `new` keyword, and space is never explicitly returned in the

language. Currently this is handled by the pre-InceOS system using a reference counted memory allocation system.

Chapter 5

Scheduler

The following chapter details the design and implementation of the scheduler within InceOS. It discusses the component control block which is used to represent a component, the scheduling algorithm, and interrupt driven pre-emption.

5.1 Component Control Block

Within a general purpose OS, such as Linux, threads represent a schedulable entity to user programs. A data type, know as a *task control block (tcb)*, is used to represent a thread within the core of the kernel. Amongst other things, a tcb contains the current state, unique identifier, and priority level of a thread. These values can be used by the scheduling algorithm to schedule the thread, or by the system to know if a thread should be reclaimed as it is a zombie thread [1].

Just as in a general purpose OS, InceOS requires a data structure to maintain the representation of a schedulable entity. This is known as the component control block (CCB), Listing 5.1.

[1] In pthread parlance, a zombie thread refers to a thread which has been started but not joined.

```
typedef struct ccb{
      unsigned char status;      //component status
      unsigned char ident;       //component unique ID
      struct ccb *next;          //next CCB on run queue
      void (*behaviour)();       //components behaviour
      void *state;               //component's this pointer
      Channel blocked_channel;   //channel component has blocked on
      Channel chans[SCHED_CHANNELS];  //component's channels
      SELECT_DATA *select_ptr; //channels being selected from
#ifdef TARGET_XEN              //Xen specific context information
      unsigned long sp;        //stack pointer
      unsigned long pc;        //program counter
#else                          //MSP430 specific context information
      unsigned short sp;         //stack pointer
      unsigned short pc;         //program counter
#endif
      char stack[1];             //the component's stack
}CCB;
```

Listing 5.1: Component Control Block

The first field of a CCB is used to indicate the current state of the component. This value will contain one of the values discussed in Section 5.1.1. The second field, `ident`, refers to a unique identifier associated with each CCB. The `unsigned char` data type used is eight bits on the msp430 [6], which is the micro-controller found on the t-mote. This gives 256 unique values to represent components, which is currently sufficient. However, should this change and more values be required, then by simply using an unsigned integer to store the value, 65536 possible unique values are generated at the expense of only 1 more byte per CCB on the t-mote. The third field is used as a pointer to the next CCB, as required for the ready queue discussed below. The fourth field is used to store the function pointer which was passed to `component_create()`, and is used to initially represent the behaviour function associated with component. `state` is a pointer to the *this* pointer for the component, and is required both for initially providing the behaviour function with its *this* pointer, and when destroying the component, see Section 5.5.

The following three fields are used to facilitate the communications semantics within Insense, a full description of which can be found in Chapter 6. The first field, `blocked_channel`, is used to indicate the channel which the current component has blocked on while either trying to execute a send or receive request. The second, `chans`, is used to specify the channels which are associated with the component that this CCB represents. As this is a static array, the number of channel which may be associated is limited to a predefined value, `SCHED_CHANNELS`. Currently this value is 20, which was decided after an analysis of the existing Insense applications. This array is initialised during the creation of a CCB. The final field is a pointer to a `SELECT_DATA` data type. This is used when a component must block while executing a select request over a number of channels.

The next fields are architecturally specific to the target platform. The values, sp and pc, are used to hold the stack pointer and program counter respectively. In this case, the two architectures are the Xen hypervisor [1], and the t-mote sky. When executing a context switch, it is these values which

are used to both save (swap out) part of the state of one component and restore (swap in) another. The final field is used to store the stack which is associated with each component, the size of which was initially specified to the `component_create()` system call. The stack is allocated from the heap, together with the CCB, by specifying the size of the CCB structure plus that of the required stack. The generic pointer which is returned is cast to a CCB structure, leaving the extra space to be used as the stack. The stack field of the CCB acts as a place holder for the actual stack, and is also used when allocating the initial value of the stack pointer (sp). Stack allocation is done in this way to eliminate the possibility of memory fragmentation that may occur from two separate allocations of the CCB and stack. It is possible that this is a situation where static allocation could be used for the CCBs, as with the `CHAN`s, and the stacks could be dynamically allocated as required. However, currently Insense language programs use a relatively small number of components, which are not often destroyed (stopped), therefore this was not done. Although, this could be achieved by a simple modification to the implementation.

Contiki also has an equivalent data type know as a process which is used to represent a protothread. In this case the process would be equivalent to the CCB, and protothread equivalent to the component.

5.1.1 Component States

In order to implement the blocking rendezvous model that is required by Insense, a component may be in one of two fundamental states: running or blocked. This model was too simple when designing InceOS and as a result there are now eight different states, as enumerated by Listing 5.2.

```
enum component_status{
    RUNNING,              //Currently running
    READY,                //Ready to run
    INTERRUPT_READY,      //Component was unblocked by an interrupt
    BLOCKED_SENDER,       //Component blocked trying to send
    BLOCKED_RECEIVER,     //Component blocked trying to receive
    BLOCKED_SELECT,       //Component blocked trying to select
    BLOCKED_SENSOR,       //Component blocked waiting for a sensor
    KILLED                //Component should be killed
};
```

Listing 5.2: CCB Operational States

The first two states, `RUNNING` and `READY`, are used to indicate a component that is currently executing or is ready to run, respectively. A component is placed in the `INTERRUPT_READY` state after it has been made eligible to run again by a hardware interrupt. This is used by system components only, as explained in Chapter 7, as this state is required to explicitly indicate when a component has been made runnable again by an interrupt. Accordingly, if a component that has blocked due to a system call returns while in this state, a unique value is returned to indicate this (-2). Currently it is only system components which use this feature, see Sections 6.3.7 and 7.1.

The `BLOCKED` states are used to indicate that a component has blocked while trying to perform

the indicated action: sending, receiving or selecting on a channel(s). The exception to this is BLOCKED_SENSOR. This is used when a component must block while *only* waiting for a hardware sensor; currently the only usage of this state is for the button component. This is explained fully in Section 7.2.1. This was used to save both space and time. Unlike the other blocked states, this state does not require the transfer of any data, and therefore does not require some of the actions and assertions that must be made by the OS, such as ensuring the correct buffer size, or ensuring that no components have blocked waiting on this component when it blocks. Conversely, by simply blocking directly into the BLOCKED_SENSOR state, this process is not required.

The final state is used to indicate that a component is to be terminated, and should be disposed of in a graceful manner. This state is required as a component needs a third party to dispose of it, see Section 5.5.

5.2 Scheduling Algorithm

A simple round robin scheduling algorithm is used within InceOS at present, and consists of a single run queue. Any component which is pre-empted or yields is placed at the end of this queue. The next eligible thread to run is always situated at the head of this queue. In terms of complexity it is O(1) for both insertion and removal.

Even though there is only one ready queue, the scheduler still supports two priority levels within the system: normal and interrupt. Normal refers to the case described above where a component is placed at the end of the run queue. Interrupt refers to the case where a component that may receive an interrupt, such the timer or radio components discussed in Chapter 7, is unblocked and placed at the head of the ready queue. As the next component to be run is always taken from the front of this queue, components that are placed at the head are given a priority boost.

When invoking the scheduler, the process is as follows. First, save the CCB of the currently running component at the end of the run queue. Second, get the CCB at the head of the run queue which will be the next component to run. If the saved currently running CCB and the next CCB to be run are not the same then execute a context switch, otherwise do not. Finally, process any components which have exited, see Section 5.5.

This choice for a simple scheduler was originally made to enable development and testing of other elements of InceOS. The simple scheduler is still present due to the time required to explore the effect of different policies and their effects on scheduling.

5.2.1 Sleep Vs Idle

In a larger system, such as Linux, whenever there is no work found, the idle thread is run until more work becomes available.. In an embedded system where the power supply is limited, the mote is placed into a sleep state when no work is found in order to preserve battery power. Sleep states turn off different hardware modules within a mote depending on the sleep state used. Within the t-mote there are four different levels of sleep, or low power mode. that will disable different combinations of the cpu, hardware clocks and the radio. When no work is found in InceOS, low power mode 3 (LPM3) is used, meaning that all hardware is turned off except the two hardware clocks, and the

radio. This enables any outstanding timer requests or incoming radio packets to be delivered, see Chapter 7. Upon generation of an interrupt from one of these components, all hardware on the t-mote is reactivated, ready to be used.

5.2.2 Blocking

Usually it is the case that a scheduler will also maintain a queue of elements representing the blocked components, as well as those which are ready to run; however this is the not the case in InceOS. As explained in Section 6.1, each channel is by default associated with the component it has been created by. Within InceOS this binding operates in both ways: a channel maintains a reference to its component, and a component maintains a reference to its channels. The `chans` field in Listing 5.1 is used to maintain this association for a component, with the `comp` field in Listing 6.1 used to make the association for a channel.

By maintaining these associations, a component that has blocked will always be referenced by the channels which it owns. This implementation is correct due to the blocking semantics required by Insense. When a component blocks on a channel with no bindings, then the Insense language requires that this component should stay blocked until such time as another channel is bound to that channel and a data transfer operation is executed.

As will be explained in Chapter 6, it is the responsibility of the channel system calls to unblock a component when appropriate, and it is this requirement, coupled with the fact that channels maintain references to the components to which they are attached, that means a blocking queue is not required. In the above situation, any action to transfer data across the newly bound channel will cause the blocked component to become unblocked and re-scheduled.

System components who block on the `BLOCKED_SENSOR` state present a special case. In this situation it is required that the interrupt service routine for the associated sensor posse a reference to the component that blocks in such a state. Whenever an interrupt is generated by this sensor, it is its responsibility to unblock and reschedule this component. A reference to the component to be unblocked is obtained during the initialisation of the sensor. This close coupling is another reason why only system components can use this state.

5.2.3 Unblocking

Unblocking a component is a fairly simple process. In the normal situation, unblocking a component consists of placing it at the end of the run queue. A special case, which is reserved for components that are awoken by interrupts, involves placing the component at the head of the run queue. In each case, scheduling will continue as normal the next time the scheduler is invoked.

5.3 Pre-emption

Within the pre-InceOS system, a component is represented by a protothread [12], see Section 2.3.

There are two problems with protothreads. Firstly, they are cooperative, i.e. a protothread will run until it relinquishes control of the processor; this can happen either by an explicit yield call or blocking system call. This means that each component in the pre-InceOS system will not be guaranteed a fair share of the processor, and that the progress of the different components will not be balanced. Secondly, a protothread works by repetitively calling the function which represents the thread. A protothread does not possess its own stack, instead all protothreads share a single stack. This does have the advantage of reducing the size requirements of a protothread, however this means that no local state persists after a protothread yields or blocks, as the call frame being used for the protothread is removed from the single stack.

In relation to Insense, these issues cause several problems. The first point requires that the Insense compiler insert an explicit yield at the end of every behaviour loop in order to prevent a component from monopolising the processor, even though the application may not require it. This prevents any one particular component making consistent progress, even though progress can still be made. The second point necessitates the use of statically allocated global variables or dynamically allocated memory instead of using local variables.

Within InceOS, a pre-emptive multitasking environment is provided. Here a component will relinquish control of the processor by an explicit yield call, a blocking system call, or by relying on a hardware interrupt to pre-empt it after the time quantum has expired[2]. The option to yield refers to the `component_yield()` system call, Section 4.1.4, and the blocking option refers to the blocking operations discussed above, and used in the system calls described in Sections 6.3.9, 6.3.7, 6.3.8.

Pre-emption is achieved via one of the t-mote's two timer interrupts, specifically timerA, the one second timer. The timer itself actually generates an interrupt thirty two times a second, resulting in a minimum time quantum of approximately thirty milliseconds. When timerA generates an interrupt, the associated interrupt handler routine is invoked. The handler will then invoke the scheduler routine, which in turn will cause a context switch depending on the scheduling algorithm.

By comparison with protothreads, each component having its own stack enables local state to be saved and restored across a context switch, as well as removing the need for unnecessary yield statements to be inserted into the behaviours of components by the Insense compiler.

5.4 Architectural Specific Considerations

The implementation of InceOS is such that it is independent of the hardware platform upon which it is operating; however, as is often the case, there are a number of exceptions.

5.4.1 Context Switching

The act of switching between different components is inherently specific to the architecture which is being used. On the t-mote sky, a context switch is quite simple by comparison with an architecture such as the x86.

[2] A time quantum refers to the maximum continuous amount of time a component may execute for before being pre-empted

First, all of the sixteen registers on the msp430 micro-controller, which is used on the t-mote, are pushed onto the stack. Then the values inside the stack pointer and program counter, registers 1 and 0, are saved into the `pc` and `sp` fields of the CCB for the currently running component. Then the stack pointer for the new component to run is copied from the `sp` field in its CCB into register 1 and the `pc` value is pushed onto the stack. A `RET` instruction is then used, which will place the first value on the stack (program counter) into the program counter register and continue execution from that point. Unlike other architectures, such as the x86, there is only one stack register. On an x86 platform there are two such registers, one of which points to the top of the current call frame and one which points to the bottom. The component that has just been switched in must now pop all all the registers which were previously saved on the stack when it was switched out. At this point the switched in component is now running on its own stack, and after returning from the schedule system call, plus any function or interrupt that invoked the scheduler, it will resume execution.

When a component is created and first switched in, the `pc` field in its CCB is set to a wrapper function that is used to start a new component, thus avoiding any problems with a new component trying to pop registers from its stack which do not exist. The wrapper function is also used to re-enable pre-emption when the new component is executing. After some post-implementation reflection on the presence of essentially two program counters within a CCB, it was decided that the pc of the behaviour could be placed on the stack during creation. Within the wrapper function this value could simply be jumped to, with the address of the *this* pointer placed on the stack. In this way, the need for two program counters in the CCB would be removed and some of the overhead of the initial function call to the behaviour would be removed. The option to keep this function call was considered, as a component exit could be placed after the call, ensuring that a component would always exit, however it was decided that some system elements did not require this functionality, and that the compiler would always place a `component_exit()` call after the end of a behaviour loop.

Having to return through function calls or an interrupt at the end of a context switch is inefficient, both in space and time. A possible solution would be to store the return address and current stack pointer of the entity which is invoking the schedule function. These values could then be stored inside the CCB fields and used in the context switch, resulting in the switched in component resuming execution from immediately after the initial invocation of the context switch. The program counter, which is saved on the stack, could be accessed directly as the OS code is static, meaning that its position on the stack could be calculated. Also, the stack pointer value could be calculated due to the static stack depths of the OS code. The only issue with this would be which registers should be saved, requiring that on entry to the scheduler all registers are saved, regardless of whether or not a context switch occurs.

However, the degree of this inefficiency is small. Considering that only a system call or interrupt can cause a context switch, execution would have to return to another location within the OS. This is due to the conditional nature of the rendezvous mechanism and its implementation. For example, if a component blocks as there is no-one to send to, it must return after the block so as the number of bytes sent is returned to the Insense application, see Chapter 6. This would save exactly 6 bytes of space, scheduler call frame, and 10 CPU cycles (0.4 μs). In the case of pre-emption, the saving would be greater as the interrupt context would not need to be explicitly removed, saving 20 bytes and costing 27 cycles (1.08 μs)

Protothreads do not have context switching in the same sense as described due to their stack-less non-pre-emptive nature. As all protothreads must explicitly block or yield, Contiki needs only to select the next function pointer to call from its list of protothreads.

5.4.2 Stacks

As discussed previously in Section 5.3, pre-emption is made possible by each component having its own stack. The size of this stack is determined at compile time by the Insense compiler and passed as a parameter to the `component_create()` system call. However, this value is the amount of stack space required to run the C implementation of the Insense application; it does not take into account the stack space required by InceOS, specifically the space required by the system calls, interrupt handlers and context switching.

In order to accommodate the extra space required, the maximum possible stack depth was determined. This value was then added to the size of the stack when allocating a CCB for a component. This is an architecture specific element due to the fact that different architectures may require different stack sizes depending on the size of the types that it uses; for example, the size of an `int` on the t-mote sky is sixteen bits, however on the Imote2 [32] it is 32 bits. This is due to the word sizes of the respective systems being 16 and 32.

Also, during a context switch all of the registers are saved on to the stack. Again different architectures will have varying numbers of registers of different lengths.

5.4.3 Interrupts and Pre-Emption

InceOS is a pre-emptive system, as noted earlier. Consequently, some situations which require serialised access to sections of code must be protected by use of a lock. It has been explicitly stated that Insense does not use mutexs or semaphores and that InceOS does not provide them. However, this lock is not a mutex or semaphore, in fact it is simply a status flag. Whenever a critical section is entered, all hardware interrupts are disabled, by clearing a bit in the control register of the msp430, clearing the value in the status flag, and restoring the hardware interrupts, by resetting the bit in the control register. When leaving a critical section the process is the same, except the status flag is set to true.

This status flag, known as the pre-emption flag, is tested every time the timerA interrupt handler is invoked. If the flag is true, the scheduler is invoked, swapping out the current component and swapping in the next. If the flag is false, the interrupt does not invoke the scheduler, but can still perform any other functionality required by the timer.

By only disabling pre-emption and not disabling all hardware interrupts for critical sections within the OS, interrupts such as timer expiration notifications and incoming radio packets can still be generated and wake the relevant component, but do not pre-empt the currently running component. This feature is only available to, used by, and required by the OS itself.

5.5 Component Death

When a component is to be destroyed, all memory that has been allocated to it must be returned to the heap, and any channels which have been associated with the component made eligible for use again. The memory to be returned includes the memory allocated for the *this* pointer and the component's CCB, which includes its stack. As each component has its own stack, it is not possible

to deallocate this stack with a function that is running on that stack, consequently a component can not destroy itself.

Instead, disposing of a component is split into two sections. Firstly, a component will call `component_exit()` which will set the component's state to `KILLED`, after which the component yields itself. At this point, the CCB representing the component would normally be placed at the end of the run queue, however the CCB's status is noted as being set to `KILLED` and the CCB is placed in a separate *suicide* queue, also known as limbo. From here scheduling continues as normal, with either the next CCB taken from the run queue or the node being put to sleep. In either case, once the next component to be run is selected, the context switch will occur. At this point control would normally return the behaviour loop of the component, however before this the suicide queue is serviced, and any CCBs on this queue are disposed of. Specifically, this consists of *free*ing the space allocated for the *this* pointer structure, stack, and CCB itself, as well as calling `channel_destroy()`, Section 6.3.2, for each channel in the `chan` array.

Chapter 6

Channels

This chapter discusses the design and implementation of the channel abstraction within the OS and how it is associated with the channel concept in the Insense language.

6.1 A Channel

Within InceOS a channel has two representations. The first is used to represent a channel as seen by the compiled Insense application, and the other if for use within the OS.

Insense applications use a channel definition as follows : `typedef int Channel;`

Whereas, within the OS a channel is represented by a C `struct`, as shown in Listing 6.1.

```
typedef struct channel {
    unsigned char dir_use;       //Indicates if a channel is in use
                                    and its direction
    unsigned char num_conns;     //Number of bindings
    CCB *my_component;           //Owning component
    Channel conns[MAX_LINKS];    //List of bindings to channels
    void *message;               //pointer to message buffer
    unsigned int message_len;    //length of message buffer
}CHAN;
```

Listing 6.1: InceOS representation of a Channel

Two separate representations were chosen to ensure that only the OS is be able to manipulate the underlying channel representation, removing the possibility that the Insense application may, accidentally or otherwise, modify it.

Within the OS an array of `CHAN`'s is statically allocated, the size of which represents the total number of channels that the OS can support. The `Channel` type serves as an index into this array,

enabling the Insense application to maintain a handle on a channel for use with the system calls defined below, but without needing a handle on the underlying representation.

Section 3.1.1 describes the different features that are associated with channels, such as direction, and these are reflected in the fields of `CHAN`, which are briefly described in Listing 6.1.

As a channel is accessed via the `Channel` index, it is possible that an incorrect or malicious value may be used instead of a legal one. It is the job of `dir_use` to indicate whether or not the value attempting to be indexed is currently in use or not. `dir_use` is used as a bit field with the second right most bit (000000**1**0) used to indicate if the channel is in use or not with a 1 or 0 respectively. This field is also required to specify the direction of this channel is a similar manner, in this case using the right most bit (0000000**1**) . The OUT direction is specified with 0 and IN direction with a 1. The use of C bitfields was considered to store these values as they would only consume 1 bit of space each, however the C compiler does not give guarantees of the order in which such values are placed within the struct, which could lead to errors.

The pre-InceOS system creates channels as they are required and does not use (or require) any field to represent if a channel is in use, as they all are. It does specify the direction of the channel, however this is a pointer to a string requiring 2 bytes of space. It is worth mentioning at this point that the pre-InceOS system also specifies the type of data that a channel conveys. This is not done in InceOS as it was decided during the design phase that this responsibility would be delegated to the Insense compiler.

The number of channels that any particular channel is bound to is specified by `num_conns`. This is associated with the array `conns` which specifies the indices of the channels to which this channel is bound. This is similar to the situation found in a CCB. Within Insense, each channel is associated with a component as was explained in Section 5.2.2. Within a `CHAN`, this association is represented by a pointer to the associated component: `my_component`.

The pre-InceOS system made excessive use of buffering during data transfers over channels, see Section 1.1.3. Within InceOS no buffering takes place during data transfer, instead the OS requires that an Insense application pass a pointer to a buffer that it has allocated, as well as the length of this buffer. This is similar to the style of communication used in Berkeley Sockets [30]. When data is transferred, it is copied from one buffer to the other, see Section 6.3.9. It is `message` and `message_len` which are used to store the pointer to the buffer and its size respectively within a `CHAN`. By facilitating message communication in this way, the need for multiple internal buffering is removed and replaced by a more intuitive style. It saves on the amount of space required, memory fragmentation, and the complexity of the actual transfer with regards to the actual location of the data.

As Listing 6.1 shows, the number of bindings that a channel may have is limited to `MAX_LINKS`. This decision was again a combination of analysis of example Insense applications, and an attempt to lower RAM occupancy. Also as before, the following system calls are parametrised to be independent of the value of this variable when checking for legality. At present this value is set to 8. As an aside, it is worth noting that the layout for the `CHAN` and `CCB` were chosen such that they would enable the best compacting of the data types into the smallest possible space.

6.2 Configuration

As noted in Section 6.1, a statically allocated array is used to represent the maximum number of channels that are offered by the system. This is in contrast to the pre-InceOS system implementation where each new channel associated with a component is dynamically allocated when required and maintained in a linked list. By using static allocation as opposed to dynamic allocation, memory fragmentation is significantly reduced. Whenever a channel is created in the pre-InceOS, Contiki is asked for a pointer to the start of a memory location (from the heap), as big as the channel being created, to represent the channel. When the time comes to destroy the channel, the memory it occupies is returned to the memory allocation system within Contiki. During the lifetime of the channel, the structure of the heap will have changed meaning that there is no guarantee that the returned memory will align with another free memory location. The result is a fragmented heap, meaning that when an allocation request is made for n bytes, n bytes may be available but they are not contiguous, and therefore the request is denied. This is an expected problem with simple memory allocators, such the one used in Contiki, however the problem is not helped due to the comparatively small size of a channel. In contrast, by preallocating the maximum number of channels required, which could be determined by static analysis at compile time, channels will take up space within the bss section of the compiler output and not fragment memory. This style of pre allocation is used for other elements of InceOS as expanded upon by the following sections and chapters. It is also a well known system developer's "trick of the trade".

The size of the channel array is governed by the `NUM_CHANNELS` constant. The implementation of the following system calls are independent of the value that this constant can take. At present an unsigned 1 byte (8 bit) value is used to hold an index into this array, which limits the total number of channels to 255, with the 256th value being used to indicate that an entry in a `CHAN`'s connections are unused (`UNUSED_CHAN`), explained in Section 6.3.3. This decision was made to save space, considering the constrained hardware environment of the t-mote, and by examination of the number of channels being used by Insense applications. However, should the number of supported channels need to increase, only four changes need be made to the header file of the channel module. Originally another value was reserved to indicate that a channel had blocked while attempting to perform a data transfer action on a channel with no connections. However, this was not required as the Insense language dictates that a component must remain blocked until a data transfer completes, making blocking on a channel with no connections not a special case, discussed in Section 6.3.9.

By comparison, the pre-InceOS system uses a linked list associated with every component to store the channels which are created as required. In terms of access times it is O(n) for searching which is required every time a channel operation is performed, however it is O(1) for insertion. InceOS offers O(1) access times when performing a channel access, as the Insense application will provide an index, however when creating a channel the array must be searched, offering O(n) complexity.

6.3 System Calls

The following section enumerates and describes the system calls that are associated with channels. Listing 6.2 shows the prototypes for these system calls.

```
Channel         channel_create(enum direction d)
void            channel_destroy(Channel id)
 int            channel_bind(Channel id1, Channel id2)
void            channel_unbind(Channel id)
Channel         channel_duplicate(Channel id)
int             channel_adopt(Channel id)
int             channel_send(Channel id, void *buffer, int len)
int             channel_receive(Channel id, void *buffer, int len)
int             channel_select(struct select_struct *s)
```

Listing 6.2: InceOS System Calls for Channels

6.3.1 Channel Create

This system call creates a new channel and has the prototype:
`channel_create(enum direction d)`,
where the parameter `d` indicates the type of the channel to be created: `IN` or `OUT`.

As previously stated, an array of `CHAN`'s is statically allocated, meaning that no actually memory allocation is required when creating a channel. Instead, the `channel_create()` system call is tasked with first checking that the creation is a legal action, selecting a `CHAN` for use as the channel, and then initialising it.

Ensuring that the creation is legal requires two checks. The first ensures that the component, with which the channel will be associated, has space to store the index of the channel to be created, see Section 5.1. The second ensures that there is a free `CHAN` within the statically allocated array. If either of these conditions fail, -1 is returned to indicate an error.

If a free `CHAN` is found then it is initialised. This consists of first setting it to be in use; assigning it the direction of the parameter `d`; setting the number of connections to zero; and setting channels in the `conns` array to be unused. It is not required to initialise the buffer pointer and length field as the send, receive, and select semantics prevent a situation where these fields would ever be in an erroneous state before being used by the respective functions, as discussed later. The final action is to associate the channel with a component, however this may not always be the component creating it. A channel may be created within a constructor function, in which case the channel should be bound to the component which is being created. In order to make this distinction, an internal function within the kernel, `component_in_create()`, is used to determine whether the new channel should be associated with the current component, or the component being created. This check is performed before the first legality test mentioned above. Once this is done, the index of the `CHAN` is placed into the array of associated channels in the CCB for the appropriate component. The final step is to return the index of the `CHAN` to the caller, thus indicating a successful channel creation.

6.3.2 Channel Destroy

This system call destroys a channel and has the prototype:
`channel_destroy(Channel id)`,
where the parameter `id` indicates the index of the `CHAN` to be destroyed.

By comparison with creating a channel, destruction of a channel is relatively simple. The first task is to unbind any connections to or from this channel. This is achieved by calling the `channel_unbind()` system call if there are any bindings, discussed below. The next step is to disassociate the `CHAN` from the component and then set it as being unused.

The pre-InceOS system has no explicit mechanism for the destruction of a channel, it instead relies on deallocating the memory that was initially assigned for the channel when its owning component is destroyed, as discussed in Section 4.1.3.

6.3.3 Channel Bind

This system call binds together two channels and has the prototype:
`channel_bind(Channel id1, Channel id2)`,
where the parameters `id1` and `id2` indicate the indices of the channels to be bound.

Within InceOS, the act of binding two channels together is a two stage process. The first ensures that the parameters are legal indices, and that the channels themselves are allowed to form another binding, i.e. they have not exceed the maximum number of allowed connections for a channel, and that they are not already bound to each other. If any of these conditions are violated, then -1 is returned to the caller indicating an error.

If the validation is successful, the next step is to update the `conns` array of the involved channels with a reference to the other, as well as incrementing the `num_conns` field. The value 1 is then returned to indicate success.

When binding two channels in the pre-InceOS system, a pointer to each channel is inserted into a linked list in the other. This list is used to maintain the channels to which a particular channel is bound. The system then alerts the protothreads, who are controlling the components which own the channels, that a binding has been made, by generating an event specified in the runtime. This is required as the event will cause any component that blocked due to a data transfer to resume its transfer. This will be expanded on in Section 6.3.9.

6.3.4 Channel Unbind

This system call removes any binding between a channel and its peers, as well as removing any connection a peer has to this channel. It has the prototype:
`channel_unbind(Channel id1)`,
where the parameter `id1` indicates the index of the channel to be unbound.

As before, this is a two stage process. First, the index which is passed as a parameter is validated, and then the unbinding begins. This is carried out in two nested loops. The logic is that for each

binding, remove the link that the bound channel has to the current channel, and then mark that space in the channel's connection array as unused. Finally, reset the num_conns field to zero. It is worth noting that the act of unbinding a channel simply resets it and is distinct from destroying a channel completely.

It is worth noting that it is the bind and unbind functions that maintain the connection lists for channels, enabling the send, receive and select calls, discussed below, to simply use these values and not modify them. This is why a component may block on a channel with no connections without considering it a special case.

A similar approach is taken within the pre-InceOS system.

6.3.5 Channel Duplicate

This system call creates a duplicate copy of a channel, and has the prototype:
channel_duplicate(Channel id1),
where the parameter id1 indicates the index of the channel to be duplicated.

Unlike the other system calls, channel_duplicate() and channel_adopt() do not appear obvious by looking at the Insense language. Insense allows a channel to be sent across a channel, and once it has been received by a component, the channel may be used as normal. It is also the case that if a channel was bound to other channels before being sent, then it should still be so bound after it has been sent. In order to facilitate this requirement, these two system calls were created.

To duplicate a channel, a new one is first created. Then all the fields, as shown in Listing 6.1, are copied, and the index of the new channel returned to the user ready to be sent across the channel. There are two exceptions to this. Firstly, the my_component field in not set. This is because it is expected that this channel is being duplicated to be sent across a channel. As a result the adopt call will handle the binding of a channel to a component. Secondly, the new channel must be bound to all of the connections of the previous channel. In the event that a binding can not be made the allocated channel is destroyed, and -1 is returned to indicate an error.

It is worth noting that this style does not currently support channels being sent across the radio, due to the fact that it is the index of the channel which is sent. The node that receives this index does not have a copy of the list of channels with which the index is associated. This is also a symptom in general of the fact that radio communication is still explicit in InceOS. When the radio is abstracted over, enabling inter-node component level communication, this problem would be solved by the OS.

6.3.6 Channel Adopt

This system call binds a channel to the calling component, and has the prototype:
channel_adopt(Channel id1),
where the parameter id1 indicates the index of the channel to be adopted.

As alluded to previously, this system call is used to associate a duplicated channel with the calling component. It is expected that this system call will be used when receiving a channel from a data

transfer across a channel.

The pre-InceOS system assigns this received channel to one of the fields within its *this* pointer; it is the responsibility of the Insense compiler to handle this. As an aside, in order to send a channel over another channel it must first be cast to an `any` type, equivalent to a C void*. This is due to the fact that the pre-InceOS system currently enforces typed channels, and a channel by itself is not a type in the language. Upon receipt of a channel, a component must *project* or cast a channel to return it to being a typed channel, in terms of the type that the channel conveys. This is not a concern in InceOS as it relies on the compiler to enforce typed channels, all that InceOS requires is that the buffer that the data is going to is large enough to hold the channel index.

6.3.7 Channel Receive

This system call attempts to receive data from any of its bound channels, or block if no data is available. It has the prototype:
`channel_receive(Channel id, void *buffer, int len)`,
where the parameter `id` indicates the index of the channel to be received on, `buffer` is the space to hold the incoming datum, and `len` is the length of the buffer.

This system call is more complicated that those previously explained. This is because it does more than modifications to data types; it, along with the following two calls, is used to implement part of the communication semantics used by Insense: the blocking rendezvous model, Section 3.1.2.

This is the first, and simplest, of the data transfer operations in InceOS. It is used to receive data from any of the components that are currently, or will be, sending data on an `OUT` channel, which is bound to the `IN` channel specified by `id`. The receive call, as with the others, is a blocking call and will either successfully transfer data from the senders buffer to its own, enter a blocked state, as shown in Table 3.1, or generate an error.

Within this, and the subsequent system calls, errors can occur from incorrect user-specified values and buffer sizes. In order to save repetition, these issues are addressed in Section 6.3.11.

The first operation when receiving is to ensure that the channel being received over is legal, in terms if being in use and being the right polarisation. Once this is complete a reference to the buffer and its length are stored, this is required in case the component needs to block, explained below. At this point there are two possibilities.

The first is that the channel has no connections, indicated by the `num_conns` field in the `CHAN`, and therefore no one to accept data from. In this case the system call will set the `blocked_channel` field of the CCB to the channel it is receiving on and then block, placing the receiving component into the `BLOCKED_RECEIVER` state. When this component is rescheduled it will begin executing immediately after this point. At this point it must decide what value to return to the user, and this is dependant on its state. If the component was unblocked by a send request, Section 6.3.9, then the number of bytes which were received into the buffer will be returned. However, if the component was unblocked by an interrupt then the value of -2 is returned to indicate this.

The second possibility is that the channel has one or more connections. Each connection is then checked to ascertain if it is an eligible candidate for data transfer. This is determined by two factors. First, the component which owns the bound channel must be in the `BLOCKED_SENDER`

state. Second, the `blocked_channel` field of this component's CCB must contain the bound channel. The reasons for this are noted in Section 6.3.9. Each connection which is eligible is added to a list of eligible channels.

After each connection has been examined there is still the possibility that no eligible connections were found, in this case we revert to the first situation. Otherwise a non deterministic choice is made from the list by using a random number generator. The data is then transferred from the sending component's buffer to the receiving component's buffer, the sending component unblocked, and the number of bytes received returned to the caller.

6.3.8 Channel Select

This system call attempts to receive data from any of its specified channels, or block if no data is available. It has the prototype:
`channel_select(struct select_struct *s),`
where the parameter `s` is a pointer to the select data structure.

This system call provides support for the select feature of the Insense language, Section 3.1.2. In order to specify the conditions, channels, and default selection, a data structure called `select_struct` is used and can be see in Listing 6.3.

```
struct select_struct
{
    int nchans;                    //number of channels
    Channel *chans;                //list of channels
    int whensANDdef;               //bitmask indicating the when
                                   //clauses and default clause
    void *buffer;                  //ptr to the receiving buffer
    int size;                      //size of the buffer
    int *len;                      //number of bytes transferred
};
```

Listing 6.3: Channel Select Data Structure

This structure is used instead of passing the variables as parameters, due to an error with compiler for the msp430 in which parameters were not correctly accessed in the new call frame of the select function. This is an unresolved issue with the size optimisation level which attempts to reduce the size of the binary file which it produces. The structure itself is used to indicate the number of channels to be selected over, as well as the channel indices themselves. A bit mask is used to indicate which channel guards are true and which are false, with the bit in the nth position from the right indicating the guard value for the nth channel. The left most bit is used to indicate whether of not a default clause is specified, meaning that if no eligible channels are found, the system call should return the number of channels to the caller, thus indicating to the caller to invoke the default clause. The `buffer` and `size` fields are used to hold a reference to the data buffer and its length, as specified by the component. The final field is used to specify the number of bytes transferred, as explained below. An example of this particular system call can be seen in Appendix B.

Once in the system call, a list of eligible channels is constructed. At this point eligibility is based on each channel's associated guard value. If no channel's are eligible and the default value is false, -1 is returned to indicate a programming error. Otherwise, if there are no eligible channels and the default value is true, then the number of channels is returned, as mentioned above. If there are eligible channels then they must each have all their connections queried to ascertain if data is ready to be received across them. This is a similar process used in the above system call, although it is broken into two stages.

The first is used to determine which of the channels being selected over are ready. The second will non-deterministically choose which of these channels will be used, after which another non-deterministic choice is made of that channel's eligible connections, according to the requirements in the above section. The data is copied from the sender's buffer into the receiver's buffer, and the sending component unblocked. However, unlike the send and receive calls, the select call returns the position in the `chans` array, within the `select_struct`, of the channel received over. The number of bytes transferred are stored in the `len` field of the `select_struct`. The enables the caller to know what channel was selected, and how many bytes were transferred.

If no channels are found to be ready to send their data and the default value is false, the component must block, otherwise it will return the number of channels to be selected over, indicating the default was chosen. If the component is to block, then it must do so while waiting on up to 20 channels, the maximum number of channels a component may have. Another data structure known as `SELECT_DATA` is used to enable this, and can be seen in Listing 6.4.

```
typedef struct select{
    struct select *next;      //pointer to next list member
    unsigned char nchans;     //Num of Channels being selected over
    unsigned char *chans;     //Array of select channels
    unsigned char pushing_channel; //Channel which pushed the data
    int *written_bytes;       //Number of bytes written
}SELECT_DATA;
```

Listing 6.4: Blocking Select Data Structure

This data structure is maintained within a linked list, and so the first field is used to point to the next element of the list. The following two fields are used to note the number of channels that a component has blocked selecting over, and a list of these channels. The fourth field is used to indicate the channel that pushed the data, this is required to identify which of the channels being selected over was chosen. The final field indicates the number of bytes which were written during the data transfer.

As with the array of `CHAN`'s, an array of these structures are statically allocated, however as they do not need to be accessed individually, they are stored in a linked list. Whenever one is required it is taken from the list, and when it is finished with it is replaced. If more are required than available, then more are allocated from the heap.

Before entering a blocked state, the system call will first obtain, and populate one of these structures with the channels where the guard values were true. It will then place a reference to this structure into the `select_ptr` field of the CCB of the currently running component, as required for the

following send system call. After this is complete, the call will block the current component, entering it into the BLOCKED_SELECT state.

Once the component is unblocked, it will return the SELECT_DATA structure to the linked list to be reused. If the component was awoken by a system call, then the position of the selected channel and the number of bytes transferred are returned to the caller as described above. If the component was awoken by an interrupt, -2 is returned.

6.3.9 Channel Send

This system call attempts to send data from any of its bound channels, or block if no data is available. It has the prototype:
`channel_send(Channel id, void *buffer, int len)`,
where the parameter `id` indicates the index of the channel to be sent over, `buffer` is the space to hold the outgoing datum, and `len` is the length of the buffer.

Before attempting to send, the pointer to the data (buffer) and the data's length (len) are placed into the `message` and `message_len` fields of the `CHAN`. This is done to enable data to be copied directly from the buffer provided to the send call, into the buffer provided by the receive or select calls above. This method is used as it does not require the OS to store any duplicate copies of the data, thus saving space. In particular this overcomes one of the major limitations with the pre-InceOS system, as discussed in Section 1.1.3

When sending data over a channel there are two possible legal situations: there are no channels bound to it (no connections), or there are 1 or more bindings (connections).

In the first case, the `blocked_channel` field of the CCB for the sending component is set to the index of channel being sent over, and the component placed into the BLOCKED_SENDER state, Section 5.1. This will indicate to any subsequent select or receive calls that this component is ready to send data across the specified `blocked_channel`.

In the case where there is one or more connections, it is required to ascertain which of the connections are eligible to receive data. This is established by the state of the component associated with the receiving channel, specifically it must either be in the BLOCKED_SELECT state, or in the BLOCKED_RECEIVER state.

If a channel is found with a component in the BLOCKED_SELECT state, then the SELECT_DATA structure, which is referenced by the `select_ptr` field in the component's CCB, is queried. If any of the channels contained in this structure form a connection with the channel being sent over, then their indices are added to a list of eligible channels.

If a channel is found with a component in the BLOCKED_RECEIVER state, and its component's `blocked_channel` field is the same as the receiving channel, then the channel is added to the same list of eligible channels as above. This is due to the fact that there is no preference or weighting between components that have blocked in a receiving state, or in a selecting state. The need for the receiving (or sending) component to have blocked on a particular channel is required as a component may have more than one channel; even thought it is in a blocked state, it may have entered that state due to another channel, as shown in Figure 6.1.

In Figure 6.1, X would be the component executing a send, Y would be the channel being examined, and the error would be caused by Y trying to receive from Z (channel D), not X (channel A). Here Y would be placed in the BLOCKED_RECEIVER state, but not due to X. The blocked_channel field over comes this problem.

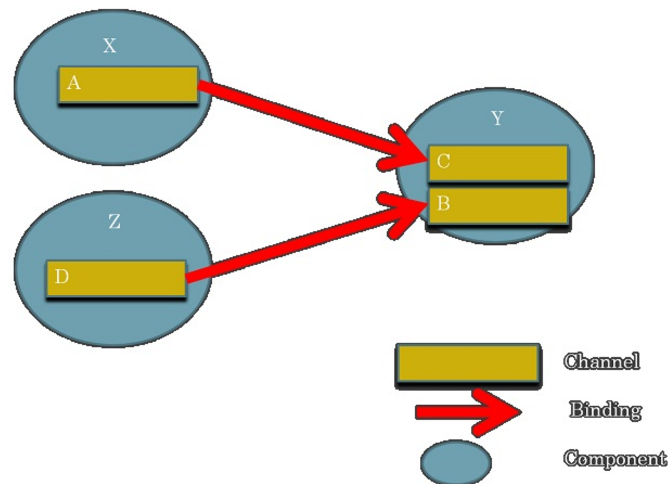


Figure 6.1: Component With Multiple Channels

If a component meets these criteria for being eligible, then the index of the channel it is blocked on is added to a list of eligible connections.

After all connections have been examined, there is the possibility that no eligible connections were found, and the eligible list is empty. In this case the procedure is the same as if the channel were not bound to any other channels, as described above.

If eligible connections were found, an index from the eligible list is chosen at random by using a pseudo-random number generator, thus meeting the non-deterministic requirement for channel selection from Insense. The data to be sent is then copied to the buffer supplied by the receiving or selecting component, and the component unblocked. The send call will then return the number of bytes transferred to the caller. There is an extra requirement for components in the BLOCKED_SELECT state. The index of the sending channel must be placed in the pushing_channel field of the select structure to indicate which channel pushed the data, as described in Section 6.3.8.

6.3.10 Multicast Send

This system call attempts to send data from any of its bound channels, but does not block. It has the prototype:
`channel_multicast_send(Channel id, void *buffer, int len),`
where the parameter `id` indicates the index of the channel to be sent over, `buffer` is the space to hold the outgoing datum, and `len` is the length of the buffer.

In some situations, the blocking semantics required by Insense do not fit the needs of system components, such as the timer, rtimer or radio, Chapter 7. Consequently, another channel system call is available, but **only** to system components, not Insense applications. This function is analogous to the above send call, the main two differences being that this system call will never block, and it will send its data to every connection which is ready to receive it. Also, instead of returning the number of bytes transmitted, it will return the number of successful sends which were completed. The reasons for this call are discussed in the following chapter.

The runtime does has similar functionality, however it is not an isolated function, rather it is integrated into the implementation.

6.3.11 Channel Errors

Within InceOS, should an error occur for any reason within a system call, then the value -1 is returned to the caller. For data transfer functions, errors can be generated by a number of situations. One reason previously mentioned is attempting to perform an action on an illegal channel index. It is also the case that preforming a send on an IN channel, or performing a receive or select action on an OUT channel will raise an error.

A more fundamental error is specifying a buffer which is too small, compared to the size of the buffer on the other side of the channel. For example, if a receive request blocks trying to receive with a buffer 10 bytes long, then an error is generated if a send request is attempted with a buffer of 9 bytes or less. The same is true if a component, which is executing a receive or select, specifies a buffer smaller than that of the sender. Again, if this situation occurs, then -1 is returned to the caller. It is important to note that the component which has blocked is unaffected, and remains in a blocked state ready to have its data transferred. In this way, the blocked component specifies the size of the buffer, even if it should have specified a larger buffer to facilitate the Insense program. This is a situation where the Insense compiler must ensure that it generates code to ensure the correct buffer sizes, particularly as InceOS delegates the responsibility of typed channels. If user-specified code within the behaviour of a component attempts to populate a buffer with too much data, then a buffer overflow will occur. This code is out-with the jurisdiction of the OS, and such a buffer overflow will have unknown side effects.

Due to the fact that the select call relies on the `SELECT_DATA` structure when blocking, if there are no structures available to use, the select call will attempt to dynamically allocate a new one. If there is no memory available to allocate a new structure, then the call will return -1.

Chapter 7

System Components

This chapter discusses the services that the OS provides. These includes the two timers, sensors, debug, dynamic memory allocation, and the radio. They are different from the system calls described in the previous chapters because the system calls are used to facilitate the requirements of the languages, whereas this chapter details the services which are available to the Insense applications themselves. In particular, the timer, rtimer, sensor, button and radio components are the well-known components which an Insense application has knowledge of.

7.1 Timers

The t-mote sky platform offers two timers: a one second timer (timerA) and a more fine grained millisecond timer (timerB). In order to provide these services to an Insense application, each timer has an associated component.

7.1.1 Timer Component

The component associated with timerA is simply the *timer* component.

Requests are made of the timer component by an Insense application which sends a request message along a channel to the timer component. The layout of this message can be seen in Listing 7.1. The Insense application must specify the duration of the time interval that it requires, the channel upon which it will listen for the *tick* from the timer component, and whether or not this timer request is periodic. Due to the larger granularity of this timer, compared to timerB, it is used as a general purpose timer in Insense applications.

When an Insense application specifies a duration for a time interval, it is specified in terms of multiples of `CLOCK_SECOND`. This is a value available to any Insense application and specifies the number of clock ticks which timerA makes in a second (32). The multiplicative factor can be less than one, resulting in a values less than 32. However, as this value is an integer it will use integer multiplication, accordingly the result will be an integer.

```
struct timer_request{
  clock_time_t interval;   //the duration of the time out
  Channel chan;      //the channel that the tick will be sent over
  unsigned char periodic;  //if it should be a periodic tick
};
```

Listing 7.1: Timer A Insense Application Request Structure

Within the timer's behaviour loop, the first action is to listen on its request channel. In order for an application to make a timer request it must first bind one of its channels to the timer's request channel, after which it may send a `timer_request` on that channel to the timer. The timer component's channels, as with all system components, are well known. The timer will then attempt to store the received request in the outstanding timer queue, which is sorted by the order of earliest deadline. If this is successful, then the timer will attempt to bind one of its channels to the channel that was passed along in the `timer_request`. This is required to guarantee that only the component who made the request will receive the tick from the timer. If many request channels were bound to a single timer output channel, then it could not be guaranteed that the correct component would receive the correct tick at the correct time.

If there is no space for another timer in the list, then the channel passed in the request is bound to a special error output channel and an error value of zero is sent to the requesting component, indicating an error. If there was space but the request channel could not be bound to, then the request is removed from the list and no further action can currently be taken, as the timer has no link to the requesting component. Within the timer component, the list of outstanding requests is constrained by the maximum number of channels which may be associated with a component: `SCHED_CHANNELS`. This is because of the constraint that each component must have a single connection to the timer, as explained above. This number is further constrained by two as there must be a dedicated channel used for incoming requests and one for dedicated error reporting. Should a greater list be required in the future, this component could be considered a special case and more channels added, however this is not currently required. After the completion of adding a new request, the timer must then update a static variable which is used to hold the absolute deadline of the most imminent outstanding timer request. As the timer list is sorted, this value is simply the deadline of the request at the head of the timer queue. After this, the component will return to listening on its input channel.

As an aside, in an attempt to save on heap fragmentation and access times, the timer requests are statically allocated in an array. These elements are then used to populate a linked list of *free requests*. When a timer is used to hold a request, it is added to another linked list of *used timers*. It should be noted that these timer requests are different from those that are sent from an Insense Application, see Listing 7.2. This different structure is required for periodic requests, where both the deadline and interval must be stored.

```
typedef struct timer_entry
{
    struct timer_entry *next;     //Next element in list
    clock_time_t deadline;        //Deadline for this timer
    clock_time_t interval;        //Interval for this timer
    char periodic;                //Timer's periodic flag
    Channel my_chan;              //Channel to send tick on when due
}TIMER_ENTRY;
```

Listing 7.2: Timer Internal Bookkeeping Structure

As explained in Sections 6.3.7, should a component be unblocked by a hardware interrupt it will return -2 instead of the number of bytes which have been written to it. It is by this method that the timer component is able to differentiate between an incoming request and a timer's expiration. If the value is -2, indicating an expired timer, the timer component will first record the time. It will then compare this value to the first element of the timer queue. If the deadline of the first timer is less than or equal to that of the current time, then a tick is sent along the singly bound channel to which the requesting component should be listening. This channel is then unbound, and the timer request is removed from the list. If the request was a periodic one, the channel is not unbound and a new deadline is calculated, as well as adding the timer request back into the outstanding timer queue. This procedure will be repeated along the timer queue until it is empty or a timer is not due. In the latter case the static variable is updated with the due time of the next timer.

It is important to note that a special send system call is used when notifying the requesting component of either a tick or error known as `channel_multicast_send()`, Section 6.3.10. This is required as it is not acceptable for a system component to block waiting to send; this would prevent requests from other components being serviced. In the eventuality that zero successful data transfers are made, indicated by the `channel_multicast_send()` returning zero, the timer entry's channel will be unbound from the receiving components channel and replaced in the free list. This is done to allow for the eventuality that a component has been destroyed before being able to receive the timer tick. However, this does leave open the situation where a component starts waiting for a tick after this happens, in which case it will indefinitely block. At present, this has not yet happened, however it could be prevented by ensuring a component will not be pre-empted between requesting the interrupt and listening for it.

In order to cause an interrupt to unblock the timer component, code was placed in the interrupt service routine for timer A, just as for pre-emption. Whenever a timer interrupt occurs, two checks are performed to decide whether or not the timer component should be unblocked. The first check is to ensure that there is a timer request present on the timer queue. As the queue is a linked list, this check will simply ensure that the head is not NULL. The second check will query the static variable which holds the outstanding deadline to decide if it is less than the current time. If both of these conditions are true then the timer component is placed at the head of the schedulers run queue, as discussed in Section 5.2.3.

7.1.2 Rtimer Component

The component associated with timerB is simply the *rtimer* component.

This timer is used more for high precision timing, for example specifying the sleep periods of the radio component.

The logic involved for the rtimer is substantially similar to that of the timer component, and so to avoid duplication it will be omitted. However, there are a few differences to note. Firstly, the rtimer does not support periodic timer requests, as reflected in the different request structure used, Listing 7.3. Consequently, the internal representation only requires the deadline and not the interval, unlike the timer component. This is due to the way in which this timer is used as a one shot timer, not a periodic one.

```
struct rtimer_request{
  rtimer_clock_t interval;       //the duration of the time out
  Channel chan; //the channel that the tick will be sent over
};
```

Listing 7.3: Rtimer Insense Application Request Structure

Secondly, when an Insense application is specifying a duration for the time out, it is specified in terms of multiples of `RTIMER_SECOND`. This is a value available to any Insense application, and specifies the number of clock ticks which timerB makes a second (4096). Thirdly, due to the more responsive nature of timerB, the maximum size of the timer queue is just five timers. This value may be changed by updating one variable, however at present there has been no need to increase this value. The same situation applies as for the above timerA in relation to multiplication of `RTIMER_SECOND`.

The pre-InceOS system only offers the one second timer to Insense applications via a function call, not over channel requests. This call is for a periodic timer request. There is a similar style structure to InceOS used, with a list of timer requests which is added to by new requests, and processed after a timer interrupt. This function interacts with the existing timer system in Contiki, and is notified of timer events via an event.

7.2 Sensors

Section 1.1.1 explained that a popular use for motes is for environmental monitoring, which is achieved via the sensors on a mote. On the t-mote in particular there are eight: a visible light sensor, a total solar radiation sensor, a temperature sensor, a humidity sensor, a radio sensor, an internal voltage sensor, an internal temperature sensor and a button sensor.

The first seven sensor requests are serviced by a single component. This component has two channels for each sensor, one for receiving requests and one for sending results. In order to gain access to the appropriate channel for the desired sensor, a well known function is queried and passed the

identifier for the sensor, which in turn will return the channel that the request should be sent along. The identifier for a sensor is a well known alias for the sensor, for example to request the internal temperature, the `INTERNAL_TEMP_SENSOR` value would be sent. There is an analogous function to determine the output channel for that sensor which takes the same aliases.

Upon receipt of a request, the sensor component will use the sent alias inside a switch statement. It will then read the memory mapped location which contains the desired result and send this value over the result channel. Again, the `channel_multicast_send()` function is used in case the application has not bound to the output channel for the requested sensor. After sending, the sensor component will again listen for incoming requests.

7.2.1 Button Sensor

Unlike the sensor component, interaction with the button on the t-mote sky is an asynchronous event generated by hardware, and requires its own component. In the sensor component a request is made by another component and a reply sent to it, whereas the button component can only be waited on by applications. The component makes use of the the `BLOCKED_SENSOR` state, and is currently the only one to do so. Normally, a component may only block due to a system call and this blocking is achieved by instructing the scheduler to place it in one of the blocked states described in Section 5.1.1. As this is a system specific component it is able to interact with the scheduler directly and be placed into the `BLOCKED_SENSOR` state. In order to wake this component, whenever the button is pressed the interrupt handler which is associated with a button press, will place the button component at the head of the ready queue. When the button is scheduled again it will send a strobe along its output channel, and return to its blocked state. As before, this output channel is a well known channel which an Insense application must first bind to and listen on. Also as before, when the button component attempts to send along the output channel it uses the `channel_multicast_send()` for the reasons previously discussed.

7.2.2 Pre-InceOS System

The pre-InceOS system uses a similar method to provide access to sensors, however there is an extra level of abstraction within the component itself when accessing the sensor data. From examination of the implementation, this would seem to be an artefact from when a component was able to directly access the sensor data, however it is still present. The sensor component still suffers from the normal overhead which is associated with using a protothread.

7.3 Radio

In order to enable different motes to communicate, a radio is used. The radio on the t-mote sky is the cc2420 [6].

To enable an Insense application to use the radio two different components are used within InceOS: the *protocol* and *xmac* components, both of which can been seen in Figure 7.1.

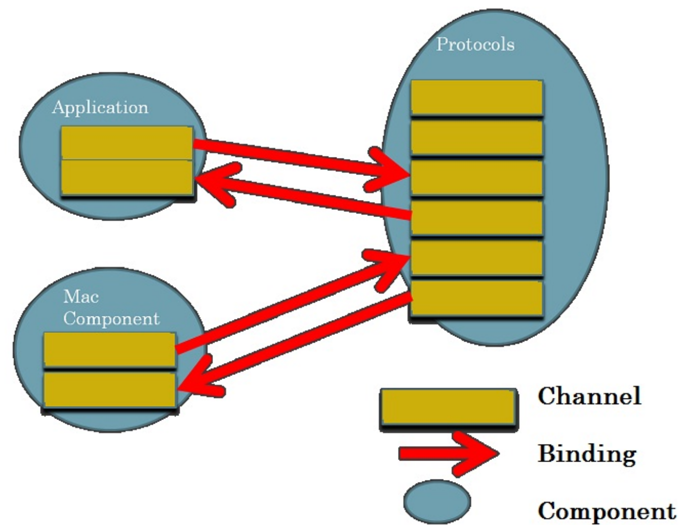


Figure 7.1: InceOS Radio Interactions

7.3.1 Radio Send

The protocol component acts as a protocol stack enabling the Insense application to send a message over the radio using one of the available protocols in the protocol component. In a similar style to the sensors, an application will select the service (protocol) it wishes to use by binding to a channel that represents that service. An application can receive a message by binding a channel to the output channel of a particular protocol. At present only a simple unicast and a broadcast protocol is supported due to time constraints, consequently this design is a proposal for the way in which other implementations of protocols would be included in InceOS.

In order to send data over the radio, the application must first construct a radio request packet, Listing 7.4.

```
typedef struct radio_packet {
    unsigned char addr[2]; //if sending = dest, if receiving = src
    char payload[MAX_PAYLOAD];
    unsigned char payload_size;
}RADIO_PACKET;
```

Listing 7.4: Radio Request Data Structure

The addr field is either the address of the recipient of the packet, when the application is sending data, or the address of the sender, when the application is receiving data from the radio. The address {0,0} is reserved and used as the broadcast address when the application wants to send the data to all reachable nodes in the network, otherwise the address of a specific node may be used. Currently an application can only obtain an address by first explicitly broadcasting a packet and examining the address contained in the RADIO_PACKET when a reply is received. See Chapter 9 for the proposed extensions to this method. The payload field is used to store the data to be sent or which is being

received. Due to hardware constraints, the maximum packet that can be sent is 127 bytes, and this is further constrained by the size of the mac layer header (5 bytes) which must be prepended to each packet. Accordingly, the maximum payload supported is 122 bytes which is the value of `MAX_PAYLOAD`. The payload size is used to indicate the number of bytes being sent, in a similar way to the len parameter in the `channel_send()` function.

Once the radio component has received a request to send a message, it then has the opportunity to first perform any action that is required by the protocol, via which the message is to be sent. Currently no action is taken. The radio component will then send the data to the xmac component.

It is the role of the xmac component to then construct a packet which will be placed into the radio's hardware buffer, ready to be sent over the radio. In order to do this a small header is prepended to the data. This header contains the source address of the current node, the destination address which is inside the request from the application, and the type of the packet being sent, either data, strobe, or ack. The data from the request is then copied into the packet and sent over the radio using the xmac protocol. The software from this point to the hardware transmission, including the radio driver, has been taken from the Contiki implementation. This was done to avoid "*reinventing the wheel*". A detailed explanation of the protocol and radio transmission can be found in [19]. The xmac component will then return to waiting for another send request or incoming packet.

7.3.2 Radio Receive

When a data packet is received by the radio, a hardware interrupt is generated which will place the xmac component at the head of the run queue. It is then the job of the xmac component to read the packet received from the radio's hardware buffer. It will then construct a `RADIO_PACKET`, populating the addr field with the source address of the received packet, the payload with the contents of the packet, and the payload_size with the number of data bytes received. This structure is then sent to the radio component.

Once the radio component has received data from the xmac component, it must then decide what to do with it depending on the protocol being used. Currently nothing is done, but it is envisaged that a protocol would require another header to be prepended when sending the data. This would enable any incoming data to query for the existence of the header and act accordingly. Once the packet has been processed by the protocol, it will then be sent along that protocols output channel using the `channel_multicast_send()`. At this point a listening Insense application will receive the data, otherwise it will be lost.

7.3.3 Issues

The main problem with the radio implementation in InceOS is that it is not complete, even though what is present works correctly. This will be expanded upon in Chapter 9.

The radio is usually the most difficult asset to manage in terms of power consumption. This is due to the attempt to try and find a balance between having it enabled as much as possible to receive packets, against having it turned off to conserve power. This is usually achieved by having a duty cycle, meaning that a separate system entity will be tasked with turning the radio on and off for a pre-specified amount of time. Within InceOS this task has been given to a system component which

interacts with the rtimer component, and is known as the sentinel component. This component works correctly, however the durations for which the radio is on and off are not correct, meaning that packets are missed. Due to timing constraints the time values to keep the radio on and off were not able to be honed, consequently the radio is currently left on permanently.

In terms of power, the radio is also able to control the range of transmission by setting the amount of power that the radio may consume when sending a packet. Currently this value is always set to maximum, and an Insense application may not modify this value. However, this can easily be accommodated by including another field in the `RADIO_PACKET` indicating the level of power to be used. The xmac component would simply set the value before creating and sending the packet. This feature was not included due to time constraints.

7.3.4 Pre-InceOS System

Within the pre-InceOS system a similar concept is used to the above, with a single radio component through which different protocols are accessed via different channels. This component then uses the Contiki radio stack to send data using the different available protocols. Currently, the pre-InceOS system does have the advantage of a fully calibrated duty cycle element.

7.4 Debug

In order to enable an Insense application to communicate debugging information, two system components are available. The *print* component and the *led* components.

7.4.1 Print Component

The `printf` function is an essential way for a C language program to convey data to a terminal output. On the t-mote a function of the same name exists which is able to pass information to a terminal display on a computer, when connected via the USB. In order to offer this functionality to Insense applications, the printf command is placed inside the behaviour of the print component. The component acts in the opposite manner to the button component. It waits for incoming requests from application components on its request channel, as before application components will first bind to this well known channel. The value passed in this request is then used as an argument to the printf function. Due to the blocking rendezvous model, locks are not required to ensure serial access when printing. The value passed should be a string which contains the value to be printed. This does mean that a maximum string size is specified due to the finite size of the data buffer. This component does offer the advantage of platform independence by only requiring the printf function to be replaced by the relevant feature of the platform. If no equivalent is available, then the component can be removed. As with all the static elements, it would be possible to have the Insense compiler determine the maximum string size required and pass it as a value to the gcc compiler for the msp430, to be used in the OS and application.

7.4.2 Led Component

Motes will often come equipped with a number of leds (light emitting diodes). In a similar way to the above printf component, the led component simply acts as a wrapper around the access of the leds to ensure serialised access. Instead of sending formatted strings to this component, well defined values are used to indicate the configuration of the leds. These values should either be sent to one of the two request channels: one for turning on leds, and the other for turning them off. For example, to turn on all of the leds the `LEDS_ALL` value would be sent along the on request channel, and to turn of the green led the `LEDS_GREEN` value would be sent along the off request channel. Again, to aid platform independence, the well known values and functions used to enable or disable a led are abstract from the workings of the component, and can be easily replaced.

Each of these components have similar equivalents within the pre-InceOS system. The difference being that InceOS does not require an indirect access via Contiki to invoke the desired action.

7.5 Dynamic Memory

Unlike the other sections in this chapter, memory allocation is not represented by a component. Instead the well known system calls `malloc()` and `free()` are presented.

Within the GCC development kit for the msp430 these system calls are provided, however they were not compatible with the threaded implementation of InceOS, and were rewritten. The problem was that when determining if a request from the heap was legal, the difference between the current stack pointer (plus a small offset) and end of the heap was used. This was not acceptable because of the following situation. Two components, A and B, have each been created with their own stacks and assume that this has consumed the heap. A's stack will be at a lower address than B's. Now, if A were to perform a memory request for n bytes, the GCC implementation would query the current stack pointer and the end of the heap, determine that there is enough space and return a pointer to that space. A now has the space that it requested, however this space is the same space as B's stack, resulting in B's stack no longer being correct as A writes to it's allocated memory.

To solve this problem was a simple case of allocating the maximum amount of memory which would be available to the system throughout its lifetime. This would service every allocation request by cleaving memory from the initial allocation. However, should a request be made for more memory after the initial allocation is exhausted, malloc reverts to the above situation. Consequently it was decided to reimplement malloc and free without the above problem.

Currently a first fit allocation system is used when allocating memory, and the implementations themselves are based on the algorithms found in The C Programming Language [21].

The pre-InceOS system also offers dynamic memory allocation, however it is reference counted. This has the advantage, compared to the above, in that any memory will automatically be reclaimed to the heap whenever the last reference to it is removed. It is the case, however, that for every allocation a function must be specified along with the size of the allocation. This is due to the fact that if the allocated space is used for a C `struct`, then it may contain fields which themselves are references to allocated memory. If the `struct`'s memory were simply deallocated, orphaned storage would be left behind. The extra space used for the function, the need to insert code to

increment and decrement the reference count, as well as the functions within the runtime that are used to support reference counting appear as overhead. Instead, the compiler can statically keep track of what is allocated and its lifetime during compilation, and simply use malloc and free as and when required. Also, as InceOS no longer requires data to be intermediately copied when conducting channel data transfers, static allocations (local variables) are used within behaviour loops to act as the buffers discussed above for channel sends, receives and selects.

7.6 Name Server

One recurrent theme through this chapter has been the use of well known channels which enable Insense applications to bind and communicate with the well-known system components. A much more desirable, and currently unimplemented, option would be the use of a single well known name server which could be queried to determine the available system component and their respective channels, in a similar style to a Java RMI name server. The need to discover the available components is required due to the different hardware platform to which it is intended the InceOS will be ported.

Chapter 8

Evaluation

The following chapter details the comparison of InceOS against the existing runtime combined with Contiki.

8.1 Experimental Set-up

In the following all tests are carried out on the t-mote sky hardware platform. This consists of a 25 MHz processor, with 10 KB of RAM and 48 KB of flash storage. The radio used was the cc2420 [6]. During all tests, the t-mote was connected to a computer (pc) via the USB serial port, providing full power.

In the following graphs, each blue data point represents the average of 100 iterations of the system element being tested. For example, in Figure 8.1 each data point represent the average time of 100 calls to create a component. The error bars are the standard deviation (σ) over the 100 results for that particular data point. Each graph also shows the maximum time in red, and minimum time in green to complete the action being tested. For each comparison between InceOS and the runtime, all elements were kept the same, this includes the number of component in each system.

As noted in the proposal for this work, it was a concern that the Insense compiler would not be targeted to the virtual machine in time for testing and this was the case. As a result, all of the Insense examples used to test InceOS were hand crafted to mimic the expected output from the compiler if it were to have been re-targeted.

In the following results, timing was taken from the millisecond timer, timerB. One of the problems when collecting results arises from the granularity of the timer, specifically it is not fine enough to observe completely accurate values, as the smallest observable time difference is two hundred and forty four microseconds. This is insufficient in some cases, such as the context switch times for InceOS which are calculated at approximately nine to fifteen microseconds. This also leads to noticeable quantisations in the results, particularly for InceOS. For example, in Figure 8.1 (b), a distinctive oscillation between the values of 244 ms and 488ms can be seen in the maximum duration of creating a component.

8.2 Size

In terms of size there are two measurable quantities. The first is the amount of flash space that is occupied and the second is the amount of RAM that is occupied.

8.2.1 Flash Occupancy

Element	Code Size (bytes)	Data (bytes)	bss (bytes)
Contiki (complete)	60506 (123.00%)	315	6517
InceOS (complete)	13684 (27.84%)	72	3032
runtime (complete)	9275 (18.87%)	58	0

Table 8.1: Complete System Flash Occupancy

Table 8.1 shows the differing amount of flash consumed on the t-mote by each system alone, the null application is specified. Firstly, the absolute number of bytes used, in brackets showing the total amount of space which could be consumed as a percentage. Secondly, the size consumed by global and static variables which are not initialised (data). Thirdly, global and static variables which are initialised to zero (bss).

As previously noted, Contiki and the runtime both use conditional compilation in an attempt to reduce the size of the binary which is produced. In this way, only the required system elements are included in the compilation action; for example, if an Insense application does not use the radio, the runtime will not include the runtime elements required to use the radio, and Contiki will not include its elements required to run the radio. Contiki will also do this independently for its own native applications. Consequently, Table 8.1 shows the amount of space that would be consumed by each system in its entirety with every component included, even though it is not possible for Contiki to fit on the mote. By comparison with Contiki and the runtime combined, InceOS is less then a quarter the size in terms of flash occupancy, less than a fifth the size in terms of the data section, and under half the size in terms of the bss section. This is not a fair comparison as InceOS does not support all of the functionality offered by Contiki, consequently the following tables show the flash occupancy when running an Insense application which uses different system elements.

Element	Code Size (bytes)	Data (bytes)	bss (bytes)
Contiki	19236 (39.14%)	224	1985
runtime	7598 (15.46%)	38	160
application	938 (1.9%)	22	20
TOTAL	27772 (56.5%)	284	2165

Table 8.2: Runtime Hello World Sizes

Element	Code Size (bytes)	Data (bytes)	bss (bytes)
InceOS	13648 (27.84%)	72	3032
application	110 (0.15%)	0	0
TOTAL	17012 (27.99%)	72	3032

Table 8.3: InceOS Hello World Sizes

Element	Code Size (bytes)	Data (bytes)	bss (bytes)
Contiki and runtime	34679 (70.55%)	272	2247
application	7674 (15.61%)	22	20
TOTAL	42326 (86.11%)	294	2267

Table 8.4: Runtime Radio Send Sizes

Element	Code Size (bytes)	Data (bytes)	bss (bytes)
InceOS	13648 (27.84%)	72	3032
application	578 (1.11%)	0	0
TOTAL	15656 (28.95%)	72	3032

Table 8.5: InceOS Radio Send Sizes

Comparing Tables 8.2 and 8.3 shows the sizes for a simple hello world program which outputs "Hello World". Comparing the system sizes, InceOS is smaller than the combination of the runtime and Contiki. This is due to InceOS being specific to the needs of the Insense language, rather than the current conglomeration of an Insense specific runtime animated by a general purpose embedded OS. In particular, there is a large reduction in the size required for the application itself.

Tables 8.4 and 8.5 show the sizes for an application using the radio. This is where the most noticeable savings made by InceOS can be seen. Firstly, due to conditional compilation, the runtime and Contiki combination is larger when compared to the simpler Hello World program. InceOS stays the same size. Secondly, due to the presence of stacks in InceOS, it does not require as much static allocation or dynamic variables, instead using local variables. Accordingly, the size of the radio application for InseOS is considerably smaller than the pre-InceOS system.

By reducing both the size of the animation system and that of the C program produced by the Insense compiler, less flash space is used and is available for either growth of the OS to include more services, or for the authoring of more Insense applications with greater complexity. Currently InceOS is only implemented on the t-mote sky platform with 48KB of flash, however the small size of InceOS and the applications that it will animate will remain almost constant as they are run on different platforms with greater flash capacities. The only possible growth in terms of size would come from the platform specific elements, however these are only a very small part of the implementation.

8.2.2 RAM Occupancy

Due to the threaded nature of InceOS, each component requires its own stack and, as previously mentioned, this stack must be big enough to support both the component's behaviour and the functions calls made into the OS. Accordingly, InceOS has a higher RAM usage than the pre-InceOS system.

Element	Available RAM (bytes)
InceOS	3491
Runtime Without Radio	3416
Runtime With Radio	3284

Table 8.6: Post Initialisation Available Memory

Table 8.6 shows the amount of memory available immediately after each system has been initialised. Contrary to initial assumptions, InceOS only shows a slightly larger amount of available RAM, however there are two things to consider.

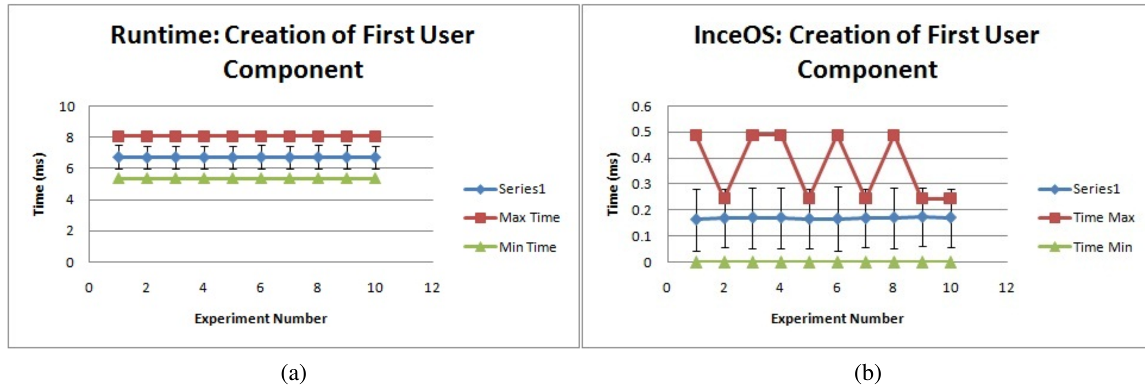
The first is that InceOS uses a large amount of statically allocated space, as can be seen in the larger bss sizes for InceOS in the tables in Section 8.2.1. As an example, the `CHAN` structure occupies 9 bytes of space, and there are currently 160 of them allocated. This alone consumes 1.4 KB of space. The second is the size of stack required per component. Currently, 120 bytes are added to each requested component to allow for system and interrupt stack requirements. This is in addition to the 19 bytes for the CCB structure, required for each component. As the above results are for post system initialisation, the system and well known components have already been allocated. The amount of memory available before the system creates any system components is 6787 KB of RAM.

8.3 Timing

This section presents a comparison of the duration of different actions in each system, primarily focusing on the system calls.

8.3.1 System Call Timings

Component Create



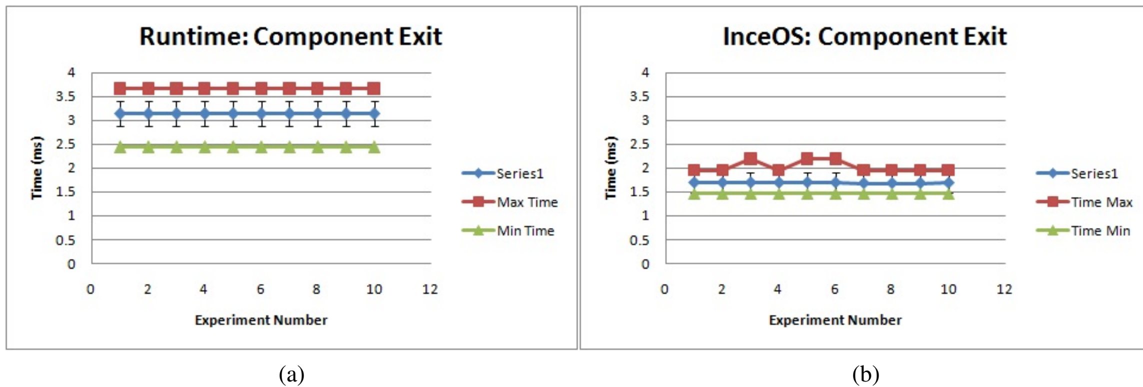
(a) (b)

Figure 8.1: Component Creation

Each graph in Figure 8.1 shows the duration for the creation of an Insense application's first component, immediately after the system has been initialised. By a comparison of the average times, InceOS is 67 times faster than the current runtime, reducing the average from 6.7ms to 0.16ms. It can also be seen that InceOS presents a smaller standard deviation than the runtime, leading to more consistent results.

A justification of the improvement is because the process of creating a component in the runtime over Contiki is more involved than for InceOS, Section 4.1.2. The Insense compiler must statically create a pointer which will be used to hold a reference to the component. The creating component will call a function which is used to allocated a Contiki protothread, passing it a pointer to the creating component. This function is also used to allocate memory for the *this* structure for the component. During this initialisation, the component structure will use the unique identifier for the Contiki process as its own, it will also take copies of all the data structures required to interact with the protothread. The statically allocated global variables associated with the component are then initialised with a component specific function, essentially the constructor function. It is at this point that any required channels are created. These global variables are required due to the fact that a protothread does not maintain its local state after a blocking or yielding call. The final task is to actually create the protothread and start it. This involves several Contiki system calls, and eventually posting an event back to the calling component to indicate that the new protothread is complete and that it may continue its execution. It is worth noting at this point that the runtime still constructs a "vtable" in order to keep track of all the state and functions associated with a component, even though this is a depreciated aspect of the runtime.

Component Exit



(a) (b)

Figure 8.2: Component Exit

A comparison of the time required to terminate a component is shown in Figure 8.2. Although not as drastic an increase as can be seen in the creation of a component, there is still a 46 % decrease in the time required.

Within the pre-InceOS system, disposing of a component is not as simple as in InceOS. As a component is a dynamically allocated data structure and a protothread, a component must be destroyed in two stages. Firstly, the memory allocated for the component itself and any channels it used must be returned, however Contiki must clear up the protothread that was used to animate the component. In order for Contiki to destroy a protothread a complex series of actions must take place involving many functions. Upon returning from a protothread which is to be destroyed, a function called `exit_process()` is invoked. This function must first broadcast to all protothreads in the system that this protothread is about to die. This is done to enable any services which have allocated space for the process to deallocate it. This stage alone requires that every other protothread in the system be executed before destroying the protothread. As previously discussed, each protothread will execute until it explicitly yields or blocks. The protothread to die will then again be invoked with the `PROCESS_EVENT_EXITED` event in case it requires to deallocate any state. This has already been taken care of by the runtime. Finally, the protothread is removed from the run queue.

Unlike InceOS, where the CCB data structure is allocated and deallocated, the analogous process structure within Contiki is implicitly declared as part of the function that is executed by the protothread, via a C pre-processor macro. Consequently it does not need to be deallocated when a protothread dies. However, in the case that a protothread is dead and never returns, the process structure simply occupies space in the flash.

Component Stop

System	Time (μs)
InceOS	0.32
Runtime + Contiki	0.56

Table 8.7: Component Stop Times

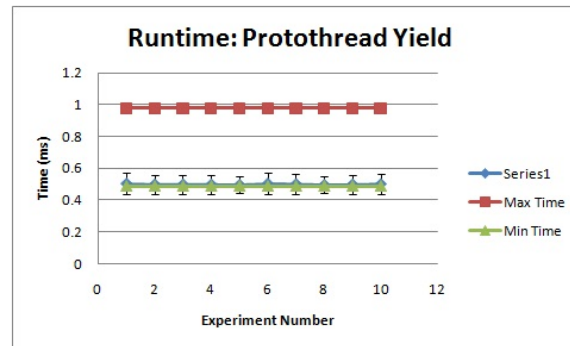
Table 8.7 shows the duration of the `component_stop()` in InceOS and its equivalent in the runtime. This shows InceOS to be faster. These results are not displayed in graph form as they were calculated from the number of assembly instructions required to execute the action. This was necessary as measurement with the timer constantly yielded zero. To illustrate, InceOS requires 8 CPU cycles and each cycle takes 40 ns, thus the action takes 320ns.

The runtime uses a similar method to InceOS to enable one component to terminate another, using the stopped member of the *this* structure. However, the runtime requires some more state to be stored. This is why InceOS is only slightly faster, and why both results are on a small scale. One major difference between the runtime and InceOS is in what happens after the stop call has finished. In InceOS, the invoking component continues execution, however the runtime will block, waiting for an event to indicate that the component which has been instructed to terminate has done so. The timing of this was not measured as it is dependent firstly on the number of other components, including those made eligible by interrupt, that are executed before the targeted component, and secondly on the point at which targeted component previously ceased execution. Between resuming execution and reaching the loop test of its behaviour, the component may block a number of times.

Component Yield and Pre-Emption

Context Switch Method	Best Case	Worst Case
Yield or Block	9.28 μs	9.8 μs
Pre-Emption	14.68 μs	15.2 μs

Table 8.8: Context Switch Times



(a)

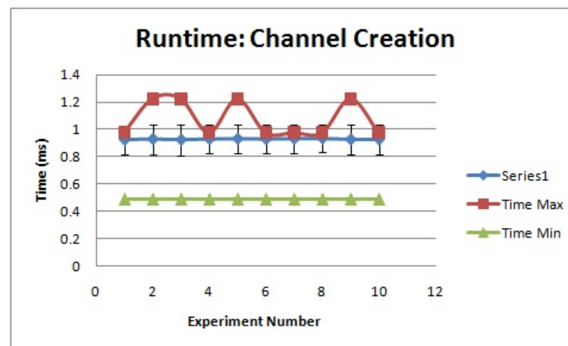
Figure 8.3: Runtime Yield

Similar to stopping a component, yielding a component in InceOS was too fast to be detected by the timer being used, and accordingly instructions were counted. This was not required for the runtime. Table 8.8 shows the times required for InceOS to swap between components either by an explicit yield request or by interrupt driven pre-emption. The worst case column refers to the situation where the InceOS run queue is emptied and the head and tail values must be correctly updated, the best is when this does not happen. Figure 8.3 shows the results for an explicit yield request in the runtime, Contiki does not support pre-emption as previously discussed. By comparing the results for an explicit yield request it can be seen that InceOS is 51 times faster than the runtime in the worst case, and 32 times faster in the worst case when comparing a runtime yield against pre-emption in InceOS. The effect of adding pre-emption can be observed in Section 8.4.

Channel Create

Contributions to Time	Time
Basic Time	4.6 μs
Number of Component Channels	0.52 μs
Total Number of Channels	0.52 μs
Number of Channel Links	0.48 μs

Table 8.9: InceOS Channel Creation



(a)

Figure 8.4: Runtime Channel Creation

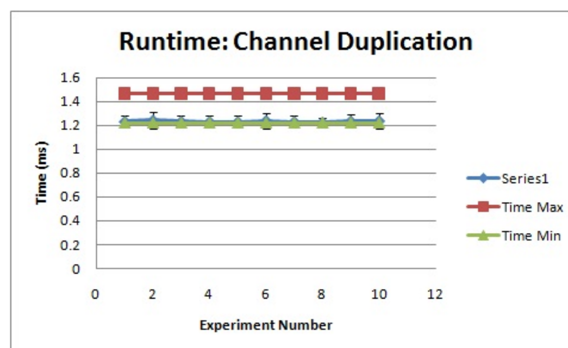
Figure 8.4 shows the time taken to create a channel within the runtime. Table 8.4 shows the basic time for creating a channel plus a number of situation dependent additions to the basic time. Number of component channels refers to the number of channels that the component to which the new channel will be assigned already has associated with it, Section 5.1. Total number of channels refers to the number of channels which must first be searched when finding a channel to use from those which have been statically allocated. The number of links refers to the number of connections that a channel has and must be initialised. For example, if a component already has 6 channels, then creating another will add 6×0.52 microseconds to the basic time. If one hundred channels have already been allocated, then this will add 100×0.52 microseconds to the basic time. If a channel is allowed to form eight different bindings, then this will add 8×0.48 microseconds to the basic time. All of these scenarios have a cumulative effect on the time. Currently, the worst case scenario in InceOS has these values at 19, 157, and 8 respectively. This gives a worst case creation time of 99.96 μs, 10 times less than the runtime average.

When creating a channel within the runtime, it is allocated directly from the heap. There is no limit imposed on the number of channels which may be created or associated with a component, only the available memory limits this. It is the job of the Insense compiler to generate code to associate channels and component. The advantage in InseOS comes from the pre-allocation of space.

Channel Destruction

The time taken to destroy a channel is exactly 2.48μs. In addition to this, the channel to be destroyed must first be unbound to prevent dangling connections, the time taken for this can be seen in Figure 8.6. Within the runtime, there is no explicit mechanism to destroy a channel as their numbers are not constrained. As a result, channels are destroyed whenever their associated component is destroyed. Time did not permit testing of this aspect, however it would be possible to measure this by measuring the destruction time of a component with 0 channels, one with n channels, and using the difference to indicate the duration.

Channel Duplicate



(a)

Figure 8.5: Runtime Channel Duplication

Figure 8.5 shows the time taken to duplicate a channel in the runtime. It should be noted that unlike InceOS, duplication of a channel is not a distinct function, but rather it occurs as and when required as an integrated part of the runtime itself.

Within InceOS, duplication of a channel takes 8.76μs. As previously mentioned, duplication of a channel relies on the create and bind system calls. Consequently, these times, Table 8.9 and Figure 8.6, are in addition to this basic time.

Channel Adopt

Within the runtime, the notion of adopting a channel does not exist. Within InceOS 13.28μs are required to adopt a channel in the worst case.

Channel Bind and Unbind

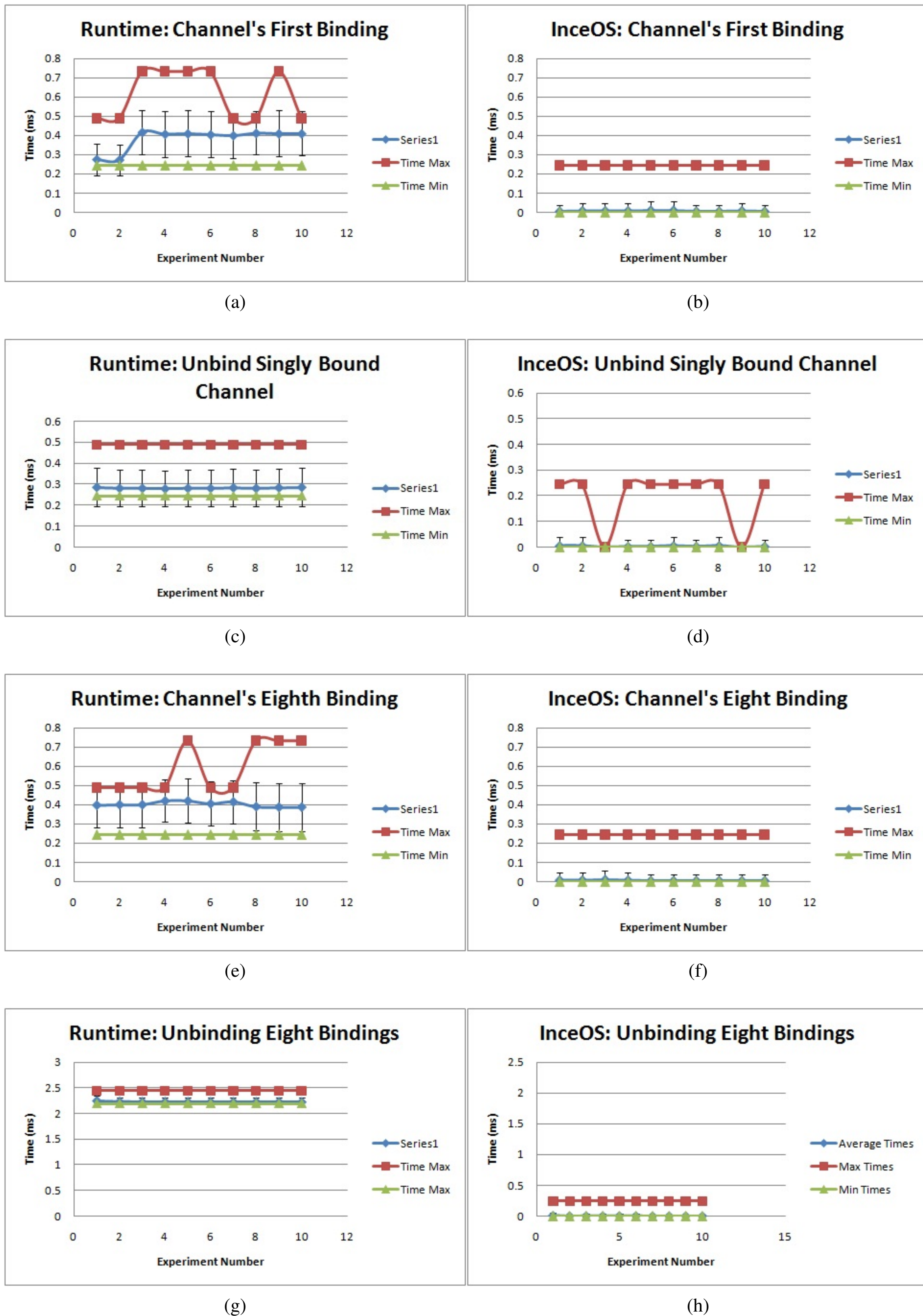
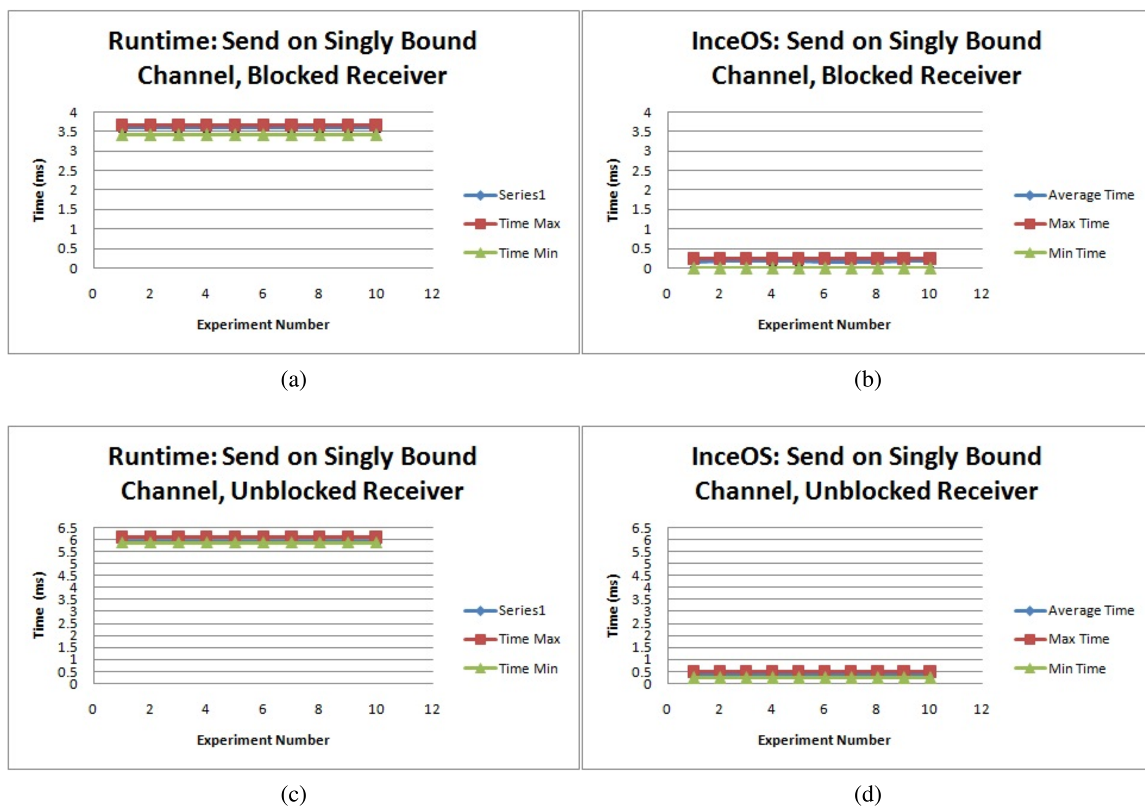


Figure 8.6: Channel Binding Times

Figure 8.6 shows the different durations required to bind together channels in the different systems. Comparing (a) - (d) it can be seen that InceOS is faster but not by such a large margin. However, comparing (e) - (h) it can be seen that InceOS scales better than the runtime, with no significant change.

As previously discussed, the pre-InceOS system has no limits on the number of channels. Therefore, when binding together channels, a pointer to each channel to be bound is placed in a linked list associated with the other. This accounts for the fairly consistent results in (a) and (c). However, when unbinding eight channels this linked list must be traversed, with each pointer followed to the other bound channel to remove the references. Unlike the direct access to data structures in InceOS, the runtime uses function calls to access the lists, and also requires functions calls to handle the reference counted memory.

Channel Send



(a) (b)

(c) (d)

Figure 8.7: Send Times of a Singly Bound Channel

Figure 8.7 shows the times required for both the runtime and InceOS to send data along a channel when the receiver is both ready (blocked) and not ready (unblocked). From a comparison of (a) and (b) it can be seen that InceOS is 22 times faster than the runtime, and from (c) and (d) and increase of 18 times.

Compared to InceOS (Section 6.3.9), the act of sending data along channel is different. Whenever a send request is placed, the runtime will generate a new Contiki protothread which is specific to

the request in terms of the component (protothread) making the request, the channel and the data - i.e. these fields are copied into a new structure supplied to the protothread. When this protothread is run it will check for any connections, if none are present it will then wait on an event signifying a binding, which is generated when two channels are bound. If there are connections, then the linked list in which these channels are kept is traversed to find any eligible matches. Matches are determined by a state flag associated with each channel. If there are eligible channels found then the data is copied directly from the sending component's buffer into the receiving channel's buffer, the receiving channel's buffer flag is reset, and an event sent to the sending component's protothread, signifying the completion of the data transfer. At some later point when the receiver is executed, the data will be copied from the channel's buffer into the component's buffer. If there are no eligible channels, then the protothread will block waiting for an event to indicate the data was taken and update its status flag. In this case, the sender will copy its data into the buffer associated with the channel being sent over. When executed, the receiver will copy the data being sent into its component buffer and post the receive event to the sender. Figure 8.8 indicates where the buffering is taking place. In the case of a send call which blocks, buffers 1, 2, and 4 are used. In the case of a send call which does not block buffers 1, 3, and 4 are used.

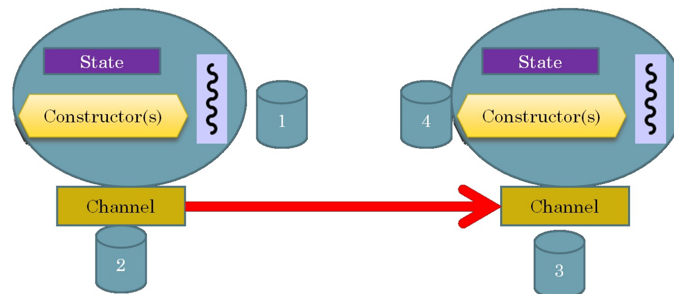
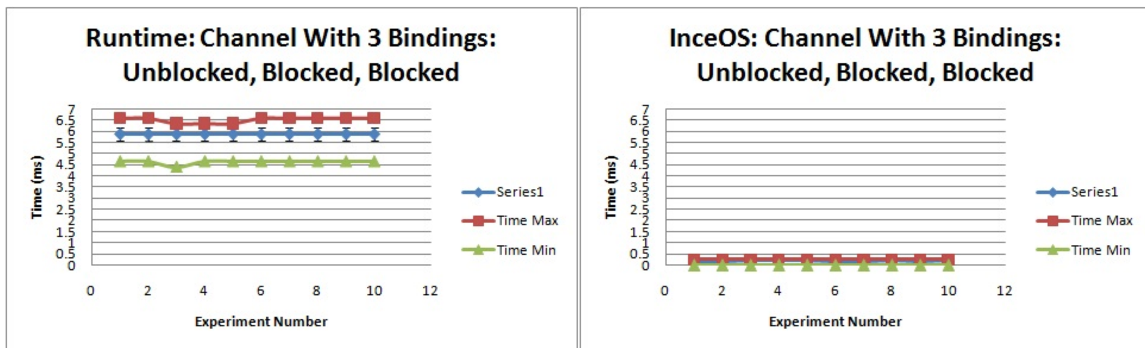


Figure 8.8: Buffering in the Runtime

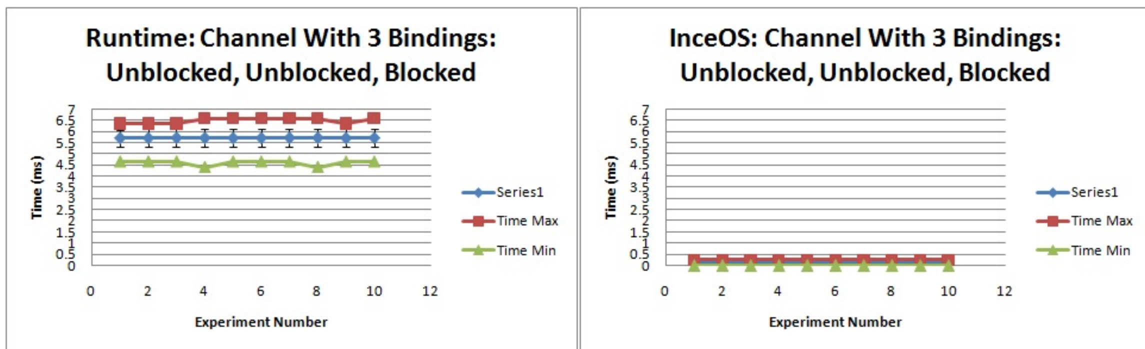
It is worth noting that Insense requires a non-deterministic section when sending or receiving across channels which have a 1:N or M:1 configuration. This is explicitly done in InceOS by using a random number generator, however within the runtime this non-determinism is simply the result of the state in which the connected channels are found in, dictated by the status flags of channels. Consequently, the runtime does not guarantee a fair selection when choosing a connection.

The faster times of InceOS can also be attributed to the close relationship between channel states and the scheduler, compared to the pre-InceOS system.

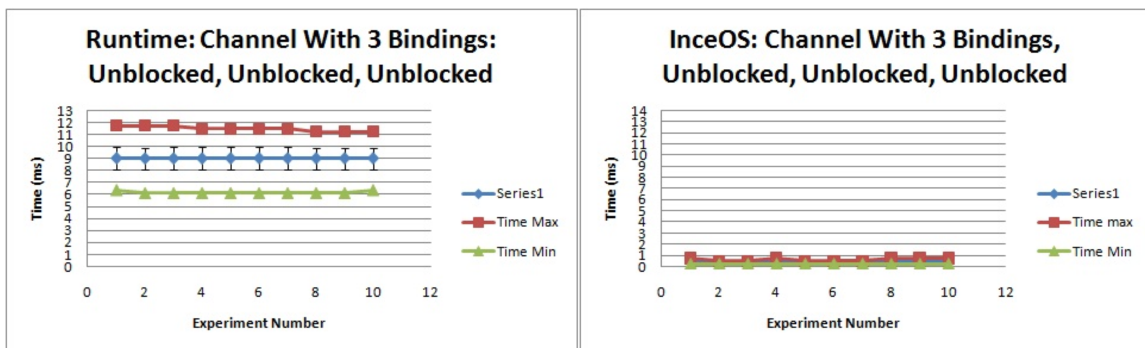
(a) (b)



(c) (d)



(e) (f)



(g) (h)

Figure 8.9: Send Times Over a Channel With 3 Bindings

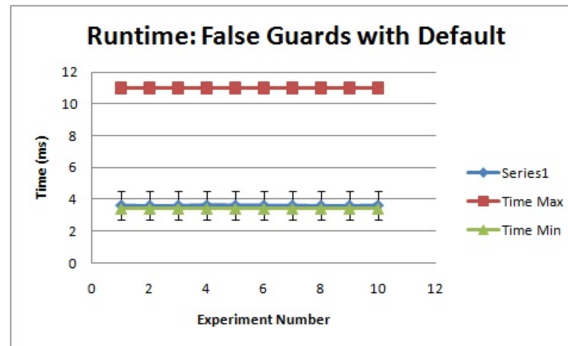
Figure 8.9 shows a comparison between InceOS and the runtime when executing a send over a channel which is bound to three others. The difference between the results, as indicated in the graph titles, is whether or not the receiving channel's component is ready (blocked) or not (unblocked). Each channel is in a separate component, and each of the above rows shows the same operation in each system.

Again, InceOS dramatically outperforms the runtime in each case. An interesting observation is the linear increase in the improvement from InceOS seen in the first three rows of 24, 32, and 38 times. The last row is only an improvement of 19 times. Time did not permit, however an extension to this experiment would be to increase the number of channels involved to determine if this increase is constant, grows, or shrinks.

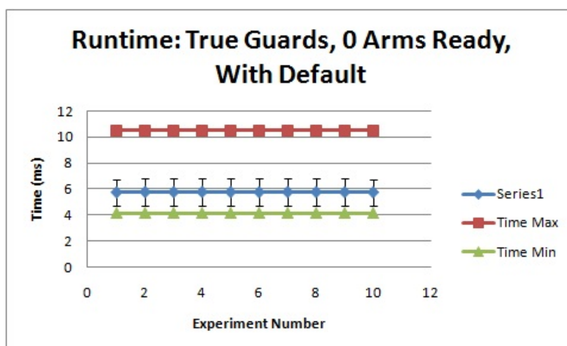
As previously discussed, the runtime does not provide fair determinism when choosing a channel from multiple eligible channels, it selects the first available. In contrast to this, InceOS examines all channels before making a decision. Consequently, if the channels in (c) were arranged with the blocked channel being the first in the connection list, then results found in (a) would be seen. However, as can been seen in (c) this performance will always be degraded when the non-eligible channels come first. The justification of the improvement of InceOS is the same as for the single send.

Tests were not performed for the receive operation as it is symmetric to the send operation in both systems.

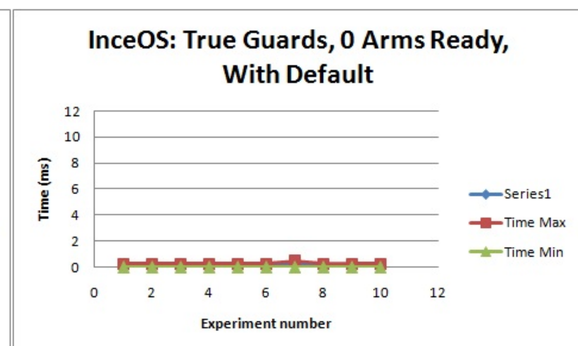
Channel Select



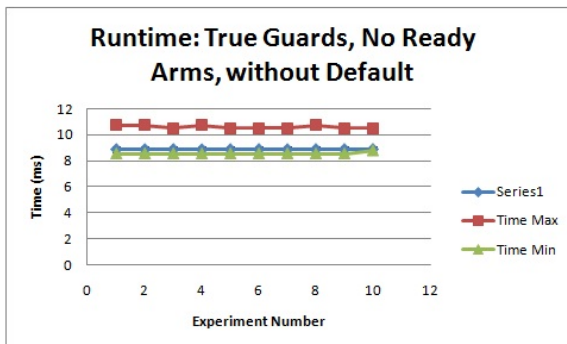
(a)



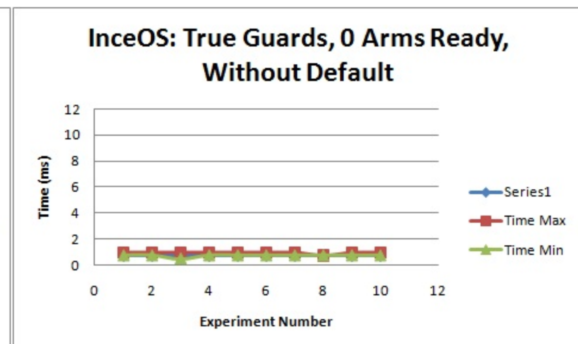
(b)



(c)



(d)



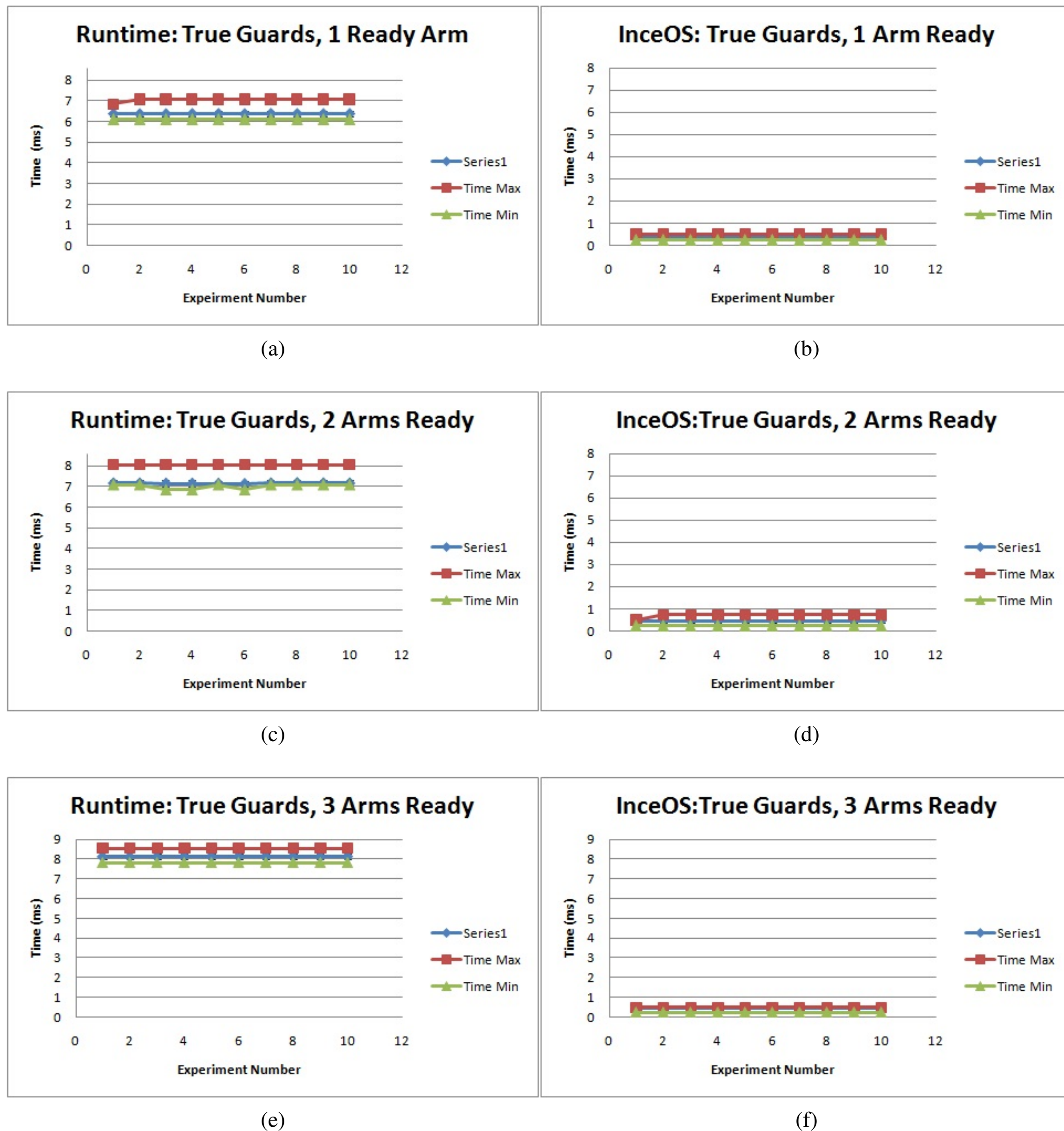
(e)

Figure 8.10: Select Times Without Ready Arms

Figure 8.10 shows the duration of the select call in each system when no arms of the select statements (channels) are either eligible to receive data due to their associated guard values, (a), or the channels are not ready to send their data, even though they are eligible, (b) - (e). There are a total of three channels being selected over. It was only required to test the guard values as either being all true or all false as once the number of eligible channels were determined from the guard values, both systems consider there to be only as many channels as were eligible.

For InceOS, the case of no guard value being true, with the default specified was faster than the system tick and was therefore calculated at 5.52μs. This is 688 times faster than the values of (a). A comparison between (b) and (c) shows an improvement of 29 times and an improvement

of 12 times between (d) and (e). These results are consistent when compared with Figure 8.7, accounting for the increased number of component operating in the system. The justification for the improvement again comes from the fact that InceOS is customised exactly to the requirements of Insense, compared to the Insense specific runtime which is subservient to Contiki.



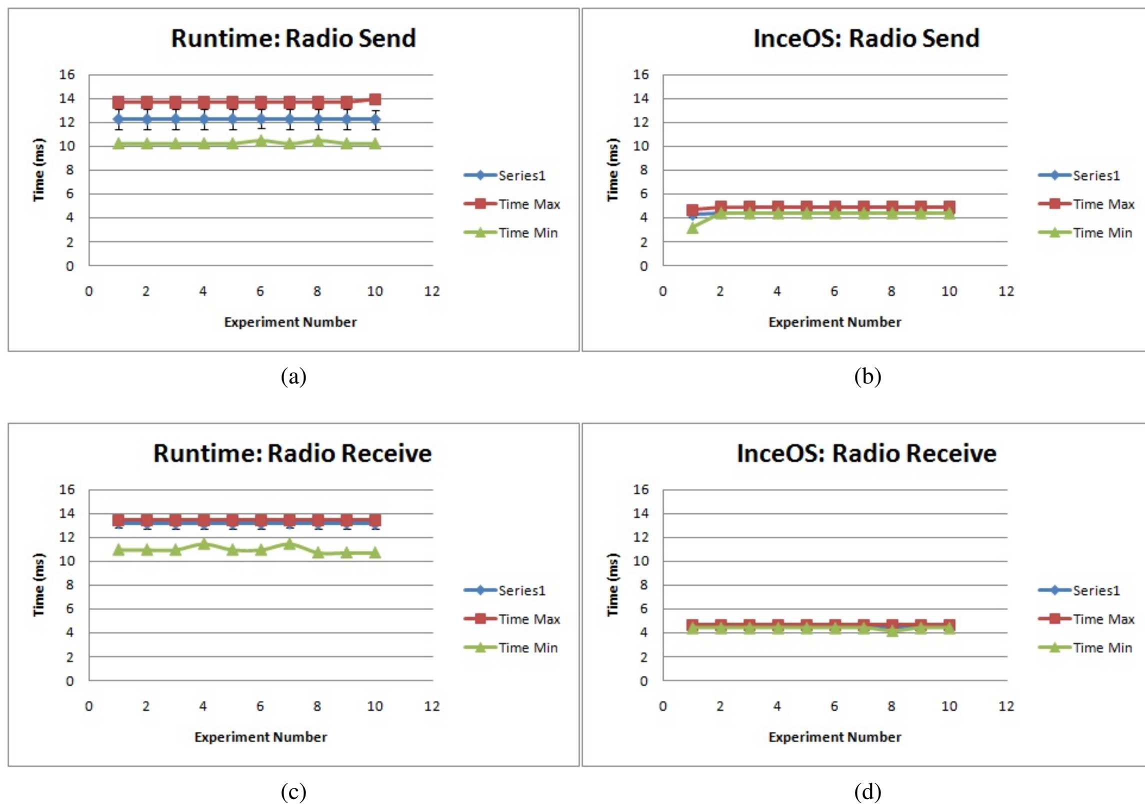
(a) (b) (c) (d) (e) (f)

Figure 8.11: Select Times With Ready Arms

Figure 8.11 shows the duration of the select statement when arms become eligible. In each row a clear improvement is shown by InceOS when compared to the runtime. When comparing (a) and (c) against (b) and (d) there is a consistent improvement of approximately 15 times, however the comparison between (e) and (f) shows an improvement of 17 times. Time did not permit, however it would have been interesting to expand the number of channels involved during the tests to discover if the improvement is linear in relation to the number of channels or if the runtime's

duration increases at a greater rate than that of InceOS.

8.3.2 Radio



(a) (b)

(c) (d)

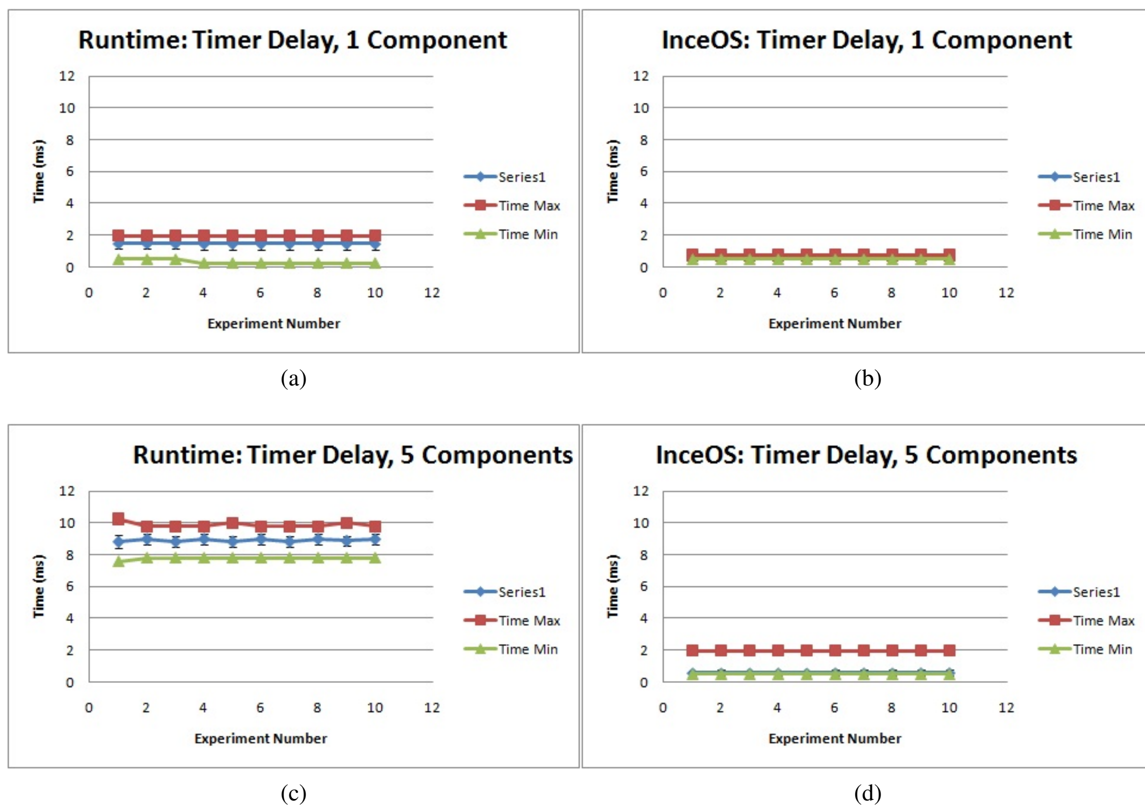
Figure 8.12: Radio Send and Receive Times

Figure 8.12 shows the time taken for a radio packet to be sent from an application component to the lower level mac layer component, (a) and (b), and for an incoming packet to be sent from the mac layer component to the receiving application component, (c) and (d). As stated in Section 7.3, the mac protocol within InceOS is not fully optimised to the same standard as the one found in Contiki, therefore a comparison of radio transmission times would not be fair. Altering the Contiki layer to contain the same deficiencies would not present an accurate representation of the pre-InceOS system. The only user component in the system is either a sending component for the send tests, or a receiving component for the receive tests.

The first point to note is the consistency in the results for InceOS, the runtime displays a 1ms increase in the average time when receiving a packet from the radio, the precise cause of this is not clear. The second point is the almost three times decrease in the duration of each action within InceOS. This is a combination of the improved channel communication facilities, as well as the more powerful scheduler. It is also the case that InceOS does not use the Rime stack [10] found in Contiki, which requires that a protocol transmit data through each lower protocol level before being physically transmitted, in the reverse situation a packet must traverse up the stack to the appropriate layer.

8.3.3 Timer Delay

Figure 8.13 shows the delay between when a timer interrupt should be delivered and when it actually is delivered using the timerA clock. This is done as the runtime does not support Insense programs using the (millisecond) timerB clock. (a) and (b) show the results for a single component requesting a timeout of one second. (c) and (d) shows the results for five components all requesting one second timeouts at the same time. For the latter results, the measurements were taken from the last component to run so as to collect the worst case results. In all cases, the delay time includes the transit time of the data from the timer to a component.



(a) (b)

(c) (d)

Figure 8.13: Rtimer Component Delay

By comparing (a) and (b) in Figure 8.13 it can be seen that InceOS is faster in reference to the averages observed, however the minimum times seen in (a) show that the runtime can be quicker. This is due to the implementation of the runtime going outside of the constraints of the language and not relying on the standard channel communication process to communicate with the component who requested the time out. Instead, the runtime directly interacts with the requesting component's channel buffer and generates the appropriate event, bypassing the creation of Contiki process and data duplication. This also accounts for the fact that the delay is smaller than the standard time required to send a message over a channel.

When comparing (c) and (d) it can be seen that this "cheating" from the runtime does not enable the improvement to scale as the number of components increases. InceOS has some initial difficulties trying to service five requests for the same moment in time, leading to a unique spike in each set of

results (d) of 2 ms . However, as the delay measured is the cumulative effect of the timer delay plus the time taken for the preceding components to execute and block, the gap between when exactly each time request is due grows, enabling InceOS to service the request in a more timely manner.

8.4 Throughput

In order to test the throughput of the system, an example Insense application was created. One representation was generated by the Insense compiler from an Insense language program, the other was hand crafted. So as not to simply test toy examples, a real world application was created which collected data from the sensor components, averaged them over one thousand iterations and then communicated those averages to another component; the radio was not included in this test as it is not complete. Considering this task is representative of the essential function of a sensor mote, it was the only one used. There were five components each performing this task in parallel for the test.

System	Duration (s)	Time Quantum (s)
Runtime	203.3982 ± 0.091	N/A
InceOS	149.7886 ± 0.615	1
InceOS	151.036 ± 0.109	0.03125

Table 8.10: Throughput Tests

Table 8.10 shows the results. It can be seen that InceOS was 54 seconds (25%) faster compared to the runtime with a time quantum of one second. When using a smaller quantum of 31ms, the performance is worse. This is due to the fact that the example task naturally blocks in order to communicate with other components, consequently a component does not execute for long enough for the pre-emption to have an effect. Therefore, the smaller quantum only has the effect of needing to check the pre-emption lock more frequently, taking time away from the component's execution. This could be indicative of pre-emption not being as useful an addition to a sensor mote as first predicted, considering that Insense uses the blocking rendezvous model, and is a good indication that further experimentation is required in this area.

Within the runtime is a component called the *tick* scheduler, and it is this that acts as the scheduler for Insense programs. This implementation is logically equivalent to that of the timer component discussed in Section 7.1.1. This scheduler works by using the channel mechanism to send "ticks" to a component, at a time initially specified by the component. A component will first block waiting for a tick, and then upon receipt of a tick, the component is able to proceed. However, as components are implemented by protothreads, and it is the responsibility of Contiki to schedule protothreads, the scheduling of a component is ultimately dictated by the scheduling of the underlying protothread by Contiki.

Blocking and unblocking components makes use of the event driven nature of Contiki. In order to enter a component into a blocked state, the runtime will invoke a `PROCESS_WAIT_EVENT_UNTIL` request on one of a number of given events that are either defined by Contiki or by the runtime. This will cause the protothread to remain in the list representing the protothreads, but not be eligible to be scheduled until the event upon which the protothread blocked is generated and delivered.

Unblocking a component (protothread) involves generating the required event and either *posting* it directly to the process, or broadcasting the event to all processes. The run queue within Contiki is a static list of protothreads, in the sense that protothreads which have blocked waiting for events will remain in the list. The scheduling algorithm consists of the following: once the current protothread returns, go to the next element in the list which is ready to run and invoke it. This process requires querying every protothread until one is found.

8.5 Power Consumption

It was intended that a comparison of the power consumed by each system during its operation would be conducted, however this was not carried out. To calculate the power which was consumed, the current I and voltage V present in the system would be required, the power P could then be calculate by $P = I \cdot V$. The voltage can be measured by using the internal voltage sensor of the t-mote, however in order to obtain the current values an oscilloscope would be required, connected across the battery terminals. At the time of conducting the experiments no oscilloscope was available, and due to timing constraints, one was not procured.

At first glance it may appear possible to monitor the voltage levels across the OS and runtime operations, however due to the fact that batteries will sacrifice longevity in order to maintain their maximum potential, this was not feasible or insightful [22].

Consequently, it can only be assumed that due to the reduced times and fewer instructions needed for system operations in InceOS compared to the runtime, less power is being used; however this is not currently proven.

Chapter 9

Future Work

Future directions for the work documented in this project can be split into two categories: improvements to the existing work and future directions.

9.1 Improvements and Optimisations

Currently InceOS uses statically allocated data structures to save on memory fragmentation during runtime; however these values are currently educated guesses. A much more desirable option is to have the Insense compiler determine the exact requirements of the system and pass these values as compiler arguments. InceOS currently uses compiler macros for these values, and this addition would be a simple change.

Insense has an error reporting system that uses the return values of certain system calls to indicate the presence of errors. Currently, InceOS does not completely support this.

InceOS requires that an extra amount of stack space be allocated in addition to the space required by a component's behaviour, in order to meet the space needs of the system calls and interrupts. Instead of allocating this space on a per stack basis a better option is to have a system stack which could be switched to upon entry of a system call. This would reduce the RAM consumed, enabling more components to exist. Consequently, the registers saved during a context switch would need to be saved in the CCB of the component. At present, this is the main limiting issue for InceOS.

Discovery of system components currently involves the use of well known functions to gain access to these components' channels. This strategy relies too much on the compiler's knowledge of what components exist in the targeted system, and is not very generic. A better solution would be to have a single well known *name server* component, which must be provided by InceOS on every platform. This would enable a single process to detect system components, by querying the name server on any platform.

Both Contiki and the runtime use conditional compilation to only include the system modules which an application is using. InceOS does not use this yet, however it is almost complete in the sense that all of the InceOS modules are individually compiled into object files, which are placed into an archive file. The normal procedure is to have the application only form links during the linking

phase with the required objects files. All that is required to successfully integrate conditional compilation into InceOS, is to decouple a number of system modules to ensure that they do not form dependencies and link with modules which are not required by the application. It is also possible to have the Insense compiler pass compiler time arguments to the gcc compiler which would not compile certain modules at all.

9.2 Future Directions

The first area for future research is in relation to the scheduler within InceOS. Currently, only a simple round robin scheduler is used, offering some priority boost to components which are unblocked by interrupts. This leaves substantial room to take better advantage of the states that components can be in. For example, if a component which was in a blocked state is unblocked by another component, it would make sense to give this component a priority boost to account for the lost time it spent being blocked. This could be achieved via a number of different priority queues within the scheduler.

More implementation than research is the goal of porting InceOS to different platforms. Apart from making InceOS and ultimately Insense more accessible, it would also enable Insense to run on platforms with more flash memory and RAM. This would enable more complex Insense applications to be run.

As previously mentioned the protocol component is not complete, missing key elements such as a network discovery protocol which is required for a sensor networked device. However, rather than simply adding this functionality to the OS, a more desirable option is to abstract over radio communications completely, enabling components on different devices to communicate in the same way that components on the same device do. This could be achieved via the use of proxy components which would be automatically generated by the Insense compiler, however this raises an issue because of the blocking semantics of the language. Once an application component has sent its data to the proxy then it should technically resume execution, however as it has not sent its data to the recipient component on the other mote, it should not continue to execute. This presents an interesting area for future research.

As stated in the introduction, Insense is a language based on the π-calculus which has enabled elements of the pre-InceOS system to be verified via the SPIN model-checker. The next logical step would be to apply these techniques to InceOS in a similar manner. Considering the simpler nature of InceOS, it is expected that this model would be simpler than the one required for the pre-InceOS system, enabling more extensive verification of the system.

Through a combination of the previous three points, one of the ultimate goals for both InceOS and Insense would be an automated development process, whereby an initial design phase could be used to decide the tasks of different components, as well as their physical locations on different motes. Also, a development phase which could include automated verification of the Insense application itself to ensure properties such as channel deadlock doesn't occur, as well as assistance to the developer in the actual implementation. These phases would seem to naturally fit with inclusion into a development environment such as Netbeans, thus fully realising the goal of Insense to remove the barrier present for non-experts to author complex, concurrent, real-time, resource-constrained applications for wireless sensor networks.

Chapter 10

Conclusion

This dissertation has described InceOS, an operating system which has been successfully created to meet the needs of the Insense language.

In Chapter 2 a scarcity of research was identified in relation to language specific operating systems for embedded devices. Instead, the focus for this particular area was either generic systems which also operate on embedded devices, or systems specifically for embedded devices that offer unintuitive, complex, and possibly irrelevant programming interfaces. In contrast, InceOS is tailored to the needs of the embedded platform, and offers a simple and tailored programming interface for the Insense compiler to generate C representations of Insense programs.

Chapter 8 shows empirically that InceOS is smaller, faster and more efficient in terms of both consumed space and speed, by comparison with the pre-InceOS system. As discussed in the previous chapter, there is still room for improvement in the OS, as well as areas for future research.

There is a noticeable growing trend for the creation of domain specific languages to solve problems, with these languages being compiled into the intermediate form of an existing system such as Java [29]. Considering the time scale of three months for design, implementation and testing of this project by one individual, it seems likely that for embedded systems, the authoring of custom operating systems is a viable option in the future.

This project has been a success, and will enable the growth of Insense as a programming language, as it is now animated by a much more feasible real-time system, enabling the resource-constrained motes to achieve an almost limitless potential.

Bibliography

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[2] Gregory D. Benson and Ronald A. Olsson. A portable run-time system for the sr concurrent programming language. In *In Proceedings of the Workshop on Run-Time Systems for Parallel Processing. IR-417, Department of Mathematics and Computer Science, Vrije Universiteit*, Geneva, Switzerland, April 1997.

[3] Luc Bläser. A component-orientated language for pointer-free parallel programming. Master's thesis, Computer Systems Institute, ETH Zürich, August 2007.

[4] Luc Bläser. A high-performance operating system for structured concurrent programs. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5, New York, NY, USA, 2007. ACM.

[5] Roy H. Campbell, Nayeem Islam, David Raila, and Peter Madany. Designing and implementing choices: an object-oriented system in c++. *Commun. ACM*, 36(9):117–126, 1993.

[6] Moteiv Corporation. *T-Mote sky manual*. Motiev Corporation, Moteiv Corporation, 55 Hawthorne St, Suite 550, San Francisco, CA 94105, June 2006. http://www.eecs.harvard.edu/(tilda)konrad/projects/shimmer/references/tmote-sky-datasheet.pdf , Accessed 19/11/2009.

[7] L. A. Crowl. Concurrent data structures and actor programming under the matroshka model. In *OOPSLA/ECOOP '88: Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 79–80, New York, NY, USA, 1988. ACM.

[8] Alan Dearle, Dharini Balasubramaniam, Jonathan Lewis, and Ron Morrison. A component-based model and language for wireless sensor network applications. In *COMPSAC '08: Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications Conference*, pages 1303–1308, Washington, DC, USA, 2008. IEEE Computer Society.

[9] Sean Dorward, Rob Pike, David Leo Presotto, Dennis Ritchie, Howard Trickey, and Phil Winterbottom. Inferno. In *COMPCON '97: Proceedings of the 42nd IEEE International Computer Conference*, page 241, Washington, DC, USA, 1997. IEEE Computer Society.

[10] Adam Dunkels. Rime — a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.

[11] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[13] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: a substrate for kernel and language research. In *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, New York, NY, USA, 1997. ACM.

[14] Bryan Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the fluke kernel. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 101–115, Berkeley, CA, USA, 1999. USENIX Association.

[15] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The *nesC* language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, volume 38, pages 1–11, New York, NY, USA, May 2003. ACM.

[16] K John Gough. Stacking them up: a comparison of virtual machines. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture*, pages 55–61, Washington, DC, USA, 2001. IEEE Computer Society.

[17] Jürg Gutknecht. Do the fish really need remote control? a proposal for self-active objects in oberon. In *JMLC '97: Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pages 207–220, London, UK, 1997. Springer-Verlag.

[18] Jennifer Hamilton. Language integration in the common language runtime. *SIGPLAN Not.*, 38(2):19–28, 2003.

[19] Paul Harvey. Xenocontiki. Technical report, The University of Glasgow, Glasgow, Scotland, April 2008.

[20] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGOPS Oper. Syst. Rev.*, 34(5):93–104, 2000.

[21] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[22] Alexandros Koliousis. *An elementary proposition on the dynamic routing problem in wireless networks of sensors*. PhD thesis, University of Glasgow, Department of Computing Science, 2010.

[23] Justin T. Maris, Matthew D. Roper, and Ronald A. Olsson. Descartes: A run-time system with sr-like functionality for programming a network of embedded systems. *Computer Languages, Systems & Structures*, 29(4):75–100, 2003.

[24] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[25] Sun Microsystems. The java hotspot virtual machine. White Paper, Sun Microsystems, 2001. Accessed on 23/11/09.

[26] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[27] Pieter Muller. A multiprocessor kernel for active object-based systems. In *JMLC '00: Proceedings of the Joint Modular Languages Conference on Modular Programming Languages*, pages 263–277, London, UK, 2000. Springer-Verlag.

[28] Chris Porthouse. Jazelle dbx technology: Arm acceleration technology for the java platform. Technical report, ARM, 2005.

[29] Arno Puder, Sascha Haeberling, and Rainer Todtenhoefer. An mda approach to byte code level cross-compilation. In *SNPD '08: Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 251–256, Washington, DC, USA, 2008. IEEE Computer Society.

[30] Steven H. Rodrigues, Thomas E. Anderson, and David E. Culler. High-performance local area communication with fast sockets. In *ATEC '97: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 20–20, Berkeley, CA, USA, 1997. USENIX Association.

[31] Oliver Sharma, Jonathan Lewis, Alice Miller, Al Dearle, Dharini Balasubramaniam, Ron Morrison, and Joe Sventek. Towards verifying correctness of wireless sensor network applications using insense and spin. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software*, pages 223–240, Berlin, Heidelberg, 2009. Springer-Verlag.

[32] Crossbow Technologies. Product description of the imote2 hardware platform. Web Site, 2009. http://www.xbow.com/Products/productdetails.aspx?sid=253 , Accessed on 29/03/2010.

[33] M. Weiser, A. Demers, and C. Hauser. The portable common runtime approach to interoperability. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles*, pages 114–122, New York, NY, USA, 1989. ACM.

Appendix A

Examples

The goal of this chapter is to show an example of how an Insense language program is turned into an equivalent C language implementation for use with the InceOS.

The program itself is collecting data from the sensors (available to the t-mote sky) and then printing the readings on standard output, as well as reflecting any changes by activating the corresponding leds.

A.1 Insense Language Version

A.2 Compiled C Implementation

Appendix B

InceOS Virtual Machine Specification